Distribution Category:
Mathematics and Computer
Science (UC-405)

ANL-91/34

ANL--91/34

DE92 006819

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439-4801

PROCEEDINGS OF THE WORKSHOP ON

COMPILATION OF (SYMBOLIC) LANGUAGES FOR PARALLEL COMPUTERS

held October 31 — November 1, 1991
San Diego, CA

compiled by

*Ian Foster and Evan Tick**

Mathematics and Computer Science Division

November 1991

*Current address: University of Oregon, Eugene, Oregon

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Agenda

THURSDAY, OCTOBER 31

1:20 Welcome and Introductions

1:30 Interfacing Performance Measurement Capabilities into a Parallel Language Compiler
Carl Kesselman

2:00 Message-oriented Parallel Implementation of Flat GHC
Kazunori Ueda

2:30 An Overview of the Fortran D Programming System
Charles Koelbel

3:00 Break

3:30 OSCAR Fortran Compiler
Hironori Kasahara

4:00 Coordination Language Design and Implementation Issues
Steve Lucco

4:30 Break

5:00 Designing Imperative Programming Languages for Analyzability: Parallelism and Pointer Structures
Laurie Hendren

5:30 Compile-Time Parallelization of Prolog
Hakan Millroth

6:00 Compiling Crystal for Massively Parallel Machines
Young-il Choi

FRIDAY, NOVEMBER 1

8:30 A New Method for Compile-Time Granularity Analysis
Evan Tick

9:00 GST: Grain-Size Transformations for Efficient Execution of Symbolic Programs
Andrew Chien

9:30 Break

10:00 Using Domain-Specific, Abstract Parallelism
Ira Baxter

10:30 Applying Abstract Interpretation to Identify Numerical Code in Logic Programs
Arvind Bansal

11:00 Break

11:30 Data Locality
Monica Lam

12:00 Compiling FP for Data-Parallel Systems
Clifford Walinsky

12:30 Lunch

1:30 Improving Compilation of Implicit Parallel Programs by Using Runtime Information
John Sargeant

2:00 Generalized Iteration Space and the Parallelization of Symbolic Programs
Luddy Harrison

2:30 Break

3:00 Dataflow Analysis of Concurrent Logic Languages
Will Winsborough

3:30 Compiler Support for the Refinement and Composition of Process Structures
Ian Foster

4:00 General Discussion and Conclusion

Contents

Proceedings of the Workshop on
Compilation of (Symbolic) Languages for Parallel Computers
held October 31 – November 1, 1991
San Diego, CA

compiled by

Ian Foster and Evan Tick

Abstract

This report comprises the abstracts and papers for the talks presented at the Workshop on Compilation of (Symbolic) Languages for Parallel Computers, held October 31—November 1, 1991, in San Diego. These unrefereed contributions were provided by the participants for the purpose of this workshop; many of them will be published elsewhere in peer-reviewed conferences and publications.

Our goal in planning this workshop was to bring together researchers from different disciplines with common problems in compilation. In particular, we wished to encourage interaction between researchers working in compilation of symbolic languages (especially logic and functional programming) and those working on compilation of conventional, imperative languages.

The fundamental problems facing researchers interested in compilation of logic, functional, and procedural programming languages for parallel computers are essentially the same. However, differences in the basic programming paradigms have led to different communities emphasizing different aspects of the parallel compilation problem. For example, parallel logic and functional languages provide dataflow-like formalisms in which control dependencies are unimportant. Hence, a major focus of research in compilation has been on techniques that try to infer when sequential control flow can safely be imposed. Granularity analysis for scheduling is a related problem. The single-assignment property (central to the dataflow model) leads to a need for analysis of memory use in order to detect opportunities for reuse. Much of the work in each of these areas relies on the use of abstract interpretation techniques.

In contrast, research in procedural languages has emphasized the problem of inferring data dependencies in order to determine when sequential control flow can safely be relaxed. A related area of research is the automatic partitioning and distribution of data structures. This topic has not been addressed in the logic and functional programming communities but is important on large-scale parallel computers.

There is clearly both a commonality of interests between researchers in these different fields and large differences in emphasis and techniques. This workshop was a first step at opening up discussions between the researchers and contributing to the solution of problems in language compilation for parallel computers.

Interfacing Performance Measurement Capabilities into a Parallel Language Compiler

Carl Kesselman, Caltech

When developing a parallel program, one is ultimately interested in how effectively that program uses the parallel computer on which it runs. In this sense, identifying and eliminating performance bottlenecks is central to parallel computing. The number of times a procedure executes on a processor, or how much time is spent waiting for interprocessor communication are typical of the types of information needed to identify and eliminate performance bottlenecks. A means for measuring quantities such as these is essential to any practical parallel programming system. Recognizing this, facilities for performance measurement have been integrated into the programming environment for Program Composition Notation (PCN), a parallel programming language based on program composition, single assignment variables, and recursively defined data structures.

Facilities for performance measurement were designed into the PCN implementation from the beginning. The approach we have taken combines novel measurement techniques with statistical performance models to provide performance measurements with extremely low overhead. In this talk, we will discuss issues in the design of performance measurement systems for parallel programs and we will show how these issues have been addressed in PCN. An overview of the implementation of performance measurement in PCN will be given, paying particular attention to the influence measurement had on the design of the PCN implementation and the PCN compiler.

Message-Oriented Parallel Implementation of Moded Flat GHC (Extended Abstract)

Kazunori Ueda
Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
Masao Morita
Mitsubishi Research Institute
3-6, Otemachi 2-chome, Chiyoda-ku, Tokyo 100, Japan

Abstract. We proposed in [UM90] a new, *message-oriented* implementation technique for Moded Flat GHC that compiles unification for data transfer into message passing. The technique was based on constraint-based program analysis that was amenable to separate compilation, and significantly improved the performance of programs that used goals and streams to implement reconfigurable data structures. In this paper we discuss how the technique can be parallelized. We focus on the *shared-goal method* for shared-memory multiprocessors, though a different scheme could be used for distributed-memory multiprocessors. Unlike other parallel implementations of concurrent logic languages which we call *process-oriented*, the unit of parallel execution is not an individual goal but a chain of message sends caused successively by an initial message send. Parallelism comes from the existence of different chains of message sends that can be executed independently or in a pipelined manner. Mutual exclusion based on busy waiting and on message buffering controls access to individual, shared goals. Typical goals allow *last-send optimization*, the message-oriented counterpart of last-call optimization. We are building an experimental implementation on Sequent Symmetry. In spite of the simple scheduling currently adopted, preliminary evaluation shows good parallel speedup and good absolute performance for concurrent operations on binary process trees.

1. Introduction

Concurrent processes can be used both for programming computation and for programming storage. The latter aspect can be exploited in concurrent logic programming to program reconfigurable data structures using the following analogy:

$$\begin{array}{rcl} \text{records} & \longleftrightarrow & \text{(body) goals} \\ \text{pointers} & \longleftrightarrow & \text{streams (implemented by lists),} \end{array}$$

where a process is said to be *implemented* by a multiset of goals.

An advantage of using processes for this purpose is that it allows implementations to exploit parallelism between operations on the storage. For instance, a search operation on

```
nt([],                  _, _, L,R) :- true | L=[], R=[].
nt([search(K,V)|Cs],K, V1,L,R) :- true | V=V1, nt(Cs,K,V1,L,R).
nt([search(K,V)|Cs],K1,V1,L,R) :- K<K1 | L=[search(K,V)|L1], nt(Cs,K1,V1,L1,R).
nt([search(K,V)|Cs],K1,V1,L,R) :- K>K1 | R=[search(K,V)|R1], nt(Cs,K1,V1,L,R1).
nt([update(K,V)|Cs],K, _, L,R) :- true | nt(Cs,K,V,L,R).
nt([update(K,V)|Cs],K1,V1,L,R) :- K<K1 | L=[update(K,V)|L1], nt(Cs,K1,V1,L1,R).
nt([update(K,V)|Cs],K1,V1,L,R) :- K>K1 | R=[update(K,V)|R1], nt(Cs,K1,V1,L,R1).

t([]               ) :- true | true.
t([search(_,V)|Cs]) :- true | V=undefined, t(Cs).
t([update(K,V)|Cs]) :- true | nt(Cs,K,V,L,R), t(L), t(R).
```

Program 1. A GHC program defining binary trees as processes

a binary search tree (Program 1), given as a message in the interface stream, can enter the tree soon after the previous operation has passed the root of the tree. Programmers do not have to worry about mutual exclusion, which is taken care of by the implementation. This suggests that the programming of reconfigurable data structures can be an important application of concurrent logic languages. (The verbosity of Program 1 is a separate issue which is out of the scope of this paper.)

Processes as storage are almost always suspending, but should respond quickly when messages are sent. However, most implementations of concurrent logic languages have not been tuned for processes with this characteristic. In our earlier paper [UM90], we proposed a *message-oriented* scheduling of goals for sequential implementation, which optimizes goals that suspend and resume frequently. Although our primary goal was to optimize storage-intensive (or more generally, demand-driven) programs, the proposed technique worked quite well also for computation-intensive programs that did not use one-to-many communication. However, how to utilize the technique in parallel implementation was yet to be studied.

Parallelization of message-oriented scheduling can be quite different from parallelization of ordinary, *process-oriented* scheduling. An obvious way of parallelizing process-oriented scheduling is to execute different goals on different processors. In message-oriented scheduling, the basic idea should be to execute different message sends on different processors, but many problems must be solved as to the mapping of computation to processors, mutual exclusion, and so on. This paper reports the initial study on the subject.

The rest of the paper is organized as follows: Section 2 reviews Moded Flat GHC, the subset of GHC we are going to implement. Section 3 reviews message-oriented scheduling for sequential implementation. Section 4 discusses how to parallelize message-oriented scheduling. Of the two possible methods suggested, Section 5 focuses on the shared-goal method suitable for shared-memory multiprocessors and discusses design issues in more detail. Section 6 shows the result of the preliminary performance evaluation. The readers are

assumed to be familiar with concurrent logic languages [S89].

2. Moded Flat GHC and Constraint-Based Program Analysis

Moded Flat GHC [UM90] is a subset of GHC that introduces a *mode system* for the compile-time analysis of dataflow caused by unification. Unification can cause bidirectional dataflow in general. Without static analysis, the bidirectionality requires more runtime checks in compiled code and can cause the failure of unification.

However, our experience with GHC and KL1 (Flat GHC augmented with constructs for controlling parallel execution [UC90]) has shown that the full functionality of bidirectional unification is seldom used and that programs using it can be rewritten rather easily (if not automatically) to programs using unification as assignment. Actually, GHC is being used as a general-purpose concurrent language, which means that the efficiency of commonplace operations is more important than the efficiency of specific complex operations. Its implementations should not be too inefficient compared with those of imperative languages. Local and global compile-time analysis is thus very important to reduce the number of runtime checks and obtain machine codes close to those obtained from imperative programs.

For global compile-time analysis to be practical, it is highly desirable that the analysis can be made separately for individual program modules in such a way that the results can be merged later. The mode system of Moded Flat GHC is thus constraint-based; that is, the mode of a whole program can be determined by accumulating the mode constraints obtained separately from each program clause. The mode constraints for each clause are given by a set of syntactic rules (described in [UM90]) each applicable to a variable or an occurrence of function symbols in the clause. Another advantage of the constraint-based system is that it allows programmers to *declare* some of the mode constraints, in which case the analysis works as mode checking as well as mode inference.

The modularity of the analysis was brought by the rather strong assumption of the mode system: whether the function symbol at some position (possibly deep in the structure) of a goal g is determined by g or by other goals running concurrently is determined solely by that position specified by a *path*, which is defined as follows: Let *Pred* be the set of predicate symbols and *Fun* the set of function symbols (we do not distinguish between constants and function symbols). For each $p \in Pred$ with the arity n_p, let N_p be the set $\{1, 2, \ldots, n_p\}$. N_f is defined similarly for each $f \in Fun$. Now the sets of *paths* P_t (for terms) and P_a (for atoms) are defined using disjoint union as:

$$P_t = (\sum_{f \in Fun} N_f)^*, \quad P_a = (\sum_{p \in Pred} N_p) \times P_t.$$

An element of P_a can be written as a string $(p, i)(f_1, j_1) \ldots (f_n, j_n)$, that is, it records the predicate and the function symbols on the way as well as the argument positions selected.

A mode is a function from P_a to the set $\{in, out\}$, which means that it assigns either of *in* or *out* to every possible position of every possible instance of every possible goal. Because a path records the predicate and the function symbols on it, whether some position is *in* or *out* can depend on the predicate and function symbols on the path down to that position.

Mode analysis tries to guarantee that unification in clause body is used as assignment. For that purpose, it checks if every variable generated in the course of execution has exactly one *out* occurrence (occurrence at an *out* position) that can determine its top-level value, by accumulating constraints between the modes of different paths. The purpose of the analysis is to obtain *partial* information on the mode sufficient for compilation; it does not aim to compute a single mode, because the mode of many uninteresting positions that will not come to exist will be unconstrained and can be left undefined. The mode information can be used for compiling unification as assignment further into message passing.

Constraint-based analysis can be applied to analyze other properties of programs as well. For instance, if we can assume that streams and non-stream data structures do not occur at the same position of different goals, we can try to classify all the positions into

(1) those whose top-level values are limited to the list constructors (*cons* and *nil*) and
(2) those whose top-level values are limited to symbols other than the list constructors,

which is the simplest kind of type inference. Other applications include the static identification of 'single-reference' positions, namely positions whose values are not read by more than one goal and hence can be discarded or destructively updated after use. This could replace the MRB (multiple-reference bit) scheme [CK87], a runtime scheme adopted in current KL1 implementations for the same purpose.

3. Message-Oriented (Sequential) Implementation

Message-oriented implementation compiles the generation of stream elements into procedure calls to the consumer of the stream. A stream is an unbounded buffer of messages in principle, but message-oriented implementation tries to reduce the overhead of buffering and unbuffering by transferring control and messages simultaneously to the receiver whenever possible. To this end, it tries to schedule goals so that whenever the producer of a stream sends a message, the consumer is suspending on the stream and is ready to handle the message. Of course, this is not always possible because we can write a program in which a stream *must* act as a buffer; messages are buffered in that event.

Process-oriented implementation tries to achieve good performance by reducing the frequency of goal switching and taking advantage of last-call optimization. Message-oriented implementation tries to reduce the cost of each goal switching operation and the cost of data transmission between goals.

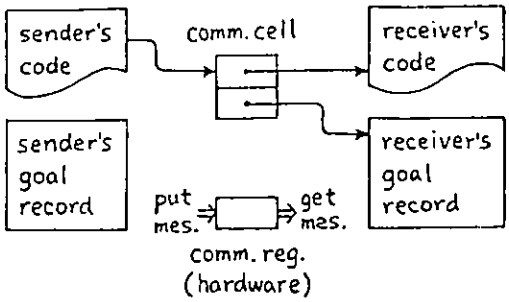


Fig. 1 Immediate message send

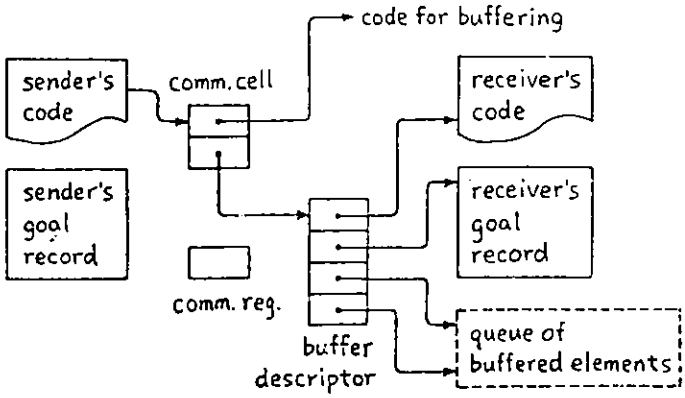


Fig. 2 Buffered message send

Suppose two goals, p and q, are connected by a stream s and p is going to send a message to q. Message-oriented implementation represents s as a two-field *communication cell* that points to (1) the instruction in q's code from which the processing of q is to be resumed and (2) q's goal record containing its arguments (Fig. 1). To send a message m, p first loads m on a hardware register called the *communication register*, change the current goal to the one pointed to by the communication cell of s, and call the code pointed to by the communication cell of s. The goal q gets m from the communication register and may send other messages in its turn. Control returns to p when all the message sends caused directly or indirectly by m have been processed. However, if m is the last message which p can send out immediately (i.e., without waiting for further incoming messages), control need not return to p but can go directly to the goal that has outstanding message sends. This is called *last-send optimization.*

We have observed in GHC/KL1 programming that the dominant form of interprocess communication is one-to-one stream communication. It therefore deserves special treatment, even though other forms of communication such as broadcasting and multicasting become a little more expensive. One-to-many communication is done either by the repeated sending of messages or by using non-stream data structures.

Techniques mentioned in Section 2 are used to analyze which positions of a predicate and which variables in a program are used for streams and to distinguish between the sender and the receiver(s) of messages.

When a stream must buffer messages, the communication cell representing the stream points to the code for buffering and the descriptor of a buffer. The old entries of the communication cell are saved in the descriptor (Fig. 2). In general, a stream must buffer incoming messages when the receiver goal is not ready to handle them. The following are the possible reasons [UM90]:

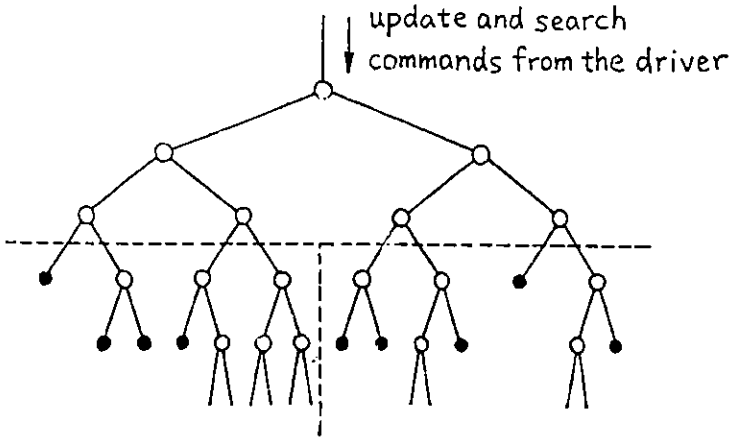


Fig. 3 Binary search tree as a process

(1) (selective message receiving) The receiver is waiting for a message from other input streams.
(2) The receiver is suspending on non-stream data (possibly the content of a message).
(3) The sender of a message may run ahead of the receiver.
(4) When the receiver r belongs to a circular process structure, a message m sent by r may possibly arrive at r itself or may cause another message to be sent back to r. However, unless m has been send by last-send optimization, r is not ready to receive it.

The receiver examines the buffer when the reason for the buffering disappears, and handles messages (if any) in it.

4. Parallelization

How can we exploit parallelism from message-oriented implementation? Two quite different methods can be considered:

Distributed-goal method. Different processors take charge of different goals, and each processor handles messages sent to the goals it is taking charge of. Consider a binary search tree represented using goals and streams (Fig. 3) and suppose three processors take charge of the three different portions of the tree. Each processor performs message-oriented processing within its own portion, while message transfer between portions is compiled into inter-processor communication with buffering.

Shared-goal method. All processors share all the goals. There is a global, output-restricted deque [K73] of outstanding work, from which an idle processor gets a new job. The job is usually to resume the execution of the body goals of a clause. If it is a message send followed by the rest of the work for the clause, the processor performs the message send and subsequent message sends it causes, putting the rest of the work back to the top of the deque. This allows different chains of message sends to be performed in parallel. In the binary tree example, different processors will take care of different operations sent to the

root. A tree operation may cause subsequent message sends inside the tree, but they should be performed by the same processor because there is no parallelism within each operation.

Unlike the shared-goal method, the distributed-goal method can be applied to distributed-memory multiprocessors as well as shared-memory ones to improve the throughput of message handling. On shared-memory multiprocessors, however, the shared-goal method is more advantageous in terms of latency (i.e., responses to messages), because (1) it performs no inter-processor communication within a chain of message sends and (2) good load balancing can be attained easily. The shared-goal method requires a locking protocol for goals as will be discussed in Section 5.1, but enables more tightly-coupled parallel processing that covers a wider range of applications. Because of its greater technical interest, the rest of the paper is focused on the shared-goal method.

5. Shared-Goal Implementation

In this section, we discuss new technicalities in implementing the shared-goal method. Space limitations do not allow the full description of the implementation, so we choose to use examples to explain our intermediate code.

5.1 Locking of Goals

Consider a goal `p(Xs,Ys)` defined by the single clause:

```
p([A|Xs1],Ys) :- true | Ys=[A|Ys1], p(Xs1,Ys1).
```

In the shared-goal method, different messages in the input stream `Xs` may be handled by different processors that share the goal `p(Xs,Ys)`. Any processor sending a message must therefore try to lock the goal record (placed in the shared memory) of the receiver first and obtain grant for the exclusive access to it. The receiver must remain locked until it sends a message through `Ys` and restores the dormant state.

The locking operation is important in the following respect as well: In message-oriented implementation, the order of the elements in a stream is not represented spatially as a list structure but as the chronological order of message sends. The locking protocol must therefore make sure that when two messages, α and β, are sent in this order to `p(Xs,Ys)`, they are sent to the receiver of `Ys` in the same order. This is guaranteed by locking the receiver of `Ys` *before* `p(Xs,Ys)` is unlocked.

5.2 Busy Wait vs. Suspension

How should a processor trying to send a message wait until the receiver goal is unlocked? The two extreme possibilities are (1) to spin (busy wait) until unlocked and (2) to give up

(suspend) the sending immediately and do some other work, leaving the notice to the receiver that it has a message to receive. We must take the following observations into account:

(a) The time each reduction takes, namely the time required for a resumed goal to restore the dormant state, is usually short (several tens of CISC instructions, say), though it can be considerably long sometimes.
(b) As explained in Section 5.1, a processor may lock more than one goal temporarily upon reduction. This means that busy wait may cause deadlock when goals and streams form a circular structure.

Because busy wait incurs much smaller overhead than suspension, Observation (a) suggests that the processor should spin for a period of time within which most goals can perform one reduction. However, it should suspend finally because of (b).

Upon suspension, a buffer is prepared as in Fig. 2, and the unsent message is put in it. Subsequent messages go to the buffer until the receiver has processed all the messages in the buffer and has removed the buffer. As is evident from Fig. 2, no overhead is incurred to check if the message is going to the buffer or to the receiver. The receiver could notice the existence of the outstanding messages by checking its input streams upon each reduction, but it incurs overhead to (normal) programs which don't require buffering. So we have chosen to let the *sender* schedule the *retransmitter* of the messages when it creates a buffer. The retransmitter occasionally tests if the receiver has been unlocked, in which case it sends the first message in the buffer and re-schedules itself.

For the shared resources other than goals (such as logic variables and the global deque), mutual exclusion should be attained by busy wait, because access to them takes a short period of time. On the other hand, synchronization on the values of non-stream variables (due to the semantics of GHC) should be implemented using suspension as usual.

5.3 Scheduling

Shared-goal implementation exploits parallelism between different chains of message sends that do not interfere with each other. For instance, the binary search tree in Fig. 3 can process different operations on it in a pipelined manner, as long as there is no dependency between the operations (e.g., the key of a search operation depending on the result of the previous search operation). When there is dependency, however, parallel execution can even lower the performance because of synchronization overhead.

Another example for which parallelism does not help is a demand-driven generator of prime numbers which is made up of cascaded goals for filtering out the multiples of prime numbers. The topmost goal receiving a new demand from outside filters out the multiples of the prime computed in response to the last demand. However, it doesn't know what

prime's multiples should be filtered out, and hence will be blocked, until the last demand has almost been processed.

These considerations suggest that in order to avoid ineffective parallelism, it is most realistic to let programmers specify which chains of message sends should be done in parallel with others and which should not. The simple method we are using currently is to have (1) a global deque for the work to be executed in parallel by idle processors and (2) one local stack for each processor for the work to be executed sequentially by the current processor. Each processor obtains a job from the global deque when its local stack is empty. We use a global deque rather than a global stack because, if the retransmitter of a buffer fails to send a message, it must go to the tail of the deque so it may not be retried soon.

Each job in a stack/deque is uniformly represented as a pair $\langle code, env \rangle$, where *code* is the job's entry/resumption point and *env* is its environment. The job is usually to start the execution of a goal or to resume the reduction of a goal, in which case *env* points to the goal record on which *code* should work. When the job is to retransmit buffered messages, *env* points to the communication cell pointing to the buffer.

5.4 Reduction

This section outlines what a typical goal should do during one reduction, where by 'typical' we mean goals that can be reduced by receiving one incoming message. As an example, consider the distributor of messages defined as follows,

```
p([A|Xs],Ys,Zs) :- true | Ys=[A|Ys1], Zs=[A|Zs1], p(Xs,Ys1,Zs1).
```

where A is assumed *not* to be a stream. The unoptimized intermediate code for above program is:

```
entry(p/3)
  rcv_value(A1)
  get_cr(A4)
  send_call(A2)
  put_cr(A4)
  send_call(A3)
  execute.
```

The A*i*'s are the arguments of a goal and temporary variables to be recorded in the goal record. Other programs may use X*i*'s, which are (possibly virtual) general registers local to each processor. The label `entry(p/3)` indicates the initial entry point of the predicate.

The instruction `rcv_value(A1)` waits for a message from the input stream at the first argument. If messages should be already buffered, it takes the first one and put it on the communication register. A retransmitter of the buffer is put on the deque if more messages exist; otherwise the buffer is made to disappear (Section 5.7). If no messages are buffered,

which is expected to be most probable, `rcv_value` records the address of the next instruction in the communication cell, unlocks the goal record, and suspends until a message arrives. The goal is usually suspending at this instruction.

The instruction `get_cr(A4)` saves into the goal record the message in the communication register, which the previous `rcv_value` has received. Then `send_call(A2)` sends the message through the second stream. Control is transferred to the receiver of the stream unless the stream is being buffered. When control eventually returns, `put_cr(A4)` restores the communication register and `send_call(A3)` sends the next message.

When control returns again, `execute` performs the recursive call by going back to the entry point of the predicate p. Then the `rcv_value(A1)` instruction either finds buffered messages or finds nothing. In the former case, a retransmitter of the buffer must have been scheduled. So `rcv_value(A1)` can suspend until the retransmitter sends a message. In the latter case, the recursive call is suspended at the same instruction as the last time. Thus in either case, `execute` effectively does nothing but unlocking the current goal. This is why last-send optimization can replace the last two instructions into a single instruction, `send_jmp(A3)`. The instruction `send_jmp(A3)` locks the receiver of the third stream, unlocks the current goal, and transfers control to the receiver without stacking the return address. Last-send optimization enables the current goal to receive the next message earlier and allows the pipelined processing of message sends.

The above instruction sequence performs the two message sends sequentially. However, a variant of `send_call` called `send_gcall` stacks the return address on the global deque instead of the local stack, allowing the continuation to be processed in parallel.

We have established a code generation scheme for general cases including the spawning and the termination of goals (Section 5.5), explicit control of message buffering (Section 5.6), and suspension on non-stream variables. Several optimization techniques have been developed for goals with a single input stream and for goals whose input streams are known to carry messages of limited forms (e.g., non-root nodes of a binary search tree (Fig. 3)). Finally, we note that although process-oriented scheduling and message-oriented scheduling differ in the flow of control, they are quite compatible in the sense that an implementation can use both in running a single program. Our experimental implementation has actually been made by modifying the process-oriented implementation.

5.5 Examples

Here we give the intermediate code of the naïve reverse program (Fig. 4). In order for the code to be almost self-explanatory, some comments are appropriate here.

Suppose the messages m_1, ..., m_n are sent to the goal `nreverse(In,Out)` through `In`, followed by the *eos* (end-of-stream) message indicating that the stream is closed. The

```
nreverse([H|T],O) :- true | append(O1,[H],O), nreverse(T,O1).        (1)
nreverse([],   O) :- true | O=[].                                    (2)
append([I|J],K,L) :- true | L=[I|M], append(J,K,M).                  (3)
append([],   K,L) :- true | K=L.                                     (4)
```

```
entry(nreverse/2)
  rcv_value(A1)                  receive a message from the 1st arg (the program is
                                 usually waiting for incoming messages here)
  check_not_eos(101)             if the message is eos then collect the current
                                 comm. cell and goto 101
  get_cr(X3)                     save the message H in the comm. reg. to the
                                 register of the current PE
  commit                         Clause 1 is selected (no operation)
  put_cc(X4)                     create a comm. cell with a buffer
  push_value(X3,X4)              put the message H into the buffer
  push_eos(X4)                   put eos into the buffer
  gr_setup(append/3,3)           create a goal record for 3 args and record the name
  put_com_variable(X3,sarg(1))   create a locked variable O1 and set it to
                                 X3 and the 1st arg of append/3
  put_value(X4,sarg(2))          set [H]
  put_value(A2,sarg(3))          set O
  call_proc                      execute append/3 until it suspends
  put_value(X3,A2)               set O1
  return                         unlock the current-goal and do the job on the
                                 local stack top
label(101)
  commit                         Clause 2 is selected (no operation)
  send_call(A2)                  send eos in the comm. reg. to the receiver of O
  proceed                        deallocate the goal record and return
entry(append/3)
  deref(A3)                      dereference the 3rd arg L
  rcv_buffer(A2)                 make sure that the 2nd arg K buffers messages
  rcv_value(A1)                  receive a message from the 1st arg.
  check_not_eos(102)             if the message is eos then collect the current
                                 comm. cell and goto 102.
  commit                         Clause 3 is selected (no operation)
  sendn_jmp(A3)                  send the received message to the receiver of L,
                                 assuming that L has been dereferenced
label(102)
  commit                         Clause 4 is selected (no operation)
  send_unify_jmp(A2,A3)          make sure that messages sent through A2 are
                                 forwarded to the receiver of A3, and return
```

Fig. 4 Intermediate code for naïve reverse

number B coming through the second stream. Suppose B(> A) arrives and the first clause commits. Then the second stream should become a buffer and B will be put back. The first stream, now being a buffer, is checked and a retransmitter is stacked if it contains an element; otherwise the buffer is made to disappear. Finally A is sent to the receiver of the third stream. The above procedure may look complex, but this program is indeed one of the hardest ones to execute in a message-oriented manner. A simpler example of selective message receiving appears in the concatenation of two streams, as described in Section 5.5.

Suspension on non-stream data. The most plausible case is the suspension on the content of a message (e.g., the first argument of a update command to the binary search tree). When a goal receives from a stream s a message that is not sufficiently instantiated for commitment, it changes s to a buffer and put the message back to it. The retransmitter is hooked on the uninstantiated variable(s) that caused suspension, and is invoked when any of them are instantiated.

The sender of a stream running ahead of the receiver. It is not always possible to guarantee that the sender of a stream does not send a message before the receiver commences execution, particularly when they run in parallel. A stream is initialized to a buffer in that event.

Circular process structure. When the receiver sends more than one message in response to an incoming message, the sequential implementation must buffer subsequent incoming messages until the last message is sent out. In parallel implementation, the same effect is automatically achieved by the lock on the goal record and hence the explicit control of buffering is not necessary.

The retransmission of a buffer created by the receiver of a stream is explicitly controlled by the receiver, while the retransmitter of a buffer created by the sender is scheduled asynchronously with the receiver.

5.7 Mutual Exclusion of Communication Cells

Because the communication cell for a stream may be updated both by the sender and the receiver of the stream, some method of mutual exclusion is called for. The simplest solution would be to lock a communication cell whenever accessing it, but locking both a goal record and a communication cell for each message sending would be too costly.

The solution we adopted does not incur any overhead in ordinary message sends: While the receiver is updating the communication cell, its first field temporarily points to a code that makes the sender to retry the message send. This is not yet sufficient because there is a slight possibility that

(1) the sender follows the pointer in the second field of the communication cell, and then

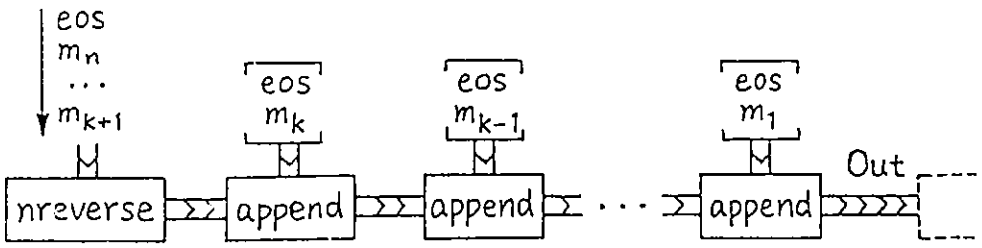


Fig. 5 Process structure being created by nreverse([m_1,...,m_n],Out)

nreverse goal generates one suspended append goal for each m_i, creating the structure in Fig. 5. The ith append has as the second argument a buffer with two messages, m_i and *eos*. The final *eos* message to nreverse causes the second clause to forward the *eos* to the most recent append goal holding m_n. The append goal, in response, lets different (if available) processors send the two buffered messages m_n and *eos* to the append holding m_{n-1}. The message m_n is transferred all the way to the append holding m_1 and appears in Out. The following *eos* causes the next append goal to send m_{n-1} and another *eos*.

The performance hinges on how fast an append goal can transfer messages. For each incoming message, it checks if the message is not *eos* and then transfers the message and control to the receiver of the output stream. The message remains on the communication register and need not be loaded or stored.

The send_unify_jmp(r_1,r_2) instruction is for the unification of two streams. If the stream r_1 has a buffer (which is the case in nreverse), its contents are first sent to the receiver of r_2. Then an arrangement is made for r_1 so that next time a message is sent through r_1, the sender is made to point directly to the communication cell of r_2.

5.6 Buffering

As discussed in Section 5.2, the producer of a stream s creates a buffer when the receiver is locked for a long time. However, this is a rather unusual situation; a buffer is usually created by s's *receiver* when it remains unready to handle incoming messages after it has unlocked itself. Here we re-examine the four reasons of buffering in Section 3:

Selective message receiving. This happens, for instance, in a program for merging two (ordered) streams of integers:

```
omerge([A|X1],[B|Y1],Z) :- A< B | Z=[A|Z1], omerge(X1,[B|Y1],Z1).
omerge([A|X1],[B|Y1],Z) :- A>=B | Z=[B|Z1], omerge([A|X1],Y1,Z1).
```

Two numbers, one from each input stream, are necessary for the reduction. Suppose the first number A arrives through the first stream. Then the goal omerge checks if the second stream has a buffered value. Since it doesn't, the goal cannot be reduced. So it records A in the goal record and change the first stream to a buffer, because it has to wait for another

(2) the receiver starts and completes the updating of the communication cell, and then
(3) the sender locks the (wrong) record obtained in Step (1) and calls the code pointed to by the first field of the updated communication cell,

which happens in the following cases:

(a) the receiver creates a buffer for the cell,
(b) the receiver removes a buffer for the cell,
(c) the goal record of the receiver is moved due to reduction, and
(d) the receiver unifies the stream which the cell represents with another stream by `send_unify_jmp`.

Case (a) is handled by the code for buffering. The other cases are handled by *not* letting the receiver update the communication cell but letting the mislocked buffer have a recovery pointer to the right goal record. The communication cell is updated by the sender when it follows the recovery pointer.

6. An Experimental System and Its Performance

We have almost finished the design of an initial version of the abstract machine instruction set for the shared-goal method. An experimental runtime system for performance evaluation has been developed on Sequent Symmetry, a shared-memory parallel computer with 20MHz 80386's The system is written an assembly language and C, and the abstract machine instructions are expanded into native codes automatically by the loader. A compiler from Moded Flat GHC to the intermediate code is yet to be developed.

The current system employs a simple scheme of parallel execution as described in Section 5.3. When the system runs with more than one processor, one of them acts as a master processor and the others as slaves. They act in the same manner while the global deque is non-empty. When the master fails to obtain a new job from the deque, it tries to detect termination and exceptions such as stack overflow. The current system does not care about perpetually suspended goals; they are treated just like garbage cells in Lisp. A slight overhead of counting the number of goals in the system would be necessary to detect perpetually suspended goal [IO90] and/or to feature the *shoen* construct of KL1 [UC90], but it would scarcely affect the result of performance evaluation described below.

Locking of shared resources, namely logic variables, goal records, communication cells, the global deque, etc., is done using the xchg (exchange) instruction as usual.

Using Program 1, we measured the processing time of 5000 `update` commands with random keys given to an empty binary tree and the processing time of 5000 `search` commands (with the same sequence of keys) to the resulting tree with 4777 nodes. The number

Table 1. Performance Evaluation (in seconds)

Language	Processing	binary process tree (5000 operations) (search)	(update)	naïve reverse (1000 elements)
GHC	1 PE (no locking)	1.25	1.83	2.23 (225 kRPS)*
	1 PE	1.38	2.10	3.27 (154 kRPS)
	2 PEs	0.78	1.15	2.43 (207 kRPS)
	3 PEs	0.55	0.81	1.71 (294 kRPS)
	4 PEs	0.44	0.63	1.33 (377 kRPS)
	5 PEs	0.36	0.53	1.10 (456 kRPS)
	6 PEs	0.33	0.46	0.96 (523 kRPS)
	7 PEs	0.33	0.39	0.85 (591 kRPS)
	8 PEs	0.33	0.36	0.77 (652 kRPS)
C (recursion)	cc -O	0.71	0.72	
C (iteration)	cc -O	0.32	0.35	

(* kilo Reductions Per Second)

of processors was changed from 1 to 8. For the one-processor case, a version without locking/unlocking operations was tested as well. The numbers include the execution time of the driver that sends messages to the tree. The result was compared with two versions of (sequential) C programs using records and pointers, one using recursion and one using iteration. The performance of `nreverse` (Fig. 4) was measured as well. The results are shown in Table 1.

The results show good (if not ideal) parallel speedup, though for `search` operations on a binary tree, the performance is finally bounded by the sequential nature of the driver and the root node. Access contention on the global deque can be another cause of overhead. Note, however, that the two examples are indeed harder to execute in parallel than running independent processes in parallel, because different chains of message sends can pass the same goal. Note also that the binary tree with 4777 nodes is not very deep.

The binary tree program run with 4 processors outperformed the optimized recursive C program. The iterative C program was more than twice as fast as the recursive one and was comparable to the GHC program run with 8 processors. The comparison, however, would have been more preferable to parallel GHC if a larger tree had been used.

The overhead of locking/unlocking is about 30% in `nreverse` and about 10% in the binary tree program. Since `nreverse` is one of the fastest programs in terms of the kRPS value, we can conclude that the overhead of locking/unlocking is reasonably small on average even if we lock such small entities as individual goals.

As for space efficiency, the essential difference between our implementation and C implementations is that GHC goal records have pointers to input streams while C records do not consume memory by being pointed to. The difference comes from the expressive

power of streams; unlike pointers, streams can be unified together and can buffer messages implicitly.

7. Conclusions and Future Works

The main contribution of this paper is that message-oriented implementation of Moded Flat GHC was shown to benefit from small-grain, tightly-coupled parallelism on shared-memory multiprocessors. Furthermore, the result of preliminary evaluation shows that the absolute performance is good enough to be compared with C programs.

These results suggest that the programming of reconfigurable storage structures that allow concurrent access can be a realistic application of Moded Flat GHC. Programmers need not worry about mutual exclusion necessitated by parallelization, because it is achieved automatically at the implementation level. In procedural languages, parallelization may well require major rewriting of programs. To our knowledge, how to deal with reconfigurable storage structures efficiently in non-procedural languages without side effects has not been studied in depth.

We have not yet fully studied the language constructs and its implementation for more minute control over parallel execution. The current scheme is a simple extension to the sequential system and is rather tentative; it worked well for the benchmark programs used, but will not be powerful enough to be able to tune the performance of large programs. We need a notion of priority that should be somewhat different from the priority construct in KL1 designed for process-oriented parallel execution. KL1 provides the *shoen* (manor) construct [UC90] as well, which is the unit of execution control, exception handling and resource consumption control. How to adapt the *shoen* construct to message-oriented implementation is another research topic.

References

[CK87] T. Chikayama and Y. Kimura, Multiple Reference Management in Flat GHC. In *Proc. 4th Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 276–293.

[IO90] Y. Inamura and S. Onishi, A Detection Algorithm of Perpetual Suspension in KL1. In *Proc. Seventh Int. Conf. on Logic Programming*, MIT Press, 1990, pp. 18–30.

[K73] D. E. Knuth, *The Art of Computer Programming, Vol. 1 (2nd ed.)*. Addison-Wesley, Reading, MA, 1973.

[S89] Shapiro, E., The Family of Concurrent Logic Programming Languages. *Computing Surveys*, Vol. 21, No. 3 (1989), pp. 413–510.

[UM90] K. Ueda and M. Morita, A New Implementation Technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming*, MIT Press, 1990, pp. 3–17. A revised, extended version to appear in *New Generation Computing*.

[UC90] K. Ueda and T. Chikayama, Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (Dec., 1990), pp. 494–500.

An Overview of the Fortran D Programming System*

Seema Hiranandani
Ken Kennedy
Charles Koelbel
Ulrich Kremer
Chau-Wen Tseng

Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

The success of large-scale parallel architectures is limited by the difficulty of developing machine-independent parallel programs. We have developed Fortran D, a version of Fortran extended with data decomposition specifications, to provide a portable data-parallel programming model. This paper presents the design of two key components of the Fortran D programming system: a prototype compiler and an environment to assist automatic data decomposition. The Fortran D compiler addresses program partitioning, communication generation and optimization, data decomposition analysis, run-time support for unstructured computations, and storage management. The Fortran D programming environment provides a static performance estimator and an automatic data partitioner. We believe that the Fortran D programming system will significantly ease the task of writing machine-independent data-parallel programs.

1 Introduction

It is widely recognized that parallel computing represents the only plausible way to continue to increase the computational power available to computational scientists and engineers. However, it is not likely to be widely successful until parallel computers are as easy to use as today's vector supercomputers. A major component of the success of vector supercomputers is the ability to write machine-independent vectorizable programs. Automatic vectorization and other compiler technologies have made it possible for the scientist to structure Fortran loops according the well-understood rules of "vectorizable style" and expect the resulting program to be compiled to efficient code on any vector machine [6, 32].

Compare this with the current situation for parallel machines. Scientists wishing to use such a machine must rewrite their programs in an extension of Fortran that explicitly reflects the architecture of the underlying machine, such as a message-passing dialect for MIMD distributed-memory machines, vector syntax for SIMD machines, or an explicitly parallel dialect with synchronization for MIMD shared-memory machines. This conversion is difficult, and the resulting parallel programs are machine-specific. Scientists are thus discouraged from porting programs to parallel machines because they risk losing their investment whenever the program changes or a new architecture arrives.

One way to overcome this problem would be to identify a "data-parallel programming style" that allows the efficient compilation of Fortran programs on a variety of parallel machines. Researchers working in the area, including ourselves, have concluded that such a programming style is useful but not sufficient in general. The reason for this is that not enough information can be included in the program text for the compiler to accurately evaluate alternative translations. Similar reasoning argues against cross-compilations between the

*This research was supported by the Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center.

current parallel extensions of Fortran.

For these reasons, we have chosen a different approach. We believe that selecting a data decomposition is one of the most important intellectual step in developing data-parallel scientific codes. However, current parallel programming languages provide little support for data decomposition [26]. We have therefore developed an enhanced version of Fortran that introduces data decomposition specifications. We call the extended language Fortran D, where "D" suggests data, decomposition, or distribution. When reasonable data decompositions are provided for a Fortran D program written in a data-parallel programming style, we believe that advanced compiler technology can implement it efficiently on a variety of parallel architectures.

We are developing a prototype Fortran D compiler to generate node programs for the iPSC/860, a MIMD distributed-memory machine. If successful, the result of this project will go far towards establishing the feasibility of machine-independent parallel programming, since a MIMD shared-memory compiler could be based directly on the MIMD distributed-memory implementation. The only additional step would be the construction of an effective Fortran D compiler for SIMD distributed-memory machines. We have initiated at Rice a project to build such a compiler based on existing vectorization technology.

The Fortran D compiler automates the time consuming task of deriving node programs based on the data decomposition. The remaining components of the Fortran D programming system, the static performance estimator and automatic data partitioner, support another important step in developing a data-parallel program—selecting a data decomposition. The rest of this paper presents the data decomposition specifications in Fortran D, the structure of a prototype Fortran D compiler, and the design of the Fortran D programming environment. We conclude with a discussion of our validation strategy.

2 Fortran D

The data decomposition problem can be approached by considering the two levels of parallelism in data-parallel applications. First, there is the question of how arrays should be *aligned* with respect to one another, both within and across array dimensions. We call this the *problem mapping* induced by the structure of the underlying computation. It represents the minimal requirements for reducing data movement for the program, and is largely independent of any machine considerations. The alignment of arrays in the program depends on the natural fine-grain parallelism defined by individual members of data arrays.

Second, there is the question of how arrays should be *distributed* onto the actual parallel machine. We call this the *machine mapping* caused by translating the problem onto the finite resources of the machine. It is affected by the topology, communication mechanisms, size of local memory, and number of processors in the underlying machine. The distribution of arrays in the program depends on the coarse-grain parallelism defined by the physical parallel machine.

Fortran D is a version of Fortran that provides data decomposition specifications for these two levels of parallelism using `DECOMPOSITION`, `ALIGN`, and `DISTRIBUTE` statements. A decomposition is an abstract problem or index domain; it does not require any storage. Each element of a decomposition represents a unit of computation. The `DECOMPOSITION` statement declares the name, dimensionality, and size of a decomposition for later use.

The `ALIGN` statement is used to map arrays onto decompositions. Arrays mapped to the same decomposition are automatically aligned with each other. Alignment can take place either within or across dimensions. The alignment of arrays to decompositions is specified by placeholders in the subscript expressions of both the array and decomposition. In the example below,

```
REAL X(N,N)
DECOMPOSITION A(N,N)
ALIGN X(I,J) with A(J-2,I+3)
```

A is declared to be a two dimensional decomposition of size $N \times N$. Array X is then aligned with respect to A with the dimensions permuted and offsets within each dimension.

After arrays have been aligned with a decomposition, the `DISTRIBUTE` statement maps the decomposition to the finite resources of the physical machine. Distributions are specified by assigning an independent *attribute* to each dimension of a decomposition. Predefined attributes are `BLOCK`,

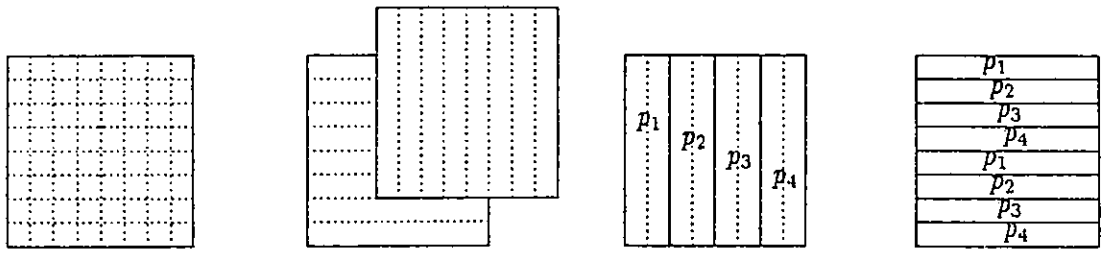


Figure 1: Fortran D Data Decomposition Specifications

CYCLIC, and BLOCK_CYCLIC. The symbol ":" marks dimensions that are not distributed. Choosing the distribution for a decomposition maps all arrays aligned with the decomposition to the machine. In the following example,

```
DECOMPOSITION A(N,N)
DISTRIBUTE A(:, BLOCK)
DISTRIBUTE A(CYCLIC,:)
```

distributing decomposition A by (:,BLOCK) results in a column partition of arrays aligned with A. Distributing A by (CYCLIC,:) partitions the rows of A in a round-robin fashion among processors. These sample data alignment and distributions are shown in Figure 1.

Predefined regular data distributions can effectively exploit regular data-parallelism. However, irregular distributions and run-time processing is required to manage the irregular data parallelism found in many unstructured computations. In Fortran D, irregular distributions may be specified through an explicit user-defined function or data array. In the example below,

```
INTEGER MAP(N)
DECOMPOSITION IRREG(N)
DISTRIBUTE IRREG(MAP)
```

elements of the decomposition IRREG(i) will be mapped to the processor indicated by the array MAP(i). Fortran D also supports dynamic data decomposition; *i.e.*, changing the alignment or distribution of a decomposition at any point in the program.

We should note that our goal in designing Fortran D is not to support the most general data decompositions possible. Instead, our intent is to provide decompositions that are both powerful enough to express data parallelism in scientific programs, and simple enough to permit the compiler to produce efficient programs. Fortran D is a language with semantics very similar to sequential Fortran. As a result, it should be quite usable by computational scientists. In addition, we believe that our two-phase strategy for specifying data decomposition is natural and conducive to writing modular and portable code. Fortran D bears similarities to both CM Fortran [31] and KALI [22]. The complete language is described in detail elsewhere [8].

3 Fortran D Compiler

As we have stated previously, two major steps in writing a data-parallel program are selecting a data decomposition, and then using it to derive node programs with explicit communications to access nonlocal data. Manually inserting communications is unquestionably the most time-consuming, tedious, non-portable, and error-prone step in parallel programming. Significant increases in source code size are not only common but expected. A major advantage of programming in Fortran D will be the ability to utilize advanced compiler techniques to automatically generate node programs with explicit communication, based on the data decompositions specified in the program. The prototype compiler is being developed in the context of the ParaScope parallel programming environment [4], and will take advantage of the analysis and

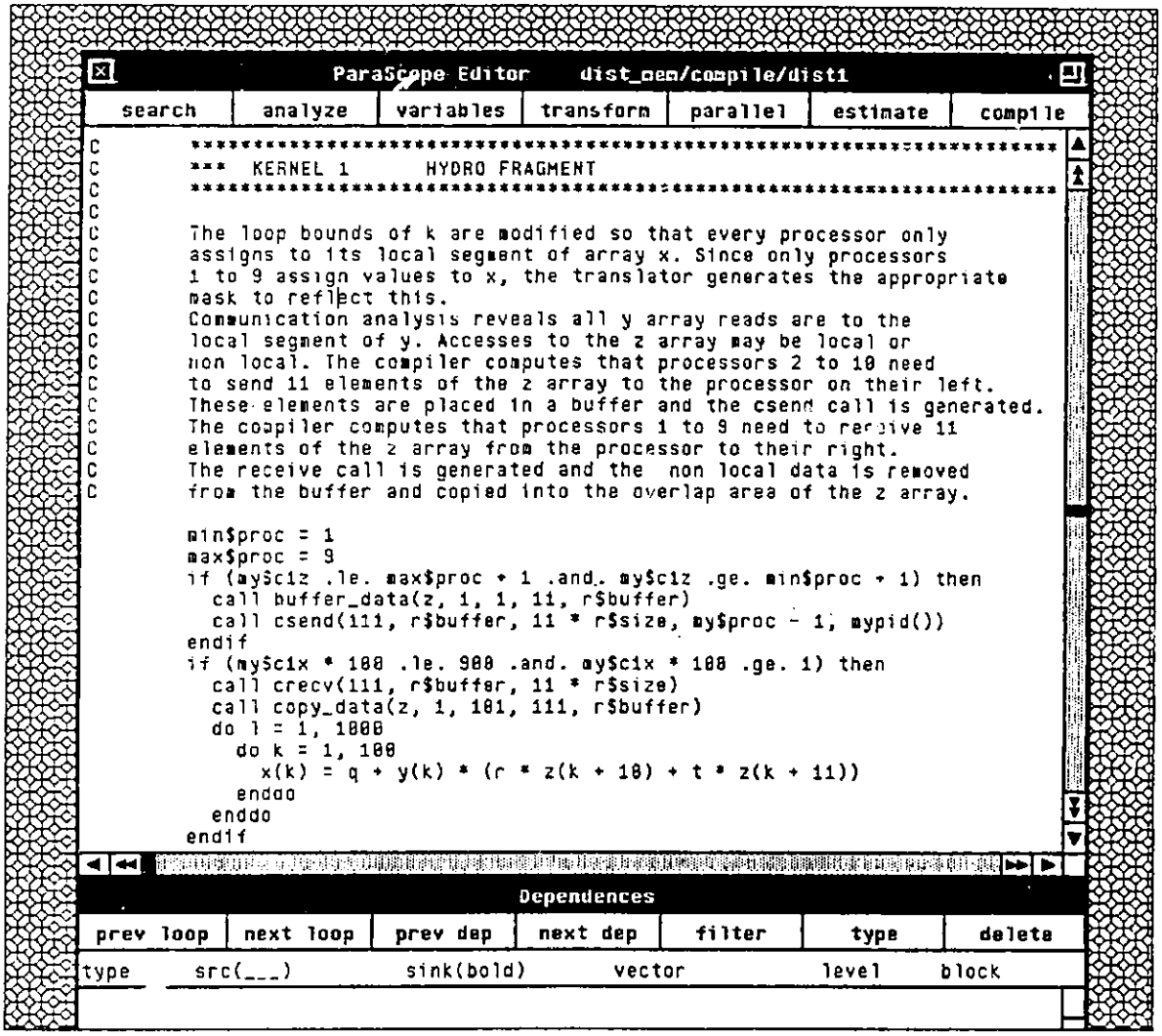


Figure 2: Fortran D Compiler Output

transformation capabilities of the ParaScope Editor [19, 20].

The main goal of the Fortran D compiler is to derive from the data decomposition a parallel node program that minimizes load imbalance and communication costs. Our approach is to convert Fortran D programs into *single-program, multiple-data* (SPMD) form with explicit message-passing that executes directly on the nodes of the distributed-memory machine. Our basic strategy is to partition the program using the *owner computes* rule, where every processor only performs computation on data it owns [5, 29, 34]. However, we will relax the rule where it prevents the compiler from achieving good load balance or reducing communication costs.

The Fortran D compiler bears similarities to ARF [33], ASPAR [18], ID NOUVEAU [29], KALI [22], MIMDIZER [13], and SUPERB [34]. The current prototype generates code for a subset of the decompositions allowed in Fortran D, namely those with BLOCK distributions. Figure 2 depicts the output of a Livermore loop kernel generated by the Fortran D compiler.

3.1 Program Partitioning

The first phase of the compiler partitions the program onto processors based on the data decomposition. We define the *iteration set* of a reference R on the local processor t_p to be the set of loop iterations that cause R to access data owned by t_p. The

iteration set is calculated based on the alignment and distribution specified in the Fortran D program. According to the *owner computes rule*, the set of loop iterations that t_p must execute is the union of the iteration sets for the left-hand sides (*lhs*) of all the individual assignment statements within the loop.

To partition the computation among processors, we first reduce the loop bounds so that each processor only executes iterations in its own set. With multiple statements in the loop, the iteration set of an individual statement may be a subset of the iteration set for that loop. For these statements we also add guards based on membership tests for the iteration set of the *lhs* to ensure that all assignments are to local array elements.

3.2 Communication Introduction

Once the computation has been partitioned, the Fortran D compiler must introduce communications for nonlocal data accesses to preserve the semantics of the original program. This requires calculating the data that must be sent or received by each processor. We can calculate the *send iteration set* for each right-hand side (*rhs*) reference as its iteration set minus the iteration set of its *lhs*. Similarly, the *receive iteration set* for each *rhs* is the iteration set of its *lhs* minus its own iteration set. These sets represent the iterations for which data must be sent or received by t_p. The Fortran D compiler summarizes the array locations accessed on the send or receive iterations using rectangular or triangular regions known as *regular sections* [12]; they are used to generate calls to communication primitives.

3.3 Communication Optimization

A naive approach for introducing communication is to insert send and receive operations directly preceding each reference causing a nonlocal data access. This generates many small messages that may prove inefficient due to communication overhead. The Fortran D compiler will use *data dependence* information to determine whether communication may be inserted at some outer loop, *vectorizing* messages by combining many small messages. The algorithm to calculate the appropriate loop level for each message is described by Balasundaram *et al.* and Gerndt [2, 10].

A major goal of the Fortran D compiler is to aggressively optimize communications. We intend to apply techniques proposed by Li and Chen to recognize regular computation patterns that can utilize collective communications primitives [24]. It will be especially important to recognize reduction operations. For regular communication patterns, we plan to employ the collective communications routines found in EXPRESS [27]. For unstructured computations with irregular communications, we will incorporate the PARTI primitives of Saltz *et al.* [33].

The Fortran D compiler may utilize data decomposition and dependence information to guide program transformations that improve communication patterns. We are considering the usefulness of several transformations, particularly loop interchanging, strip mining, loop distribution, and loop alignment. Replicating computations and processor-specific dead code elimination will also be applied to eliminate communication.

Communications may be further optimized by considering interactions between all the loop nests in the program. Intra- and interprocedural dataflow analysis of array sections can show that an assignment to a variable is *live* at a point in the program if there are no intervening assignments to that variable. This information may be used to eliminate redundant messages. For instance, assume that messages in previous loop nests have already retrieved nonlocal elements for a given array. If those values are *live*, messages to fetch those values in succeeding loop nests may be eliminated. Data from different arrays being sent to the same processor may also be buffered together in one message to reduce communication overhead.

The *owner computes* rule provides the basic strategy of the Fortran D compiler. We may also relax this rule, allowing processors to compute values for data they do not own. For instance, suppose that multiple *rhs* of an assignment statement are owned by a processor that is not the owner of the *lhs*. Computing the result on the processor owning the *rhs* and then sending the result to the owner of the *lhs* could reduce the amount of data communicated. This optimization is a simple case of the *owner stores* rule proposed by Balasundaram [1].

In particular, it may be desirable for the Fortran D compiler to partition loops amongst processors so that each loop iteration is executed on

a single processor, such as in KALI [22] and PARTI [33]. This technique may improve communication and provide greater control over load balance, especially for irregular computations. It also eliminates the need for individual statement guards and simplifies handling of control flow within the loop body.

3.4 Data Decomposition Analysis

Fortran D provides dynamic data decomposition by permitting ALIGN and DISTRIBUTE statements to be inserted at any point in a program. This complicates the job of the Fortran D compiler, since it must know the decomposition of each array in order to generate the proper guards and communication. We define *reaching decompositions* to be the set of decomposition specifications that may reach an array reference aligned with the decomposition; it may be calculated in a manner similar to *reaching definitions*. The Fortran D compiler will apply both intra- and interprocedural analysis to calculate reaching decompositions for each reference to a distributed array. If multiple decompositions reach a procedure, node splitting or run-time techniques may be required to generate the proper code for the program.

To permit a modular programming style, the effects of data decomposition specifications are limited to the scope of the enclosing procedure. However, procedures do inherit the decompositions of their callers. These semantics require the compiler to insert calls to run-time data decomposition routines to restore the original data decomposition upon every procedure return. Since changing the data decomposition may be expensive, these calls should be eliminated where possible.

We define *live decompositions* to be the set of decomposition specifications that may reach some array reference aligned with the decomposition; it may be calculated in a manner similar to *live variables*. As with reaching decompositions, the Fortran D compiler needs both intra- and interprocedural analysis to calculate live decompositions for each decomposition specification. Any data decompositions determined not to be *live* may be safely eliminated. Similar analysis may also hoist dynamic data decompositions out of loops.

3.5 Run-time Support for Irregular Computations

Many advanced algorithms for scientific applications are not amenable to the techniques described in the previous section. Adaptive meshes, for example, often have poor load balance or high communication cost if static regular data distributions are used. These algorithms require dynamic irregular data distributions. Other algorithms, such as fast multipole algorithms, make heavy use of index arrays that the compiler cannot analyze. In these cases, the communications analysis must be performed at run-time.

The Fortran D project supports dynamic irregular distributions. The *inspector/executor* strategy to generate efficient communications has been adapted from KALI [22] and PARTI [25]. The inspector is a transformation of the original Fortran D loop that builds a list of nonlocal elements, known as the IN set, that will be received during the execution of the loop. A global transpose operation is performed using collective communications to calculate the set of data elements that must be sent by a processor, known as the OUT set. The executor uses the computed sets to control the actual communication. Performance results using the PARTI primitives indicate that the inspector can be implemented with acceptable overhead, particularly if the results are saved for future executions of the original loop [33].

3.6 Storage Management

Once guards and communication have been calculated, the Fortran D compiler must select and manage storage for all nonlocal array references received from other processors. There are several different storage schemes, described below:

- *Overlaps*, developed by Gerndt, are expansions of local array sections to accommodate neighboring nonlocal elements [10]. They are useful for programs with high locality of reference, but may waste storage when nonlocal accesses are distant.
- *Buffers* are designed to overcome the contiguous nature of overlaps. They are useful when the nonlocal area is bounded in size, but not near the local array section.

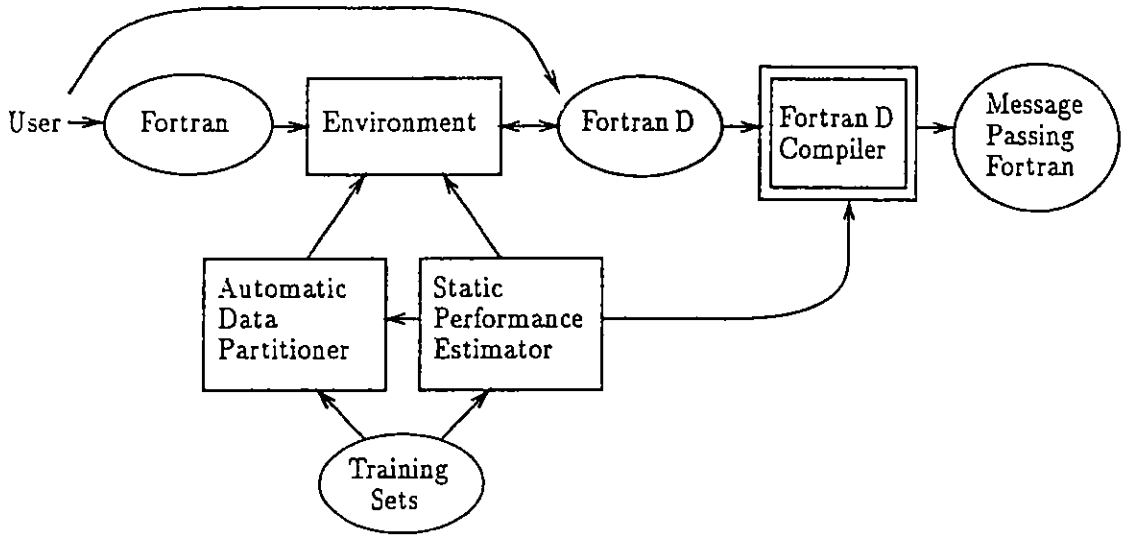


Figure 3: Fortran D Parallel Programming System

- *Hash tables* are used when the set of accessed nonlocal elements is sparse. This is the case in many irregular computations. Hash tables provide a quick lookup mechanism for arbitrary sets of nonlocal values [16].

Once the storage type for all nonlocal data is determined, the compiler needs to analyze the space required by the various storage structures and generate code so that nonlocal data is accessed from its correct location. Storage management and other parts of the Fortran D compiler are described in more detail elsewhere [14, 15].

4 Fortran D Programming Environment

Choosing a decomposition for the fundamental data structures used in the program is a pivotal step in developing data-parallel applications. Once selected, the data decomposition usually completely determines the parallelism and data movement in the resulting program. Unfortunately, there are no existing tools to advise the programmer in making this important decision. To evaluate a decomposition, the programmer must first insert the decomposition in the program text, then compile and run the resulting program to determine its effectiveness. Comparing two data decompositions thus requires implementing and running both versions of the program, a tedious task at best. The process is prohibitively difficult without the assistance of a compiler to automatically generate node programs based on the data decomposition.

Several researchers have proposed techniques to automatically derive data decompositions based on simple machine models [17, 28, 30]. However, these techniques are insufficient because the efficiency of a given data decomposition is highly dependent on both the actual node program generated by the compiler and its performance on the parallel machine. "Optimal" data decompositions may prove inferior because the compiler generates node programs with suboptimal communications or poor load balance. Similarly, marginal data decompositions may perform well because the compiler is able to utilize collective communication primitives to exploit special hardware on the parallel machine.

What we need is a programming environment that helps the user to understand the effect of a given data decomposition and program structure on the efficiency of the *compiler-generated* code running on a given target machine. The Fortran D programming system, shown in Figure 3, provides such an environment. The main components of the environment are a static performance estimator and an automatic data partitioner [2, 3].

Since the Fortran D programming system is built on top of ParaScope, it also provides program analysis, transformation, and editing capabilities that

allow users to restructure their programs according to a data-parallel programming style. Zima and others at Vienna are working on a similar tool to support data decomposition decisions using automatic techniques [7]. Gupta and Banerjee propose automatic data decomposition techniques based on assumptions about a proposed Parafrase-2 distributed-memory compiler [11].

4.1 Static Performance Estimator

It is clearly impractical to use dynamic performance information to choose between data decompositions in our programming environment. Instead, a *static* performance estimator is needed that can accurately predict the performance of a Fortran D program on the target machine. Also required is a scheme that allows the compiler to assess the costs of communication routines and computations. The static performance estimator in the Fortran D programming system caters to both needs.

The performance estimator is not based on a general theoretical model of distributed-memory computers. Instead, it employs the notion of a *training set* of kernel routines that measures the cost of various computation and communication patterns on the target machine. The results of executing the training set on a parallel machine are summarized and used to train the performance estimator for that machine. By utilizing training sets, the performance estimator achieves both accuracy and portability across different machine architectures. The resulting information may also be used by the Fortran D compiler to guide communication optimizations.

The static performance estimator is divided into two parts, a machine module and a compiler module. The *machine module* predicts the performance of a node program containing explicit communications. It uses a *machine-level* training set written in message-passing Fortran. The training set contains individual computation and communication patterns that are timed on the target machine for different numbers of processors and data sizes. To estimate the performance of a node program, the machine module can simply look up results for each computation and communication pattern encountered.

The *compiler module* forms the second part of the static performance estimator. It assists the user in selecting data decompositions by statically predicting the performance of a program for a set of data decompositions. The compiler module employs a *compiler-level* training set written in Fortran D that consists of program kernels such as stencil computations and matrix multiplication. The training set is converted into message-passing Fortran using the Fortran D compiler and executed on the target machine for different data decompositions, numbers of processors, and array sizes. Estimating the performance of a Fortran D program then requires matching computations in the program with kernels from the training set.

The compiler-level training set also provides a natural way to respond to changes in the Fortran D compiler as well as the machine. We simply recompile the training set with the new compiler and execute the resulting programs to reinitialize the compiler module for the performance estimator.

Since it is not possible to incorporate all possible computation patterns in the compiler-level training set, the performance estimator will encounter code fragments that cannot be matched with existing kernels. To estimate the performance of these codes, the compiler module must rely on the machine-level training set. We plan to incorporate elements of the Fortran D compiler in the performance estimator so that it can mimic the compilation process. The compiler module can thus convert any unrecognized Fortran D program fragment into an equivalent node program, and invoke the machine module to estimate its performance.

Note that even though it is desirable, to assist automatic data decomposition the static performance estimator does not need to predict the *absolute* performance of a given data decomposition. Instead, the it only needs to accurately predict the performance *relative* to other data decompositions. A prototype of the machine module has been implemented for a common class of *loosely synchronous* scientific problems[9]. It predicts the performance of a node program using EXPRESS communication routines for different numbers of processors and data sizes [27]. The prototype performance estimator has proved quite precise, especially in predicting the relative performances of different data decompositions [3].

A screen snapshot during a typical performance

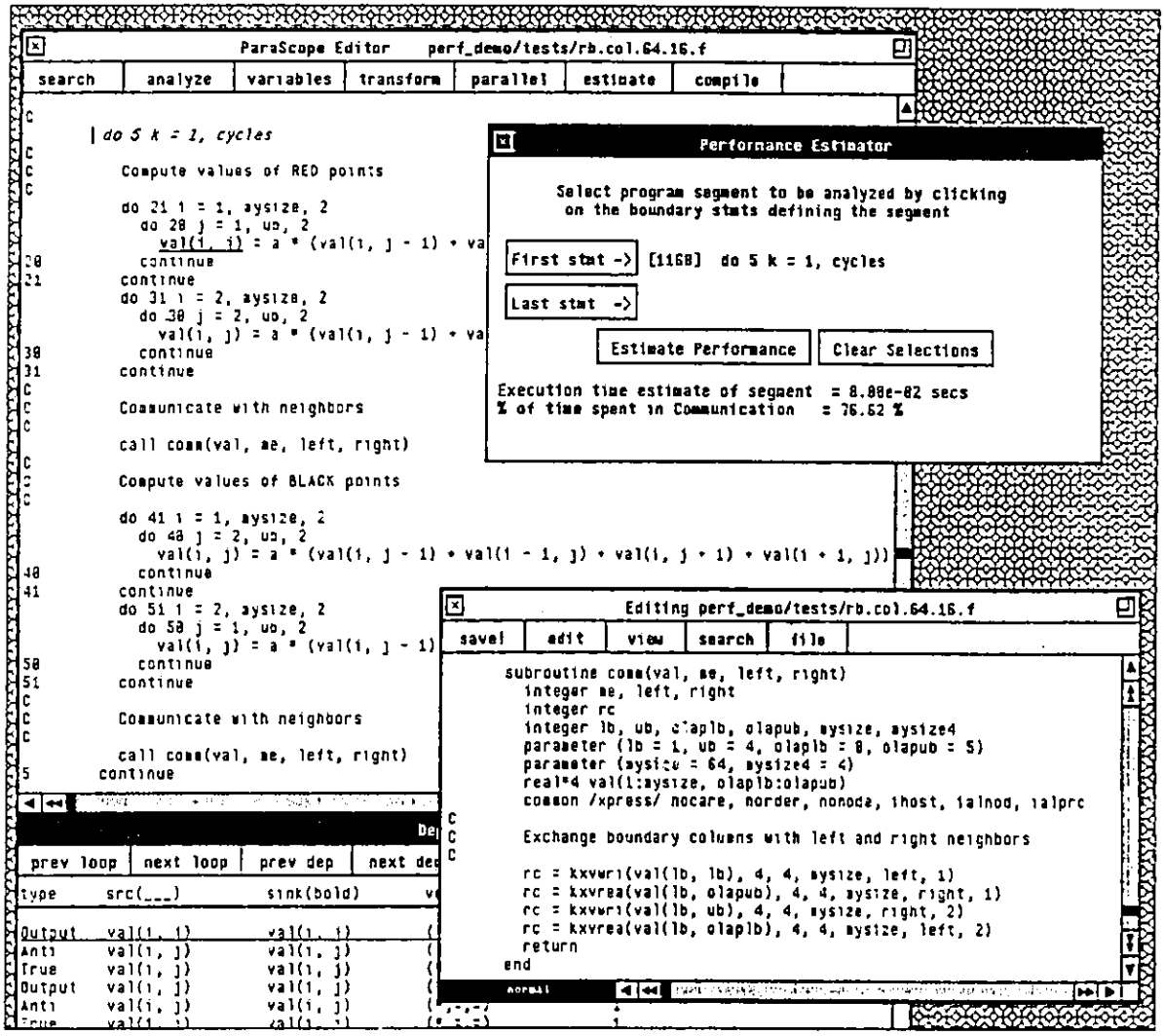


Figure 4: Static Performance Estimator

estimation session is shown in Figure 4. The user can select a program segment such as a do loop and invoke the performance estimator by clicking on the Estimate Performance button. The prototype responds with an execution time estimate of the selected segment on the target machine, as well as an estimate of the communication time represented as a percentage of the total execution time. This allows the effectiveness of a data partitioning strategy to be evaluated on any part of the node program.

4.2 Automatic Data Partitioner

The goal of the automatic data partitioner is to assist the user in choosing a good data decomposition. It utilizes training sets and the static performance estimator to select data partitions that are efficient for both the compiler and parallel machine.

The automatic data partitioner may be applied to an entire program or on specific program fragments. When invoked on an entire program, it automatically selects data decompositions without further user interaction. We believe that for regular loosely synchronous problems written in a data-parallel programming style, the automatic data partitioner can determine an efficient partitioning scheme without user interaction.

Alternatively, the automatic data partitioner may be used as a starting point for choosing a good data decomposition. When invoked interac-

tively for specific program segments, it responds with a list of the best decomposition schemes, together with their static performance estimates. If the user is not satisfied with the predicted overall performance, he or she can use the performance estimator to locate communication and computation intensive program segments. The Fortran D environment can then advise the user about the effects of program changes on the choice of a good data decomposition.

The analysis performed by the automatic data partitioner divides the program into separate *computation phases*. The *intra-phase* decomposition problem consists of determining a set of good data decompositions and their performance for each individual phase. The data partitioner first tries to match the phase or parts of the phase with computation patterns in the compiler training set. If a match is found, it returns the set of decompositions with the best measured performance as recorded in the compiler training set. If no match is found, the data partitioner must perform alignment and distribution analysis on the phase. The resulting solution may be less accurate since the effects of the Fortran D compiler and target machine can only be estimated.

Alignment analysis is used to prune the search space of possible arrays alignments by selecting only those alignments that minimize data movement. Alignment analysis is largely machine-independent; it is performed by analyzing the array access patterns of computations in the phase. We intend to build on the inter-dimensional and intra-dimensional alignment techniques of Li and Chen [23] and Knobe *et al.* [21].

Distribution analysis follows alignment analysis. It applies heuristics to prune unprofitable choices in the search space of possible distributions. The efficiency of a data distribution is determined by machine-dependent aspects such as topology, number of processors, and communication costs. The automatic data partitioner uses the final set of alignments and distributions to generate a set of reasonable data decomposition schemes. In the worst case, the set of decompositions is the cross product of the alignment and distribution sets. Finally, the static performance estimator is invoked to select the set of data decompositions with the best predicted performance.

After computing data decompositions for each phase, the automatic data partitioner must solve the *inter-phase* decomposition problem of merging individual data decompositions. It also determines the profitability of realigning or redistributing arrays between computational phases. Interprocedural analysis will be used to merge the decomposition schemes of computation phases across procedure boundaries. The resulting decompositions for the entire program and their performance are then presented to the user.

5 Validation Strategy

We plan to establish whether our compilation and automatic data partitioning schemes for Fortran D can achieve acceptable performance on a variety of parallel architectures. We will use a benchmark suite being developed by Geoffrey Fox at Syracuse that consists of a collection of Fortran programs. Each program in the suite will have five versions:

(v1) the original Fortran 77 program,

(v2) the best hand-coded message-passing version of the Fortran program,

(v3) a "nearby" Fortran 77 program,

(v4) a Fortran D version of the nearby program, and

(v5) a Fortran 90 version of the program.

The "nearby" version of the program will utilize the same basic algorithm as the message-passing program, except that all explicit message-passing and blocking of loops in the program are removed. The Fortran D version of the program consists of the nearby version plus appropriate data decomposition specifications.

To validate the Fortran D compiler, we will compare the running time of the best hand-coded message-passing version of the program (v2) with the output of the Fortran D compiler for the Fortran D version of the nearby program (v4). To validate the automatic data partitioner, we will use it to generate a Fortran D program from the nearby Fortran program (v3). The result will be compiled by the Fortran D compiler and its running time compared with that of the compiled version of the hand-generated Fortran D program (v4).

The purpose of the validation program suite is to provide a fair test of the prototype compiler and

data partitioner. We do not expect these tools to perform high-level algorithm changes. However, we will test their ability to analyze and optimize whole programs based on both machine-independent issues such as the structure of the computation, as well as machine-dependent issues such as the number and interconnection of processors in the parallel machine. Our validation strategy will test three key parts of the Fortran D programming system: the limits of our machine-independent Fortran D programming model, the efficiency and ability of our compiler technology, and the effectiveness of our automatic data partitioning and performance estimation techniques.

6 Conclusions

Scientific programmers need a simple, machine-independent programming model that can be efficiently mapped to large-scale parallel machines. We believe that Fortran D, a version of Fortran enhanced with data decompositions, provides such a portable data-parallel programming model. Its success will depend on the compiler and environment support provided by the Fortran D programming system.

The Fortran D compiler includes sophisticated intraprocedural and interprocedural analyses, dynamic data decomposition, program transformation, communication optimization, and support for both regular and irregular problems. Though significant work remains to implement the optimizations presented in this paper, based on preliminary experiments we expect the Fortran D compiler to generate efficient code for a large class of data-parallel programs with only minimal user effort.

The Fortran D environment is distinguished by its ability to accurately estimate the performance of programs using collective communication on real parallel machines, as well automatically choose data partitions that account for the characteristics of both the compiler-generated code and underlying machine. It will assist the user in developing efficient Fortran D programs. Overall, we believe that the Fortran D programming system is a powerful and useful tool that will significantly ease the task of writing portable data-parallel programs.

7 Acknowledgements

The authors wish to thank Vasanth Bala, Geoffrey Fox, and Marina Kalem for inspiring many of the ideas in this work. We are also grateful to the ParaScope research group for providing the underlying software infrastructure for the Fortran D programming system.

References

[1] V. Balasundaram. Translating control parallelism to data parallelism. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, Houston, TX, March 1991.

[2] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[3] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[4] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4):84–99, Winter 1988.

[5] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[6] D. Callahan, K. Kennedy, and U. Kremer. A dynamic study of vectorization in PFC. Technical Report TR89-97, Dept. of Computer Science, Rice University, July 1989.

[7] B. Chapman, H. Herbeck, and H. Zima. Automatic support for data distribution. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[8] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[9] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[10] M. Gerndt. Updating distributed variables in local computations. *Concurrency—Practice & Experi-*

ence, 2(3):171–193, September 1990.

[11] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[12] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[13] R. Hill. MIMDizer: A new tool for parallelization. *Supercomputing Review*, 3(4):26–28, April 1990.

[14] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[15] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. Technical Report TR90-149, Dept. of Computer Science, Rice University, January 1991. To appear in J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*, Elsevier, 1991.

[16] S. Hiranandani, J. Saltz, P. Mehrotra, and H. Berryman. Performance of hashed cache data migration schemes on multicomputers. *Journal of Parallel and Distributed Computing*, 12(4), August 1991.

[17] D. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[18] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel compute s. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[19] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[20] K. Kennedy, K. S. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[21] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[22] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4), October 1991.

[23] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[24] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[25] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[26] C. Pancake and D. Bergmark. Do parallel languages respond to the needs of scientific programmers? *IEEE Computer*, 23(12):13–23, December 1990.

[27] Parasoft Corporation. *Express User's Manual*, 1989.

[28] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[29] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[30] L. Snyder and D. Socha. An algorithm producing balanced partitionings of data arrays. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[31] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, version 5.2-0.6 edition, September 1989.

[32] M. J. Wolfe. Semi-automatic domain decomposition. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.

[33] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[34] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

OSCAR FORTRAN COMPILER

H. Kasahara, H. Honda, K. Aida, M. Okamoto and S. Narita
Dept. of Information and Computer Sciences, Waseda University
3-4-1 Ohkubo Shinjuku-ku, Tokyo, 169, Japan. Tel. 03-3209-6323, Fax. 03-3232-3594
E-mail: kasahara@cfi.waseda.ac.jp

Abstract: OSCAR FORTRAN compiler has been developed for a shared memory multiprocessor system named OSCAR (Optimally Scheduled Advanced Multiprocessor). The compiler hierarchically exploits coarse grain parallelism among loops, subroutines and basic blocks, medium grain parallelism among loop-iterations and near fine grain parallelism among statements. The coarse grain parallelism is automatically detected in the form of earliest executable conditions of the coarse grain tasks, or the macro-tasks. The earliest executable conditions are obtained by a unified control dependence and data dependence analysis. The macrotasks are dynamically assigned to processor clusters by a scheduling routine generated by the compiler. A macrotask composed of a Do-all or Do-across loop, which is assigned onto a processor cluster, is hierarchically processed in parallel in the medium grain by processors inside the processor cluster. A macrotask composed of a sequential loop or a basic block on a processor cluster is also processed in parallel in the near fine grain by using static scheduling. A prototype compiler has been implemented on OSCAR having sixteen RISC processors and its usefulness has been confirmed on the system.

Key Words: Macro-dataflow, Dynamic scheduling supported by compiler, Earliest executable conditions, Near fine grain parallel processing, Static scheduling, Multi-grain parallel processing

1. INTRODUCTION

In parallel processing of FORTRAN programs on shared memory multiprocessor systems, the Do-all and the Do-across [4][8][10] has widely been used. Thanks to strong data dependency analysis [2][3][4] and program restructuring techniques [4][9], many types of Do-loops can be concurrentized.

There still exist, however, sequential loops that can not be concurrentized efficiently because of loop carrying dependencies and conditional branches. Also, fine grain parallelism inside a basic block or coarse grain parallelism among loops, subroutines and basic blocks has not effectively been exploited on multiprocessor systems.

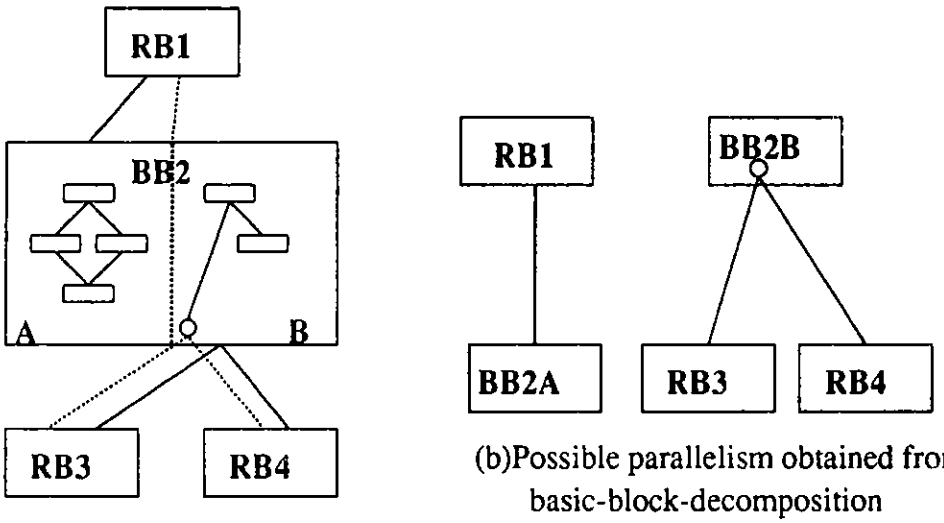
Therefore, to improve the effective performance of multiprocessor systems further, it is important to exploit the coarse grain parallelism and the fine grain parallelism as well as the medium grain parallelism among iterations. The coarse grain parallel processing on a hierarchical multiprocessor system is also called the macro-dataflow computation [5]-[7] that has not been realized yet on an actual multiprocessor system.

In the fine grain parallel processing on multiprocessor systems [30][33], an instruction level grain, which has been used by VLIW processors [12]-[15] and superscalar processors [17], seems too fine compared with the data transfer overhead among processors. Therefore, the near fine grain parallelism among statements has been exploited with the use of a static scheduling algorithm considering data transfer overhead [30]. Also, it needs architectural supports for efficient synchronization [29] and data transfer [30]. However, the parallel processing using the static scheduling [12]-[14] generally has a problem to cope with run-time uncertainties.

Considering the above facts, OSCAR compiler has adopted a multi-grain parallel processing scheme [16] that effectively combines the macro-dataflow computation, the loop concurrentization and the near fine grain processing. In the scheme, macrotasks are dynamically scheduled onto processor clusters to cope with the run-time uncertainties caused by conditional branches. A macrotask assigned to a processor cluster is hierarchically processed by using the loop concurrentization, the near fine grain parallel processing, or the macro-dataflow computation.

2. MULTI-GRAIN COMPILATION SCHEME

OSCAR compiler adopts the multi-grain compilation scheme. This section briefly describes compilation



(a)An example of a basic block having disjoint task graphs

(b)Possible parallelism obtained from basic-block-decomposition

Fig.1 BPAs generated by basic block decomposition.

schemes for the macro-dataflow and the near fine grain parallel processing because the well-known compilation schemes [4][8]-[10][20][34] can be used for the loop concurrentization.

2.1 Compilation Scheme for Macro-dataflow [33]

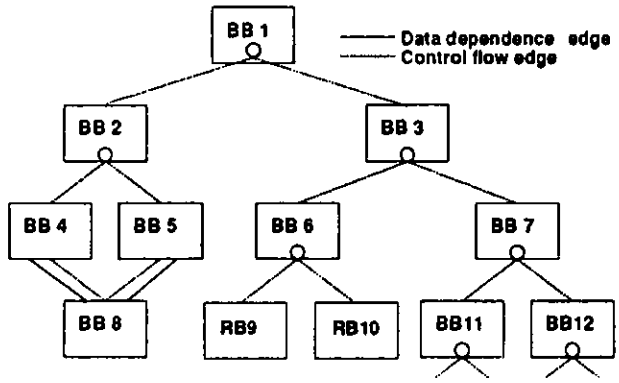
The macro-dataflow compilation scheme mainly consists of the four steps, namely, generation of macrotasks, analysis of control-flow and data-dependence among the macrotasks, extraction of parallelism among macrotasks, and generation of dynamic scheduling routine.

2.1.1 Generation of macrotasks

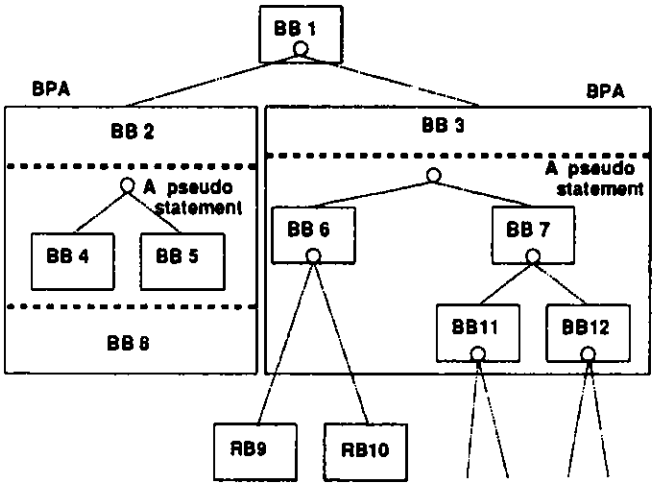
A FORTRAN program is decomposed into macrotasks that are assigned to processor cluster at runtime. Then, macrotasks should have relatively large processing time compared with dynamic scheduling overhead and data transfer overhead among macrotasks. The compiler generates three types of macrotasks, namely, a Block of Pseudo Assignment statements (BPA), a Repetition Block (RB) and a Subroutine Block (SB).

A BPA is usually defined as a basic clock (BB) [1]. However, it is sometimes defined as a part of a basic block or a block composed of multiple basic blocks.

Decomposition of a basic block into independent blocks, or BPAs, applied to extract more parallelism among macrotasks. For example, in Fig. 1(a), BB2 includes two disjoint parts, such as, a post-processing part for a preceding Do-loop, or RB1, and a pre-processing part for succeeding Do-loops, namely RB3 and RB4. Since the two parts are disjoint, BB2 can be decomposed into BB2A and BB2B as shown in Fig.1(b). By this decomposition, a group composed of RB1 and BB2A and another group composed of BB2B, RB3 and RB4 can be processed in parallel.



(a) A flow graph with several small basic blocks (BBs)



(b) BPAs generated by fusing small BBs

Fig.2 BPAs generated by basic block fusion.

Fusion of small basic blocks into a BPA is used to reduce dynamic scheduling overhead. For example, if BB4 and BB5 in a flow graph of Fig. 2(a) are small basic blocks having few statements, BB4 and BB5 are fused into a conditional branch statement shown as a small circle inside BB2. The conditional statement containing statement inside BB4 and BB5 is treated as a pseudo statement as shown in Fig.2(b). Furthermore, BB8 is fused into the block containing BB2, BB4 and BB5 if BB8 is data dependent on BB4 and BB5 as Fig. 2(a). The block generated by the basic block fusion is called BPA.

A RB is a Do loop or a loop generated by a backward branch, namely, an outermost natural loop [1]. RB can be easily defined in reducible flow graphs [6] and in irreducible flow graphs by copying code [6].

The RB can be hierarchically decomposed into sub-macrotasks when the loop concurrentization and the near fine grain parallel processing can not be applied efficiently to the RB. The sub-macrotasks are dynamically scheduled onto processors inside a processor cluster at run-time. In the decomposition of RB into sub-macrotasks, it is useful for exploiting more parallelism to structure overlapped loops by copying code [27].

As to subroutines, the in-line expansion is applied as much as possible taking code length into account. Subroutines for which the in-line expansion technique can not efficiently be applied are defined as SBs. To fully exploit parallelism among SBs and the other macrotasks in a flow graph, strong inter-procedural analysis techniques are required [28] though the inter-procedural analysis itself is beyond the scope of this paper. SBs can also be hierarchically decomposed into sub-macrotasks as well as RBs.

2.1.2 Representation of control-flow and data dependence among macrotasks by macroflow graph (MFG)

A macroflow graph explicitly represents both control flow and data dependencies among macrotasks. Fig. 3 shows an example of a macroflow graph.

In this macroflow graph, nodes represent macrotasks, such as BPAs, RBs and SBs. Dotted edges represent control flow. Solid edges represent data dependencies among macrotasks. Small circles inside nodes represent conditional branch statements inside macrotasks. In this graph, directions of the edges are assumed to be downward though arrows are omitted. MFG is a directed acyclic graph because all back-edges are contained in RBs.

2.1.3 Extraction of parallelism among macrotasks

The MFG explicitly represents the control flow and data dependencies among macrotasks though it does not show any parallelism among macrotasks. Generally, the control dependence graph, or the program dependence graph [26], represents maximum parallelism if there are not data dependencies among macrotasks [25]. In practice, there exist, however, data dependencies among macrotasks. Therefore, to extract parallelism among macrotasks from a macroflow graph, the control dependencies and the data dependencies should be analyzed in a unified manner.

In this paper, an earliest executable condition of each macrotask [31][36] is used to show the maximum parallelism among macrotasks considering control dependencies and data dependencies. The earliest executable condition of a macrotask i, MT_i, is a condition on which MT_i may begin its execution earliest.

For example, an earliest executable condition of MT_6, which is control-dependent on MT_1 and on MT_2 and is data-dependent on MT_3, is:

MT_3 completes execution **OR** MT_2 branches to MT_4.

Here, "MT_3 completes execution" means to satisfy the data dependence of MT_6 on MT_3 because the following conditions for macro-dataflow execution are assumed in this paper:

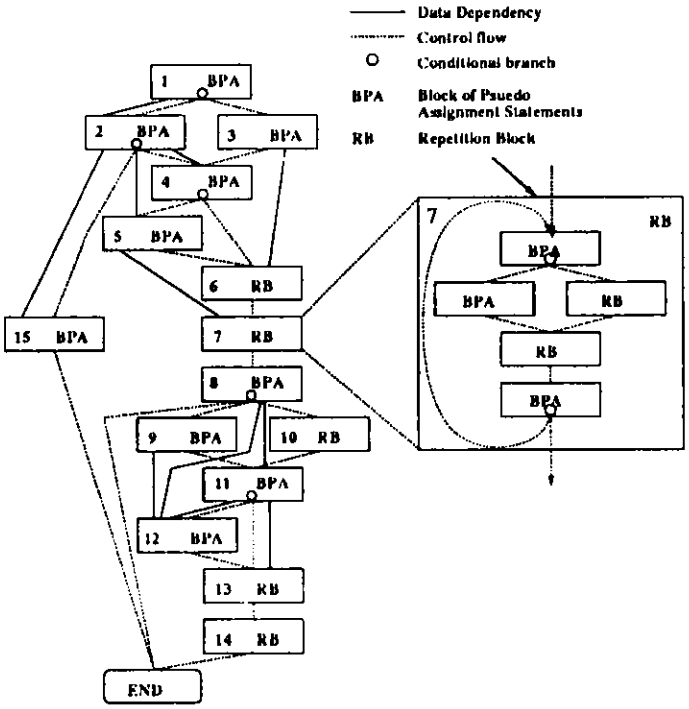


Fig.3 A macro-flow graph.

TABLE 1 Ealiest Executable Conditions of Macrotasks

Macrotask No.	Earliest Executable Condition
1	
2	1_2
3	$(1)_3$
4	2_4 **OR** $(1)_3$
5	$(4)_5$ **AND** (2_4 **OR** $(1)_3$)
6	3 **OR** $(2)_4$
7	5 **OR** $(4)_6$
8	$(2)_4$ **OR** $(1)_3$
9	$(8)_9$
10	$(8)_{10}$
11	8_9 **OR** 8_{10}
12	11_{12} **AND** (9 **OR** $(8)_{10}$)
13	11_{13} **OR** 11_{12}

1) If macrotask i (MT_i) is data-dependent on macrotask j (MT_j), MT_i can not begin execution before MT_j finishes execution.

2) A conditional branch statement inside a macrotask may be executed as soon as data dependencies of the branch statement are satisfied. That is because statements inside a macrotask are processed in parallel by processors inside a processor cluster. In other words, MT_i, which is control-dependent on MT_j, can begin execution as soon as the branch direction is determined even if MT_j has not completed execution.

The above earliest executable condition of MT_6 represents the simplest form of the condition [31][33]. An original form of the condition of MT_i [31][33] can be represented in the following;

(MT_j, on which MT_i is control dependent, branches to MT_i)

AND

(Every macrotask on which MT_i is data dependent, MT_k: $0 \le k < |NI|$, completes execution **OR** it is determined that *MT_k is not be executed*).

For example, the original form of the earliest executable condition of MT_6 is:

(MT_1 branches to MT_3 **OR** MT_2 branches to MT_4)

AND

(MT_3 completes execution **OR** MT_1, on which MT_3 is control-dependent, branches to MT_2).

The first partial condition before **AND** represents an earliest executable condition determined by the control dependencies. The second partial condition after **AND** represents an earliest executable condition to satisfy the data dependence. The second partial condition means that MT_6 may begin execution after MT_3 completes execution or after it is determined that MT_3 is not executed. In the condition, the execution of MT_3 means that MT_1 has branched to MT_3 and the execution of MT_2 means that MT_1 has branched to MT_2. Therefore, this condition is redundant and its simplest form is:

MT_3 completes execution **OR** MT_2 branches to MT_4.

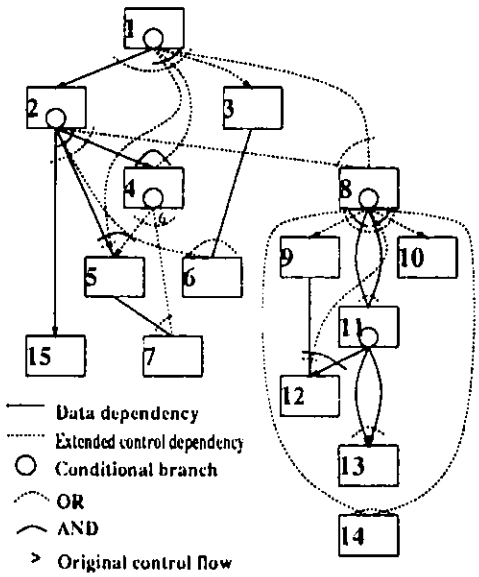


Fig.4 A macrotask graph.

The simple earliest executable conditions of macrotasks on Fig.3, which are given by OSCAR compiler automatically [31], are shown in Table 1. In the table, the earliest executable condition of MT_{12} represented by

$$11_{12}\ \textbf{AND}\ \{\ 9\ \textbf{OR}\ (8)_{10}\ \}$$

means that the condition is:

MT_{11} branches to MT_{12} and completes execution

AND

{MT_9 completes execution **OR** MT_8 branches to MT_{10}.}

The simplest condition is important to reduce dynamic scheduling overhead.

Girkar and Polychronopoulos [35] proposed another algorithm to obtain the earliest executable conditions based on the original research [31]. They solved a simplified problem to obtain the earliest executable conditions by assuming a conditional branch inside a macrotask is executed in the end of the macrotask.

The earliest executable conditions of MTs are represented by a directed acyclic graph named a macrotask graph [31][33][36], or MTG, as shown in Fig. 4. In MTG, nodes represent macrotasks. Dotted edges represent extended control-dependencies. Solid edges represent data-dependencies.

The extended control dependence edges are classified into two types of edges, namely ordinary control dependence edges and co-control dependence edges. The co-control dependence edges represent conditions on which data dependence predecessor of MT_i, namely MT_k mentioned before on which MT_i is data dependent, is not be executed [31].

Also, a data dependence edge, or a solid edge, originating from a small circle has two meanings, namely, an extended control dependence edge and a data dependence edge. Arcs connecting edges at their tails or heads have two different meanings. A solid arc represents that edges connected by the arc are in AND relationship. A dotted arc represents that edges connected by the arc are in OR relationship. Small circles inside nodes represent conditional branch statements.

In the MTG, the directions of the edges are also assumed to be downward though most arrows are omitted. Edges with arrows show that the edges are the original conditional flow edges that originate from the small circles in the MFG.

2.1.4 Generation of dynamic scheduling routine

In the macro-dataflow computation, the macrotasks are dynamically scheduled to processor clusters (PCs) at run-time to cope with runtime uncertainties, such as, conditional branches among macrotasks and a variation of macrotask execution time. The use of dynamic scheduling [20][22] for coarse grain tasks keeps the relative scheduling overhead small. Furthermore, the dynamic scheduling in this scheme is performed not by OS calls like in popular multiprocessor systems but by a special scheduling routine generated by the compiler.

In other words, the compiler generates an efficient dynamic scheduling code exclusively for each FORTRAN program based on the earliest executable conditions, or the macrotask graph. The scheduling routine is executed by a processor element.

Dynamic-CP algorithm, which is a dynamic scheduling algorithm using critical path length [18], is employed taking into consideration the scheduling overhead and quality of the generated schedule.

2.2 Medium Grain Parallel Processing

Macrotasks are assigned to processor clusters (PCs) dynamically as mentioned in the previous section. If a macrotask assigned to a PC is a Do-all loop, the macrotask is processed in the medium grain, or iteration level grain, by processors inside the PC. For the Do-all, several dynamic scheduling schemes, such as the self scheduling, the chunk scheduling and the guided self scheduling, have been proposed [10][20]. On OSCAR, however, a simple static scheduling scheme is used because OSCAR does not have a hardware support for the dynamic iteration scheduling.

If a macrotask assigned to a PC is a loop having data dependencies among iterations, the compiler first tries to apply the Do-across with restructuring to minimize the synchronization overhead [8][9]. Next, the compiler compares an estimated processing time by the Do-across and by the near fine grain parallel processing of the loop body mentioned in section 2.3. If the processing time by the Do-across is shorter than the one by the near fine grain processing, the compiler generates a machine code for the Do-across.

2.3 Near Fine Grain Parallel Processing [31][33]

A BPA is decomposed into the near fine grain tasks [31], each of which consists of a statement, and processed in parallel by processors inside a PC.

2.3.1 Generation of tasks and task graph

To efficiently process a BPA in parallel, computation in the BPA must be decomposed into tasks in such a way that parallelism is fully exploited and overhead related with data transfer and synchronization is kept small.

In the proposed scheme, the statement level granularity is chosen as the finest granularity for OSCAR taking into account OSCAR's processing capability and data transfer capability.

Fig. 5 shows an example of statement level tasks, or near fine grain tasks, generated for a basic block that solves a sparse matrix [37]. A large basic block having computational pattern like Fig.5 is generated by the symbolic generation technique [32] that has been used in the electronic circuit simulator like SPICE.

Among the generated tasks, there are data dependencies [2]-[4]. The data dependencies, or precedence constraints, can be represented by arcs in a task graph [18][23] as shown in Fig.6, in which each task corresponds to a node. In Fig. 6, figures inside a node circle represent task number, i, and those beside it for a task processing time on a PE, t_i. An edge directed from node N_i toward N_j represents partially ordered constraint that task T_i precedes task T_j. When we also consider a data transfer time between tasks, each edge generally has a variable weight. Its weight, t_{ij}, will be a data transfer time between task T_i and T_j if T_i and T_j are assigned to different PEs. It will be zero or a time to access registers or local data memories if the tasks are assigned to the same PE.

2.3.2 Static multiprocessor scheduling algorithm

To process a set of tasks on a multiprocessor system efficiently, an assignment of tasks onto PEs and an execution order among the tasks assigned to the same PE must be determined optimally. The problem that determines the optimal assignment and the optimal execution order can be treated as a traditional minimum execution time multiprocessor scheduling problem [18][23]. To state formally, the scheduling problem is to determine such a nonpreemptive schedule in which execution time or schedule length be minimum, given a set of n computational tasks, precedence relations among them, and m processors with the same processing capability. This

<< LU Decomposition >>

1) $u_{12} = a_{12} / l_{11}$
2) $u_{24} = a_{24} / l_{22}$
3) $u_{34} = a_{34} / l_{33}$
4) $l_{54} = -l_{52} * u_{24}$
5) $u_{45} = a_{45} / l_{44}$
6) $l_{55} = a_{55} - l_{54} * u_{45}$

<< Forward Substitution >>

7) $y_1 = b_1 / l_{11}$
8) $y_2 = b_2 / l_{22}$
9) $b_5 = b_5 - l_{52} * y_2$
10) $y_3 = b_3 / l_{33}$
11) $y_4 = b_4 / l_{44}$
12) $b_5 = b_5 - l_{54} * y_4$
13) $y_5 = b_5 / l_{55}$

<< Backward Substitution >>

14) $x_4 = y_4 - u_{45} * y_5$
15) $x_3 = y_3 - u_{34} * x_4$
16) $x_2 = y_2 - u_{24} * x_4$
17) $x_1 = y_1 - u_{12} * x_2$

Fig.5 An example of near fine grain tasks.

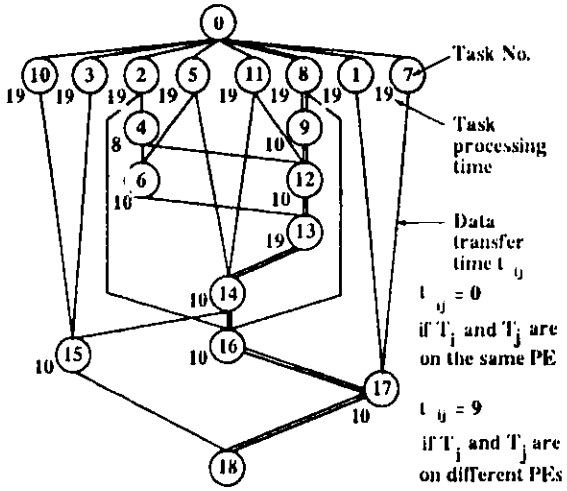


Fig.6 A task graph for near fine grain tasks.

scheduling problem, however, has been known as a "strong" NP-hard problem [19].

Considering this fact, a variety of heuristic algorithms and a practical optimization algorithm have been proposed [18][20][23]. In OSCAR compiler, a heuristic scheduling algorithm CP/DT/MISF (Critical Path/ Data Transfer/ Most Immediate Successors First) considering data transfer [30] has been adopted taking into account a compilation time and quality of generated schedules.

2.3.3. Machine code generation

For efficient execution on an actual multiprocessor system, the optimal machine codes must be generated by using the scheduled results. A scheduled result gives us the following information:

1) which tasks are executed on each PE,
2) in which order the tasks assigned to the same PE are executed,
3) when and where data transfers and synchronization among PEs are required,

and so on. Therefore, we can generate the machine codes for each PE by putting together instructions for tasks assigned to the PE and inserting instructions for data transfer and synchronization into the required places. The "version number" method [30] is used for synchronization among tasks.

At the end of a BPA, instructions for the barrier synchronization, which is supported by OSCAR's hardware, are inserted into a program code on each PE.

The compiler can also optimize the codes by making full use of all information obtained from the static scheduling. For example, when a task should pass shared data to other tasks assigned to the same PE, the data can be passed through registers on the PE. The optimal use of registers reduces the processing time markedly. In addition, the compiler can minimize the synchronization overhead by carefully considering the information about the tasks to be synchronized, the task assignment and the execution order [31].

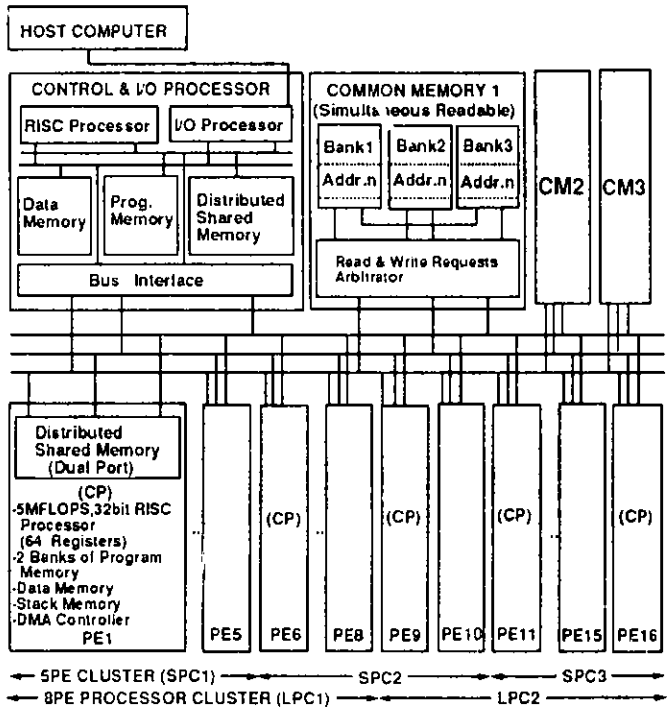


Fig.7 OSCAR's architecture.

3. OSCAR'S ARCHITECTURE

This section describes the architecture of OSCAR (Optimally Scheduled Advanced Multiprocessor). Fig.7 shows the architecture of OSCAR. As shown in Fig.7, OSCAR is a shared memory multiprocessor system in which up to sixteen processor elements (PEs) are uniformly connected to three centralized common memories (CMs) and to distributed shared memories on the PEs through three buses.

Each PE is a custom-made RISC processor with throughput of 5 MFLOPS. It consists of a main processing unit with sixty-four registers, an integer processing unit and a floating point processing unit, a data memory, two banks of program memories, a distributed shared memory, a stack memory (SM) and a DMA controller. The PE executes every instruction including a floating point addition and a multiplication in one clock. The distributed shared memory on each PE can be accessed simultaneously by the PE itself and another PE.

Also, OSCAR provides the following three types of data transfer modes by using the DPMs and the CMs:

1) One PE to one PE direct data transfers using DPMs,
2) One PE to all PEs data broadcasting using the DPM,
3) One PE to several PEs indirect data transfers through CMs.

Each CM is a simultaneously readable memory on which the same address or different addresses can be read by three PEs in the same clock.

3.1 Architectural Supports for the Macro-dataflow

OSCAR can simulate a multiple-PC system having two or three PCs by assigning one bus to each PC. A number of PCs and a number of PEs inside PC can be changed even at run-time according to parallelism of the target program, or the macrotask graph because partitioning of PEs into PCs is made by software. Furthermore, each bus has a control line for the barrier synchronization. Therefore, each PC can take barrier synchronization in a few clocks.

3.2 Architectural Supports for the Fine Grain Parallel Processing

For the near fine grain parallel processing on OSCAR, the one PE to one PE direct data transfer and the data broadcasting using the DPM are used for minimizing data transfer overhead. The direct data transfer using the DPM needs only one "data-write" onto a DPM for passing one data from one PE to another PE. On the other hand, the conventional indirect data transfer using a CM requires one "data-write" to a CM and one "data-read"

from the CM. Also, the data broadcasting reduces the data transfer time remarkably compared with the indirect data transfer through CM. Therefore, the optimal use of the three data transfer modes using static scheduling allows us to reduce data transfer overhead. Also, synchronization using DPMs reduces synchronization overhead because assigning synchronization-flags onto the DPMs prevents degradation of bus band width that is caused by the busy wait to check synchronization-flags on CMs.

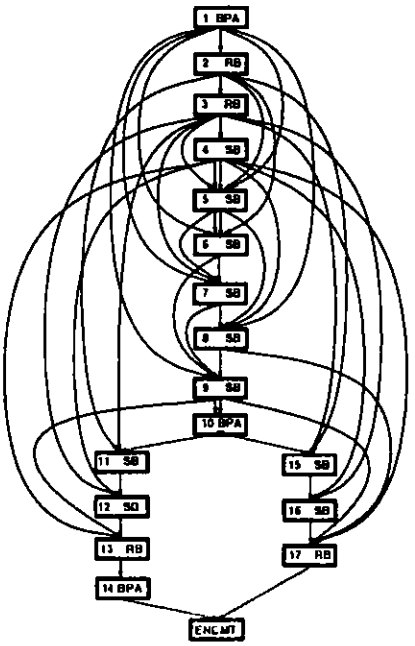
4. AN EXECUTION EXAMPLE OF OSCAR COMPILER

```
C   PROGRAM SAMPLE FOR OSCAR
    REAL A(300,300),B(300,300),C(300,300)
    REAL X(300,300),Y(300,300),Z(300,300)
    REAL V1(300),V2(300),V3(300),V4(300)
    REAL D1
    COMMON /COM1/V1
C   ------------------- MT 1
    D1= 0.0
    V1(1)= 1.0
C   ------------------- MT 2
    DO 110 I1= 1,300
     DO 100 J1= 1,300
      A(I1,J1)= I1+ J1-300
  100  CONTINUE
  110 CONTINUE
C   ------------------- MT 3
    DO 130 I2= 1,300
     DO 120 J2= 1,300
      B(I2,J2)= I2-J2+ 1
  120  CONTINUE
  130 CONTINUE
C   ------------------- MT 4
    CALL MTGEN(C)
C   ------------------- MT 5
    CALL MKVEC(A,B,C)
C   ------------------- MT 6
    CALL MTVEC(A,V2)
C   ------------------- MT 7
    CALL MTVEC(B,V3)
C   ------------------- MT 8
    CALL MTVEC(C,V4)
C   ------------------- MT 9
    CALL VECSUB(V2,V3,D1)
C   ------------------- MT10
    IF(D1.GT.0.0) THEN
C   ------------------- MT11
     CALL MATADD(A,B,X)
C   ------------------- MT12
     CALL MATSUB(B,C,Y)
C   ------------------- MT13
     DO 170 I3= 1,300
      DO 160 J3= 1,300
       Z(I3,J3)= C(I3,J3)/D1
  160  CONTINUE
  170 CONTINUE
C   ------------------- MT14
    ELSE
C   ------------------- MT15
     CALL MATSUB(B,A,Y)
C   ------------------- MT16
     CALL MATADD(C,B,X)
C   ------------------- MT17
     DO 190 I4= 1,300
      DO 180 J4= 1,300
       Z(I4,J4)= C(I4,J4)*D1*V4(I4)
  180  CONTINUE
  190 CONTINUE
C
    ENDIF
C
    END
C
    SUBROUTINE MTGEN(X)
    REAL X(300,300)
    DO 110 I= 1,300
     DO 100 J= 1,300
      X(I,J)= 0.0
  100  CONTINUE
       X(I,I)= 1.0
  110 CONTINUE
      RETURN
      END
C
      SUBROUTINE MATADD(X,Y,Z)
      REAL X(300,300),Y(300,300),Z(300,300)
      DO 120 I= 1,300
       DO 110 J= 1,300
        Z(I,J)= X(I,J)+ Y(I,J)
  110  CONTINUE
  120 CONTINUE
      RETURN
      END
C
      SUBROUTINE MATSUB(X,Y,Z)
      REAL X(300,300),Y(300,300),Z(300,300)
      DO 120 I= 1,300
       DO 110 J= 1,300
        Z(I,J)= X(I,J)-Y(I,J)
  110  CONTINUE
  120 CONTINUE
      RETURN
      END
C
      SUBROUTINE MKVEC(A,B,C)
      REAL A(300,300),B(300,300),C(300,300)
      REAL DV,R1,TA,TB,TC,SA,SB,SC,D1,D2,D3
      COMMON /COM1/V1(300)
      DO 100 I= 2,300
       DV= V1(I-1)
       R1= 0.314*DV
       TA= A(I,I)*DV
       TB= B(I,I)*DV
       TC= C(I,I)*DV
       SA= TA*TA
       SB= TB*TB
       SC= TC*TC
       D1= TA-TB
       D2= TB-TC
       D3= TC-TA
       V1(I)= (SA+ SB+ SC)*R1+ (D1*D2*D3)
  100 CONTINUE
      RETURN
      END
C
      SUBROUTINE MTVEC(A,V)
      REAL A(300,300),V(300)
      COMMON /COM1/V1(300)
      DO 110 I= 1,300
       V(I)= 0.0
       DO 100 J= 1,300
        V(I)= V(I)+ A(I,J)*V1(J)
  100  CONTINUE
  110 CONTINUE
      RETURN
      END
C
      SUBROUTINE VECSUB(S,V,R)
      REAL S(300),V(300),R
      DO 100 I= 1,300
       R= R+ S(I)-V(I)
  100 CONTINUE
      RETURN
      END
```

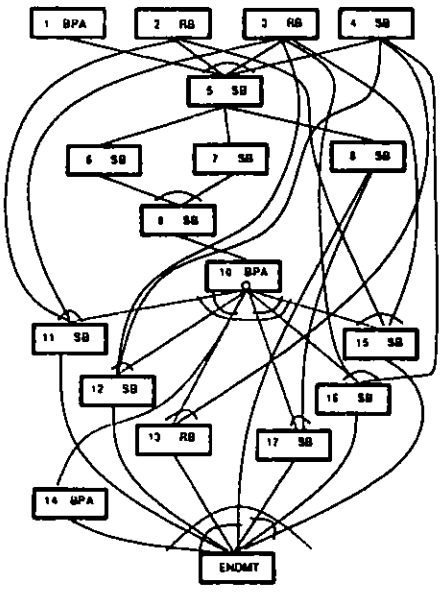
Fig.8 A sample program with 17 macrotasks.

This section briefly introduces performance of OSCAR compiler. Fig.8 is a sample FORTRAN program with 17 macrotasks including RBs, SBs and BPAs. Fig.9(a) is a macroflow graph of the program. Fig.9(b) represents a macrotask graph for the macroflow graph. This macrotask graph shows the parallelism extracted from Fig. 9(a).

The execution time of the program on OSCAR was as follows. When the program was sequentially executed by 1 PE, the processing time was 9.63s. The execution time for the macro-dataflow computation using 3 PEs was 3.32s. The processing time for the multi-grain computa-



(a) A macroflow graph.



(b) A macrotask graph.

Fig. 9 Macroflow and macrotask graphs for Fig. 8.

tion using 3 PCs, each of which has 2 PEs., namely, 6 PEs, was 1.83s. It was also observed from execution traces that the dynamic scheduling overhead was negligibly small.

5. CONCLUSIONS

This paper has described OSCAR FORTRAN parallelizing compiler very briefly. The compiler realizes multi-grain parallel processing, which combines the macro-dataflow computation, the loop concurrentization and the near fine grain parallel processing. The prototype compiler with some restrictions has been working on OSCAR. Currently, the authors are enhancing the prototype version to a practical version, which allows us to realize the combination of the macro-dataflow and the loop concurrentization on a multiprocessor supercomputer.

REFERENCES

[1] A.V.Aho, R.Sethi and J.D.Ullman, Compilers: Principles, Techniques, and Tools, Addison Wesley, 1988.

[2] U.Banerjee, Dependence Analysis for Supercomputing, Kluwer Pub., 1988

[3] D.A.Padua, D.J.Kuck and D.H.Lawrie,"High-speed multiprocessor and compilation techniques," IEEE Trans. Comput., Vol. C-29, No.9,pp.763-776, Sep. 1980.

[4] D.A.Padua, and M.J.Wolfe,"Advanced Compiler Optimizations for Supercomputers," C.ACM, Vol.29, No.12, pp.1184-1201,Dec.1986.

[5] D.Gajski, D.Kuck, D.Lawrie and A.Sameh, "CEDAR," Report UIUCDCS-R-83- 1123, Dept. of Computer Sci., Univ. Illinois at Urbana-Champaign, Feb. 1983.

[6] D.D.Gajski, D.J.Kuck, D.A.Padua, "Dependence Driven Computation," Proc. of COMPCON 81 Spring Computer Conf., pp.168-172, Feb. 1981.

[7] H.E.Husmann, D.J.Kuck and D.A.Padua,"Automatic Compound Function Definition for Multiprocessors," Proc. 1988 Int"l. Conf. on Parallel Processing,Aug.1988.

[8] M.Wolfe, "Multiprocessor synchronization for concurrent loops," IEEE software, Vol. pp. 34-42, Jan. 1988.

[9] M.Wolfe,Optimizing Supercompilers for Supercomputers,MIT Press, 1989.

[10] C.D.Polychronopoulos and D.J.Kuck, "Guided self-scheduling : A practical scheduling scheme for parallel supercomputers," IEEE Trans. Comput., Vol.c-36,12, pp.1425-1439,Dec. 1987.

[11] D.J.Kuck, E.S.Davidson, D.H.Lawrie and A.H.Sameh, "Parallel Supercomputing Today and Cedar Approach," Science, Vol.231, pp.967-974, Feb. 1986.

[12] J.A.Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code," IEEE Computer, Vol. 17,No.7, pp.45-53, Jul.1984.

[13] R.P.Colwell, et.al.,"A VLIW Architecture for a Trace Scheduling Compiler," IEEE Trans. Comp., Vol.C-37, No.8, pp.967-979, Aug.1989.

[14] J.R.Ellis, "Bulldog: A Compiler for VLIW Architectures," MIT Press,1985.

[15] A.Nicolau and J.A.Fisher, "Measuring the Parallelism Available for Very Long Instruction Word Architectures," IEEE Trans. on Computers, Vol. C-33, No. 11, pp.968-976, Nov.1984.

[16] H.Kasahara et.al. "A Multi-grain Parallelizing Compilation Scheme on OSCAR," Proc. 4th Workshop on Languages and Compilers for Parallel Computing, Aug. 1991.

[17] N.P.Jouppi, "The Nonuniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance," IEEE Trans. on Comput., vol. C-38, No.12, pp.1645- 1657, Dec.1989.

[18] E.G.Coffman Jr.(ed.), Computer and Job-shop Scheduling Theory. New York : Wiley, 1976.

[19] M.R.Garey and D.S.Johnson, Computers and Intractability : A Guide to the Theory of NP-Completeness. San Francisco : Freeman, 1979.

[20] C.D.Polychronopoulos, Parallel Programming and Compilers, Kluwer Academic Pub., 1988.

[21] V.Sarkar, "Determining Average Program Execution Times and Their Variance'", Proc. Sigplan'89, June 1989.

[22] V.Sarkar, Partitioning and Scheduling Parallel Programs for Multiprocessors, MIT Press,1989.

[23] H.Kasahara and S.Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," IEEE Trans. Comput., Vol.c-33, No.11,pp. 1023-1029,Nov.1984.

[24] H.Kasahara and S.Narita, "An approach to supercomputing using multiprocessor scheduling algorithms, " in Proc. IEEE 1st Int'l Conf. on Supercomputing, pp.139-148,Dec. 1985.

[25] F.Allen, M.Burke,R.Cytron,J.Ferrante,W.Hsieh and V.Sarkar, "A Framework for Determining Useful Parallelism," Proc. 2nd ACM Int'l. Conf. on Supercomputing, 1988.

[26] J.Ferrante,K.J.Ottenstein,J.D.Warren,"The Program Dependence Graph and Its Use in Optimization," ACM Trans. on Prog. Lang. and Syst, Vol.9,No,3.pp.319-349, July 1987.

[27] B.S.Baker,"An Algorithm for Structuring Flowgraphs," J. ACM, Vol.24, No.1, pp.98-120, Jan.1977.

[28] M.Burke and R.Cytron, "Interprocedural Dependence Analysis and Parallelization," Proc. ACM SIGPLAN'86 Symposium on Compiler Construction, 1986.

[29] M.O'Keefe and H. Dietz, "Hardware Barrier Synchronization: Static Barrier MIMD," Proc. 1990 Int'l Conf. on Parallel Processing, pp. I35-42, Aug. 1990.

[30] H.Kasahara, H.Honda, S.Narita, "Parallel Processing of Near Fine Grain Tasks Using Static Scheduling on OSCAR," in Proc. IEEE ACM Supercomputing'90, Nov. 1990.

[31] H.Honda, M.Iwata, H.Kasahara, "Coarse Grain Parallelism Detection Scheme of Fortran programs," Trans. IEICE, Vol.J73-D-I, No,12, Dec.1990 (in Japanese).

[32] F.G.Gustavson, W.Liniger and R.A.Willoughby, "Symbolic Generation of an Optimal Crout Algorithm for Sparse Systems of Linear Equations," J.ACM, vol.17, pp.87-109, Jan. 1970.

[33] H.Kasahara, Parallel Processing Technology, Corona Publishing, Tokyo, (in Japanese), Jun. 1991.

[34] S.S.Munshi and B.Simons, "Scheduling Sequential Loops on Parallel Processors," SIAM J. Comput., Vol. 19, No.4, pp.728-741, Aug., 1990.

[35] M.Girkar and C.D.Polychronopoulos, "Optimization of Data/Control Conditions in Task Graphs," Proc. 4th Workshop on Languages and Compilers for Parallel Computing, Aug. 1991.

[36] H.Kasahara, H.Honda, M.Iwata and M.Hirota, "A Macro-dataflow Compilation Scheme for Hierarchical Multiprocessor Systems," Proc. Int'l. Conf. on Parallel Processing, Aug. 1990.

[37] H.Kasahara, W.Premchaiswadi, M. Tamura, Y.Maekawa and S.Narita, "Parallel Processing of Sparse Matrix Solution Using Fine Grain Tasks on OSCAR," Proc. Int'l. Conf. on Parallel Processing, Aug. 1991.

Coordination Language Design and Implementation Issues

Steve Lucco and Oliver Sharp, UC Berkeley

One can think of a parallel program as a group of sequential sub-computations which cooperate to solve a problem. To exploit existing code and optimization tools, programmers usually choose to write these sub-computations in traditional imperative languages such as C, Fortran, or Lisp. A *coordination language* expresses data exchange and synchronization among such sub-computations. We are investigating the effectiveness of coordination languages as tools for concise expression and efficient implementation of parallel programs. In our presentation, we will outline some design and implementation issues critical to the expressiveness and performance of coordination languages. Using these criteria, we will compare existing coordination languages including Strand, PCN, Linda, Jade, Delirium, and extended Fortran dialects.

Most existing coordination languages, including Linda and the extended Fortran dialects, are *embedded*; they consist of a set of non-deterministic coordination primitives which are added to a host language program. We are investigating a new type of coordination language, which we call an *embedding* language. Our language, Delirium, is one example; Strand and PCN have also been used as embedding coordination languages.

An embedding language program is a separate text that specifies a framework for accomplishing a task in parallel; sequential sub-computations called *operators* are embedded within that framework. This organizing principle makes parallelization easier. Instead of scattering coordination throughout a program, creating a set of ill-defined sub-computations, a coordination language programmer precisely defines operators and embeds these operators within a parallelization framework. One can completely discover the topology of the program's parallel execution simply by reading its coordination code.

This type of coordination language has four advantages over embedded languages. First, because they are separate program texts, embedding language programs can express synchronization using a unified notation such as a functional or logic language. In contrast, extended Fortran dialects and languages like Linda consist of a collection of discrete synchronization primitives. Second, embedding notations support hierarchical abstraction of coordination. One can create and re-use complex patterns, such as binary reduction, through functional abstraction. Embedded languages can't support abstraction because individual embedded primitives appear as separate statements within a host language program, and they can only indirectly control the execution order of other statements in the program. Third, embedding notations have a coordination *semantics* that is distinct from the semantics of the language in which computation is expressed. Finally, sub-computations are encapsulated. That is, they have unique, well-defined entry and exit points. Debugging is

easier because individual sub-computations can be tested in isolation, on the target parallel machine or a more familiar sequential one.

On the other hand, embedding coordination languages are more difficult to implement than embedded languages. The primitives of embedded languages generally have a direct translation to operations on an abstract machine or the underlying multiprocessor, whereas existing embedding languages are declarative; they specify a dataflow and require the coordination compiler to map that dataflow onto the underlying machine.

Further, because they exist as a separate text, embedding language programs do not have direct access to the data dependence semantics of the sequential language. On distributed memory architectures, it becomes difficult for the coordination compiler to balance computational load or even enforce correct behavior without this information. Some preprocessing of the sequential language program is necessary to provide this information to the coordination compiler. This information can be presented and processed in the form of annotations such as those defined in Jade and in Fortran D.

We believe that any attempt to use a coordination language as a tool for specifying parallel program behavior will evolve into a *coordination system* with the following components:

- A coordination language and coordination compiler which includes explicit support for data parallelism.
- A sequential language (Fortran, C, C++) preprocessor that gathers information and transforms sequential programs into sets of operators.
- An annotation language for transmitting and presenting information gleaned by the preprocessor to the coordination compiler (and the user). Ours is called Dossier and is intended to be readable by humans as well as by programs.
- A runtime scheduling system that can exploit opportunities for fine-tuning that are discovered by the coordination compiler. There are a variety of optimizations that a run time system can perform, including the adjustment of grain size based on execution behavior and the pipelining of adjacent loop nests.

We have successfully implemented several large, irregular parallel programs on both shared and distributed memory multiprocessors using this organization. During the talk, we will present a case study that shows how each piece of the system contributes towards the final goal of achieving efficient execution on a variety of architectures.

Designing Imperative Programming Languages for Analyzability: Parallelism and Pointer Data Structures

Laurie J. Hendren*
Guang R. Gao
School of Computer Science
McGill University

Abstract

The rapid advance of computer architectures has provided important new challenges for programming language designers and compiler writers alike. One of these challenges is to provide programming languages that are *analyzable* so that compilers can effectively exploit the level of parallelism that is necessary for effective use of such architectures. Historically, the analysis of scientific programs using arrays has made crucial use of information such as array dimension and size, the mathematical structure of arrays, and the regularity of the looping constructs used for programming with arrays. In this paper we argue that, just as arrays are important in scientific programs, pointer data structures often play a central role in non-scientific and symbolic programs, and therefore the analyzability of pointer data structures is critical.

To illustrate both the importance of *analyzability* for programs with pointer data structures and the solution methodology proposed in this paper, we propose a programming language mechanism which significantly enhances the analyzability of pointer-based data structures frequently used in non-scientific programs. Our approach is based on exploiting two important properties of pointer data structures: *structural inductivity* and *speculative traversability*. Structural inductivity facilitates the application of a static interference analysis method for such pointer data structures based on *path matrices*, and speculative traversability is used to exploit parallelism by allowing aggressive traversals of linked pointer data structures.

In this paper we give an overview of our approach to designing analyzable imperative languages. We give a concrete example of applying the techniques to exploit fine-grain parallelism in while loops that traverse linked list structures. The effectiveness of our approach is demonstrated by applying it to a collection of loops found in typical non-scientific C programs. In addition, we outline how our approach can be used to exploit coarse-grain parallelism, and we give some challenges open in this area.

1 Introduction

In the past decade, the dramatic improvement of VLSI technology has led to modern high-performance microprocessors that support some level of fine-grain parallelism. Even with today's RISC processors, some degree of instruction-level parallelism is required to fully utilize the architecture. This increase in fine-grain parallelism, and the development of parallel architectures supporting more coarse-grain parallelism has has provided important challenges for programming language designers and compiler writers alike. It is becoming increasingly important to provide programming languages and associated compiler support such that user programs can be effectively *analyzed* in order to effectively utilize the underlying architectures. In particular, it is critical to provide compile-time analysis that results in accurate alias analysis and data-dependency information for complex data structures. This paper argues that such *analyzability* is an important principle of program language design and implementation, and that it is particularly critical for the efficient mapping of non-scientific programs to architectures supporting some level of parallelism.

*The work supported in part by FCAR, NSERC, and the McGill Faculty of Graduate Studies and Research.

The importance of the analyzability has been demonstrated by the considerable success of automatic parallelization and optimization of large-scale scientific numerical programs. In such scientific programs, arrays are the most important data structures, and the programs using these array structures can often be analyzed effectively. The keys to such analysis are:

- Arrays are frequently defined on rectangular index regions with dimensions, shapes, and bounds known to the compiler.
- Array operations are often encapsulated in "well structured" loops (**for** loops without **gotos**) with the iteration space of the loop matching the index regions of the arrays.
- In such loops, the arrays are frequently accessed ("traversed") in a regular fashion. For example, the index expression is often an affine function of the loop indices.

The mathematical structure of the arrays and the regularity of their accesses in embedded loops has lead to the development of a variety of *dependence analysis*, *loop transformation*, and *parallelization* techniques [Ban76, Ban88, ABC86, ACK87, KKP+80, PW86, Wol89]. Although many of these techniques were pioneered in the areas of vectorizing and parallelizing compilers, these techniques are also being applied to architectures supporting instruction-level parallelism. There has also been work in programming language design to enhance the analyzability of arrays [X3J90, HWe90, GYDM90, ANP89].

Unfortunately, the analysis and optimization of non-scientific programs has not been so successful. In this paper we are interested in the analyzability issues for real life non-scientific programs. There is little dispute that many such programs are currently written in imperative languages like C, and there is no sign that this trend will slow down soon. In such programs, dynamically-allocated pointer data structures play a central role. To fully exploit parallelism in such programs on we need to provide mechanisms for specifying properties of such data structures that can be used to improve the compiler analysis.

To illustrate both the importance of *analyzability* for programs with pointer data structures and the solution methodology proposed in this paper, we propose a programming language mechanism which significantly enhances the analyzability of pointer-based data structures frequently used in non-scientific programs. Our approach is based on two important properties : *structural inductivity* and *speculative traversability*. Structural inductivity facilitates the application of a static interference analysis method for such pointer data structures based on *path matrices*, and speculative traversability is utilized by a novel loop unrolling technique for while loops that exploits fine-grain parallelism by aggressively traversing such data structures. The effectiveness of this approach is demonstrated by applying it to a collection of loops found in typical non-scientific C programs. For high-performance RISC architectures, our approach resulted in a speedup of 1.17 and 1.43 over the optimized code produced by the native C compilers on SUN SPARC-based machine, and a DEC MIPS-based machine respectively. We also illustrate how our approach can also be used to expose more instruction-level parallelism for architectures supporting multiple instruction issuing/processing such as superscalar/superpipelined or VLIW machines.

The paper is organized as follows. In section 2 we present a more detailed outline of the challenges of compiling imperative programs with pointer data structures for machines with instruction-level parallelism. In section 3, we introduce programming language constructs which enhance the analyzability of programs by providing the programmer with a means of specifying two important properties of pointer data structures frequently found in non-scientific programs: *speculative traversability* and *structural inductivity*. In section 4 we give a concrete case study to show how such mechanisms can be effectively applied to unroll while loops that traverse such data structures. Also in section 4 we present the results of applying the method to optimize a collection of loops for RISC machines, and we show the applicability of the techniques for exposing instruction-level and coarse-grain parallelism. Lastly, we provide our conclusions in section 5.

2 Parallelism in the Presence of Pointers

In this section we outline the challenges of compiling for parallelism in the presence of pointer data structures. In the first subsection we review the techniques currently used for instruction scheduling for instruction-level parallelism

and we show how inaccurate alias analysis for pointer structures can greatly reduce the effectiveness of these techniques. In the second subsection we discuss a high-level loop transformation that is used to increase instruction-level parallelism for programs with arrays, and we discuss the difficulties of applying similar transformations to pointer data structures.

2.1 Instruction Scheduling

In compiling for architectures with instruction-level parallelism, instruction scheduling is a crucial component. That is, in order to effectively utilize the architecture, the compiler must take a sequential list of instructions, and rearrange the instructions in such a way as to increase parallelism (reduce execution time) while preserving the original meaning of the program.

Instruction scheduling is commonly performed for RISC architectures [Kri90]. In pipelined RISC architectures, it is desirable to arrange the code such that successive instructions can be issued to the instruction pipeline one per cycle Some operations, like a read from memory or a floating-point arithmetic operation, require more than one machine cycle to complete. However, one does not necessarily have to wait for an instruction i to complete before scheduling the next instruction j. Due to the pipelined design of the architecture, it is possible to schedule instruction j one cycle after another instruction i if j does not depend on i. Overlapping the execution of instructions in this way decreases the total execution time by introducing one form of instruction-level parallelism. Another type of instruction-level parallelism is present in architectures such as VLIW or superscalar machines which allow more than one operation to be scheduled at the same time. In such architectures the compiler must analyze the program in order to determine which operations may safely be issued in parallel.

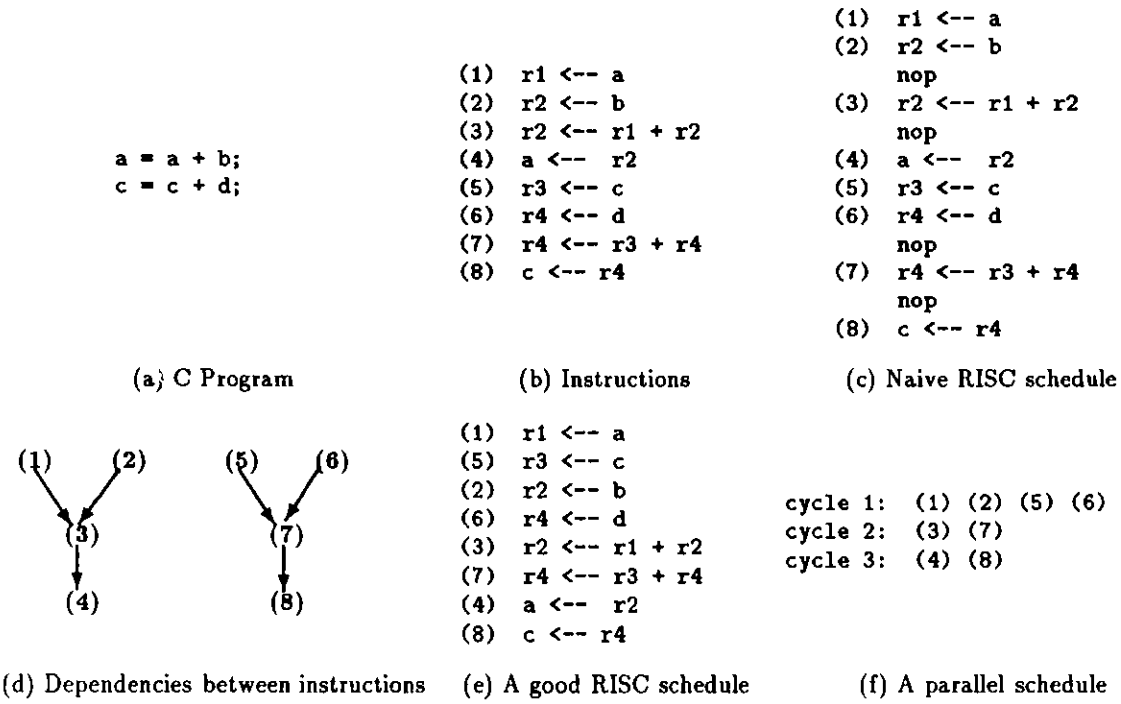


Figure 1: Scheduling example 1: a and c are scalar variables

We illustrate instruction scheduling with the small example program given in figure 1(a). Translating this program into machine instructions yields the sequential list of 8 machine-level instructions as given in figure 1(b)[1]. Now consider the problem of mapping these 8 instructions to a RISC architecture that requires two cycles for a

[1] Note that we use the notation a <-- r1 to indicate storing r1 into the memory location for variable a, and r1 <-- a to indicate loading r1 from the memory location for variable a. The actual memory location for a could be given by an offset to the stack pointer or frame pointer, or an address of a global.

load from memory and two cycles for an addition. If we take the naive approach of scheduling the instructions in the same order as we generated them in figure 1(b), then the best that we can do for our RISC architecture is the schedule given in 1(c). Note that due to the 2 cycle latency of loads and additions, we had to insert 4 nop instructions. For example, we had to insert a **nop** between instructions (2) and (3) because we needed an extra cycle to wait for the load to **r2** to complete before we could compute **r1 + r2**. We can improve upon the schedule of figure 1(c) by noting that the 8 instructions do not necessarily need to be executed in the order that they were generated. For example, if **a** and **b** are different variables, then the load of **a** does not depend on (does not need to proceed) the load of **b**. Thus, if we have a compile-time analysis that can determine that **a**, **b**, **c**, and **d** are all distinct variables (they all refer to different memory locations), then we can represent the partial order with the dependency diagram given in figure 1(d). Starting from this partial order we can produce a much better schedule (8 cycles instead of 12 cycles) as shown in 1(e). This improved schedule requires no extra **nop** instructions since all load and add instructions are scheduled at least two cycles before their results are needed. We can also use the partial order to produce a parallel schedule suitable for an architecture that can issue more than one instruction at the same time. Figure 1(f) gives a parallel schedule that can issue 2-4 operations simultaneously at each cycle.[2]

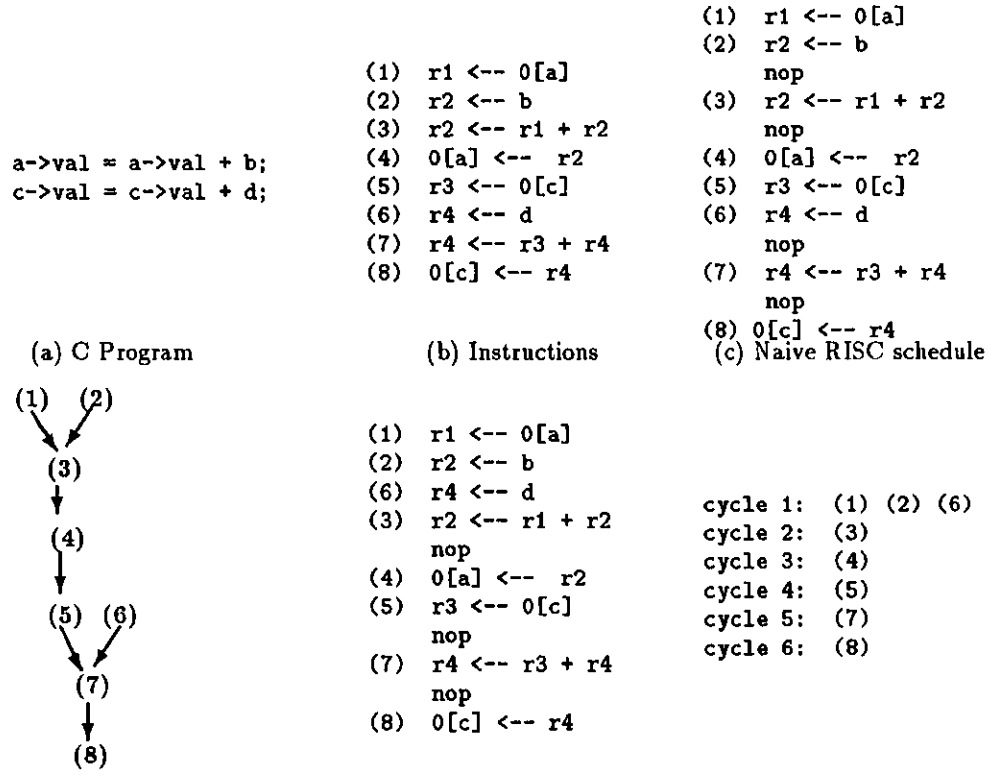


Figure 2: Scheduling example 2: **a** and **c** are pointer variables

This small example illustrates that the compiler can exploit instruction-level parallelism if it can transform a sequential list of instructions to a partial ordering of instructions and then apply a scheduling algorithm on this partial order. However, as illustrated in the example given in figure 2, the success of this approach relies heavily on accurate compile-time alias analysis. In our first program, each of the variables were scalars, and therefore well-established compile-time techniques can be used to determine that **a** and **c** refer to different memory locations (that is, **a** and **c** are not aliased)[3]. However, in the program given in figure 2(a), **a** and **c** refer to dynamically-allocated pointer data structures. As indicated in the instructions given in figure 2(b), the memory reference to

[2]This is the schedule if we assume operation latencies of 1. If we assume the same latencies as in the RISC case, then there will be empty slots between the cycles. The compiler may move more independent operations to these slots, thus exploring more instruction-level parallelism.

[3]In fact, often times this analysis can be as simple as determining that **a** and **c** refer to different offsets on the stack.

a->val and c->val is through an extra level of indirection.[4] Due to this indirection it is difficult to determine whether or not a->val and c->val refer to the same memory location, and compilers are not, in general, able to perform this analysis at compile-time. As illustrated by figure 2(d), this lack of precise alias analysis for pointer data structures leads to the introduction of possibly spurious edges in the dependency graph. For example, if we cannot determine whether or not a->val and c->val refer to the same location, then we must introduce a dependency between instructions (4) and (5). The negative effect of such extra dependencies is illustrated clearly with the RISC schedule given in figure 2(e) and the parallel schedule in figure 2(f). In both cases the schedules are considerably worse than the equivalent schedules for the scalar case given in figures 1(e) and 1(f).

2.2 Loop Transformations to Increase Available Parallelism

There has been considerable effort spent on developing compile-time techniques that transform scientific programs in such a way as to expose more parallelism. In order to introduce the notion and benefit of loop transformations, let us consider *loop unrolling*, a transformation technique that was developed for parallelizing and optimizing compilers for scientific programs and arrays [DH79]. As shown by the example in figure 3, loop unrolling essentially transforms a `for` loop into an equivalent `for` loop in which there are multiple copies of the body. Loop unrolling was initially developed as a technique for reducing loop overhead and for exposing instruction-level parallelism for machines with multiple functional units. More recently it has also been applied in conjunction with instruction scheduling for pipelined and RISC architectures [WS87, Sri91, Muk91]. By increasing the size the body of the loop, the instruction scheduler can often produce a shorter schedule for the unrolled loop.

```
for (i = 1; i <= 120 ; i++)
  a[i] = a[i] * b + c;
```

Original Loop

```
for (i = 1; i <= 120; i = i + 3)
  { a[i] = a[i] * b + c;
    a[i+1] = a[i+1] * b + c;
    a[i+2] = a[i+2] * b + c;
  }
```

Unrolled Loop

Figure 3: Loop Unrolling for Scientific Programs

Now let us consider the problem of performing a similar transformation on while loops with pointer data structures. In figure 4, we give three program fragments that were extracted from the source code for the GNU C compiler[5]. The first loop, *initialize*, simply traverses a list initializing each key field to y. The second loop, *last_item*, traverses a list to find the last item. The third loop, *reverse*, destructively reverses the list.

```
while (lp != LIST_NULL)
  { lp->key = y;
    lp = lp->next;
  }
```

Initialize each element

```
if (lp != LIST_NULL)
  { next_lp = lp->next;
    while (next_lp != LIST_NULL)
      { lp = next_lp;
        next_lp = next_lp->next;
      }
  }
last = lp;
```

Find last item

```
prev = LIST_NULL;
while (lp != LIST_NULL)
  { next_lp = lp-> next;
    lp->next = prev;
    prev = lp;
    lp = next_lp;
  }
lp = prev;
```

Reverse the list

Figure 4: Three typical loops on pointer data structures

At first glance it would appear that the loops in figure 4 are not really suited to techniques like loop unrolling. The potential problems include:

[4] We use the statement r1 <--0[a] to indicate that r1 should be loaded from the memory location that is at an offset of 0 from the location stored in variable a.

[5] We have modified the presentation of these loops so that they all use the same form and variable names.

- In the case of arrays one can compute the index of array elements and reference many elements in parallel (`a[i]`, `a[i+1]`, `a[i+2]`). However, the pointer loops appear to be inherently sequential, and the list must processed by traversing each element in turn.
- In the array example it was straightforward to compute the loop bounds and increment for the unrolled loop. In the case of pointer loops we have no idea of the length of the structure, and no simple way to construct a termination condition that results in an equivalent unrolled loop.
- Even if we could access more than one item at a time, it would appear that loops like *last_item* have no computation to do on each item, and therefore may not be good candidates for loop unrolling.
- In the case of arrays the loop body will only modify the values within the structure, but not the structure itself. That is, the shape and size of the arrays remain the same. In the case of linked structures the loop may actually change the structure of the list itself. This is true in the case of *reverse*.

3 Analyzable Pointer Data Structures

As illustrated in the previous section, there are many challenges for effectively compiling programs for instruction-level parallelism. In particular, we illustrated the negative effect of poor alias analysis for pointer data structures, and the difficulties encountered in designing loop transformations for pointer data structures. These difficulties arise because unlike data structures such as scalars and arrays which have a fixed shape and size, pointer data structures have dynamically changing shapes and sizes. In addition, programmers use dynamic data structures to implement a wide variety of structures including singly-linked lists, doubly-linked lists, circular lists, nested lists, binary trees, threaded trees, and graphs. Even though the programmer may use pointers in a very constrained manner (for example, the programmer may use a certain type of node to build only non-cyclical lists), the compiler has no way of knowing what sort of structure the programmer has in mind, and therefore the compiler cannot exploit any properties that are specific to that structure. In addition, the compiler has no knowledge of the size or length of a dynamic data structure.

In this section we introduce two characteristics of pointer data structures that allow better compile-time analysis. That is, we give some examples of how to make a programming language more *analyzable*. In the first sub-section we introduce the class of *structurally inductive* data structures and we illustrate how we can use *path matrix analysis* to provide accurate alias analysis for this class of data structures. In the second sub-section we introduce *speculative traversability*, a property that allows us to perform loop transformations without knowing the length of a data structure.

For practical reasons, including the wide-spread use of C for non-scientific programming, we have chosen to develop our approach relative to the programming language C. In fact, our techniques require only very small syntactic extensions to C that could be implemented with a straight-forward preprocessing step. However, one should not assume that the notion of designing analyzable programming languages is restricted to C, and we hope that others will take the challenge of designing better, analyzable programming languages that are suitable for architectures supporting instruction-level parallelism.

3.1 Structurally Inductive Pointer Data Structures and Alias Analysis

In order to provide more accurate alias analysis for pointer data structures, we would like programmers to be able to classify their data structures as either *inductive* or *non-inductive*. An *inductive* linked structure is one in which there are no cycles, and each node has at most one parent. *Inductive* structures include linked lists, nested lists, and trees, while *non-inductive* structures include DAGs and cyclic graphs. Inductive structures have nice properties for analysis, and techniques for alias analysis, dependence analysis, and parallelizing transformations for this class of structures have been developed [Gua88, LH88, HN90]. The exploitable properties of *inductive* structures include: (1) the component pieces of the structure (the head and tail of a list, or the left and right sub-trees of a binary tree) never share any storage, and thus computations on the components are non-interfering, (2) breaking any link yields two independent pieces, and (3) a traversal of any series of linked nodes never revisits the same node more than once.

Our approach to exploiting the analyzability of inductive structures is to allow the programmer to indicate which pointer data structures have this property. In order to discuss our language extension with respect to C, we introduce a high-level recursive type statement, **rectype**. Each rectype statement consists a list of field names, with each field having either a scalar type, or a recursive reference to the type being defined. As illustrated by the example for linked lists as given in figure 5, the **rectype** statement can be translated by the compiler into the traditional recursively defined pointer structures in C. Note that the compiler can generate both the appropriate C types for nodes and pointers to the nodes, and constant representing the the empty structure (LIST_NULL).

```
rectype LIST [inductive]
{ int key;
  LIST next;
}
```

(a)

```
#DEFINE LIST_NULL 0   /* name of the empty list */

typedef struct LIST   /* node type for ordinary lists */
 { int key;
   struct LIST *next;
 } LIST_NODE, *LIST_PTR;
```

(b)

Figure 5: A **rectype** and its C implementation

As illustrated in figure 5(a), we provide the option of indicating if a particular **rectype** has the *inductive* property. It should be noted that both *inductive* and *non-inductive* recursive structures are implemented with the same C data types. Thus, we treat the inductive specification as a directive, rather than a type. The programming language designer has a choice of how to use this directive. The easiest, and **least** safe, approach is to take this directive as a promise from the programmer that all structures built with this type will be inductive. This is analogous to allowing arrays without bounds checking. Another, and perhaps preferable, approach is to use the directive to aid the compiler in choosing the correct sort of alias analysis to perform for that data type. Thus, an analysis best suited to inductive structures can be used for each **rectype** which has been labeled as inductive. Just as dependence analysis is a natural choice for analyzing array references, *path matrix* analysis is a natural choice for inductive structures [HN90].

Path matrix analysis is an interprocedural analysis technique that was specifically developed for inductive data structures. The technique exploits the special properties of inductive structures in order to provide accurate static analyses to: (1) safely determine if pointer data structures are guaranteed to be *inductive*, (2) perform alias analysis and dependency analysis for all pointer variables that point to nodes of the inductive structure, and (3) detect non-interfering computations.

We illustrate the use of path matrix analysis for the program fragment presented in figure 6. This program first calls a function to build a list, and then executes a **while** loop to reverse the list. At the beginning of the loop, **orig_lp** points to the first item, while at the end of the loop, **new_lp** points to the first item of the reversed list, and **orig_lp** points to the last cell of the reversed list.

```
orig_lp = build_list(n);  /* build a list with n items */
lp = orig_lp;              /* orig_lp points to original head of list */
prev = LIST_NULL;
while (lp != LIST_NULL)
  { next_lp = lp->next;  /* get next node */
    lp->next = prev;     /* reverse link of lp */
    prev = lp;           /* get ready for next iteration */
    lp = next_lp;
  }
new_lp = prev;           /* new_lp points to new head of list, and
                            orig_lp now points to last element */
```

Figure 6: An example program with the *reverse* loop

In order to demonstrate how the analysis works, we used our path matrix analysis tool[6] to process the program

[6]This is an interprocedural analysis tool that is written in SML [HN90, Hen90].

(1) orig_lp:=build_list(n)

	$orig\_lp(\bullet,\odot)$
$orig\_lp$	S

(2) lp = orig_lp

	$orig\_lp(\bullet,\odot)$	$lp(\bullet,\odot)$
$orig\_lp$	S	S
lp	S	S

(3) prev = LIST_NULL

	$orig\_lp(\bullet,\odot)$	$lp(\bullet,\odot)$	$prev(o,o)$
$orig\_lp$	S	S	
lp	S	S	
$prev$			S

========== LAST ITERATION OF WHILE LOOP FIXED-POINT CALCULATION ==========

(4) next_lp = lp->next

	$orig\_lp(\bullet,o)$	$lp(\bullet,\odot)$	$prev(\bullet,\odot)$	$next\_lp(\odot,\odot)$
$orig\_lp$	S		$S?$	
lp		S		N^1
$prev$	$(S+N^+)$		S	
$next\_lp$				S

(5) lp->next = prev

	$orig\_lp(\bullet,o)$	$lp(\bullet,\bullet)$	$next\_lp(\odot,\odot)$
$orig\_lp$	S		
lp	N^+	S	
$next\_lp$			S

(6) prev = lp

	$orig\_lp(\bullet,o)$	$prev(\bullet,\bullet)$	$next\_lp(\odot,\odot)$
$orig\_lp$	S		
$prev$	N^+	S	
$next\_lp$			S

(7) lp = next_lp

	$orig\_lp(\bullet,o)$	$lp(\odot,\odot)$	$prev(\bullet,\bullet)$
$orig\_lp$	S		
lp		S	
$prev$	N^+		S

========== END WHILE/REPEAT ==========

(8) new_lp = prev

	$orig\_lp(\bullet,\odot)$	$new\_lp(\odot,\odot)$
$orig\_lp$	S	$S?$
new_lp	$(S+N^+)$	S

Figure 7: Partial trace of the path matrix computations

given in figure 6 and we extracted pieces from the trace of the analysis as given in figure 7. For each program point in the trace, we give the path matrix that summarizes the relationships (or paths) that exist between each pair of pointer variables (also called *handles*) live at that point. Also, note that each handle is decorated with a pair that looks something like *(•,⊙)*. The first item of the pair corresponds to "nilness" of the handle itself, and the second item corresponds to the "nilness" of the *next* field of the handle (• means definitely not nil, ∘ means definitely nil, and ⊙ means that it could be either). Let us consider the following points in the path matrix analysis.

Program Point (1): The first statement in our program fragment is a call to a procedure `build_list` that builds a non-cyclical list. Although we have not shown the path matrix analysis for this call, we see that the resulting path matrix contains only one live handle, *lp*, and it has the relationship S with itself. Note that the relationship S between two handles x and y indicates that these variables point to the *same* node. The fact that there is only one entry in the path matrix indicates that there are no other live pointer variables (handles) that can be reached from the head of the list *lp*, and that the analysis of the procedure call successfully determined that the structure pointed to by *lp* is indeed inductive. If the analysis of the call had not been successful (for example, the programmer may have created a cyclic structure), then a less accurate alias analysis mechanism must be used.

Program Point(2): The path matrix computed for program point 2 contains two handles, *lp* and *orig_lp*. Note that the entries reflect the fact that *lp* and *orig_lp* point to the same node.

Program Point(3): In statement 3 we see the first occurrence of the nil pointer (`LIST_NULL`). You can see that the handle *prev* is definitely nil, and it is not related to any other handle.

Program Points(4) to (7): We have given the path matrices computed for the last iteration of the while loop fixed-point calculation. You can think of these path matrices as approximating the state of the data structures for all iterations of the while loop after the first iteration. Note that at program point (4) the relationship between *lp* and *next_lp* is always N^1 (exactly one *next* link), while the relationship between *prev* and *orig_lp* is $(S + N^+)$ (they point to the same node, or there is a chain of one or more *nexts* from *prev* to *orig_lp*). Also note that there is *no* relationship between *prev* and *lp*. This corresponds to the fact that the original list is now split into two distinct pieces, the reversed part starting at *prev* and ending at *orig_lp*, and the unreversed part starting at *lp*.

Program Point(8): Finally, at program point (8), we see the relationship of $(S + N^+)$ between the new head of the list *new_lp* and the original head *orig_lp*. The S corresponds to the case when the original list only has only one item, and the N^+ corresponds to all cases for lists with more than one item.

This example shows that we can get very accurate alias analysis for inductive structures when we apply alias analysis techniques that have been carefully designed to take advantage of the special properties of such structures. Indeed, the analysis provides accurate information even though the structure is being destructively traversed. This illustrates the point that one must be able to capture special properties of data structures at the programming language level, so that the appropriate compiler analysis techniques can be developed and used for those structures.

3.2 Speculatively Traversable Pointer Data Structures

In addition to providing properties that lead to better alias analysis, we must also deal with the problem of not knowing the length or size of pointer data structures. In order to fully demonstrate this problem, let us return to the problem of unrolling `while` loops. Recall that `for` loops that operate on arrays can be easily unrolled by simply modifying the counter and termination conditions of the loop. However, with while loops on pointers, the situation is much more difficult. Consider for example the *initialize* loop given in figure 8(a).

In order to effect some sort of unrolling, we can try the brute-force approach of duplicating the body of the loop, along with the appropriate conditionals. Figure 8(b) gives an example of this approach for unrolling the *initialize* loop 3 times. Although clearly semantically equivalent to the original loop, this approach does not appear to improve the code. The loop overhead is not improved because we need to do just as many tests, and the instructions in the body remain sequential.

In figure 8(c) we give a good strategy for unrolling this loop. In this case two extra list pointers, `lp2` and `lp3` are introduced to give three independent pointers into the list. As a result, the three statements updating the fields

```
while (lp != LIST_NULL)
  { lp->key = y;
    lp = lp->next;
  }
```

(a) the original loop

```
while (lp != LIST_NULL)
  { lp->key = y;
    lp = lp->next;
    if (lp != LIST_NULL)
      { lp->key = y;
        lp = lp->next;
        if (lp != LIST_NULL)
          { lp->key = y;
            lp = lp->next;
          }
      }
  }
```

(b) *The brute-force approach*

```
lp2 = lp->next;
lp3 = lp2->next;
while (lp3 != LIST_NULL)
  { lp->key = y;
    lp2->key = y;
    lp3->key = y;
    lp  = lp3->next;
    lp2 = lp->next;
    lp3 = lp2->next;
  }
while (lp != LIST_NULL)
  { lp->key = y;
    lp = lp->next;
  }
```

(c) *A good unrolling*

Figure 8: Loop Unrolling for the *initialize* example

`lp->key`, `lp2->key` and `lp3->key` can be executed in parallel, and the number of tests is reduced to 1/3 of the original loop. However, this unrolled loop is not necessarily semantically equivalent to the original loop. The most blatant problem is that we don't know how many more items will be in the list, and the speculative computation of `lp2` and `lp3` may cause run-time errors that would not occur in the original loop. This leads us to define the property of *speculative traversability*. The idea of a speculatively traversable structure is that traversing the empty structure yields the empty structure. Thus, without knowing the length of a list, we can safely traverse the next k items without causing a run-time error. More formally, we define the following important property.

Definition 3.1 *Let t be the type of a* speculatively traversable *pointer data structure with scalar fields s_1, s_2, ..., s_m, and recursive pointer fields r_1, r_2, ..., r_n. If p is the pointer representing the empty structure for type t, then for each r_i the following equality holds: $p\text{->}r_i = p$.*

Note that a *speculatively traversable* recursive data structure is really a different type than an *ordinary* recursive data type because we have changed the meaning of operations on the empty structure. Therefore, to capture this idea we define a new type statement, **specrectype** as illustrated in figure 9.

```
specrectype LIST
{ int key;
  LIST next;
}
```

```
#define LIST_NULL list_nil      /* name of the empty list */
static LIST_NODE list_nil_node; /* node for implementing empty list */
static LIST_PTR list_nil;       /* pointer to the empty list */

void init_LIST_NULL()          /* code to initialize the empty list ....*/
{ list_nil = &list_nil_node;   /*  set pointer to empty list node       */
  list_nil->key = 0;           /*  set each scalar type to approp. zero */
  list_nil->next = list_nil;   /*  set each recursive field to self     */
}

typedef struct LIST         /* node type for speculative list */
 { int key;
   struct LIST *next;
 } LIST_NODE, *LIST_PTR;
```

Figure 9: A *speculatively traversable* **specrectype** and its C implementation

Note that the implementation of the **specrectype** definition is identical to that of the **rectype** except for the definition of the empty data structures. That is, we have implemented the empty structure so that we can speculatively traverse the data structure (extra traversals on the empty structure will always result in the empty structure).

4 Loop Unrolling for Analyzable Pointer Structures

As we discussed in section 2.2, loop-based transformations are important components of compiling for instruction-level parallelism. In this section we present a new loop unrolling technique that applies to structurally inductive and speculatively traversable pointer data structures. In the second part of this section we provide experimental results that indicate substantial performance gain for our loop unrolling technique.

In order to demonstrate a wide variety of loops we consider the six loops presented in figure 10. The first loops are the ones that we extracted from the source code of the GNU cc compiler, while the last three loops we constructed to provide loops with different characteristics. The *sum* loop is typical of a loop that is traversing a structure while accumulating a final value. The *count* loop is an example of a loop that performs some action on a subset of the items in the list. An important characteristic of this loop is that it contains a conditional in the body. The *find* loop is typical of a loop that does not traverse the entire list. The characteristic of importance in this loop is the more complex termination condition.

```
while (lp != LIST_NULL)
  { lp->key = y;
    lp = lp->next;
  }
```

Initialize each element

```
sum = 0;
while (lp != ORIG_NULL)
  { sum += lp->key;
    lp = lp->next;
  }
```

Sum all elements

```
if (lp != LIST_NULL)
  { next_lp = lp->next;
    while (next_lp != LIST_NULL)
      { lp = next_lp;
        next_lp = next_lp->next;
      }
  }
last = lp;
```

Find last item

```
count=0;
while (lp != LIST_NULL)
  { if (lp->key == y) count++;
    lp = lp->next;
  }
```

Count all elements equal to y

```
prev = LIST_NULL;
while (lp != LIST_NULL)
  { next_lp = lp-> next;
    lp->next = prev;
    prev = lp;
    lp = next_lp;
  }
lp = prev;
```

Reverse the list

```
while ((lp != LIST_NULL) &&
       (lp->key != y))
  lp = lp->next;
```

Find first element equal to y

Figure 10: Some different sorts of loops on pointer data structures

4.1 A New Loop Unrolling Technique

Each of the loops given in figure 10 traverses and processes a linked list one item at a time. That is, for each iteration of the while loop, one item of the list is processed. The basic strategy of our loop unrolling is to transform the original loop into a loop which processes more than one item on each iteration. For example, as illustrated in figure 11, we could have three pointers into the list, and process three items of the list on each iteration of the while loop. The advantages of such a transformation include: (1) loop overhead is reduced, (2) the size of the loop body is increased, thereby providing more flexibility for traditional optimizations like dead code removal and instruction scheduling, and (3) in many cases available parallelism is increased because the operations on different list items may proceed independently.

We have isolated two general patterns for performing such while loop transformations. In figure 12 we give the two patterns and the equivalent *3-unrolling* (there is an obvious generalization for *k-unrolling*). All of our example loops, except for *reverse*, fit pattern A. Pattern B is typical of loops, like *reverse*, that are updating the structure of the list as it is traversed. Note that in both cases, the unrolling of the pattern consists of two adjacent while loops. The first loop processes k items for each iteration, and the second loop as processing the remaining tail of the list in the case that the length of the list is not a multiple of k.

The following is a concise summary of the loop unrolling method. An example of applying this method is given in figure 13.

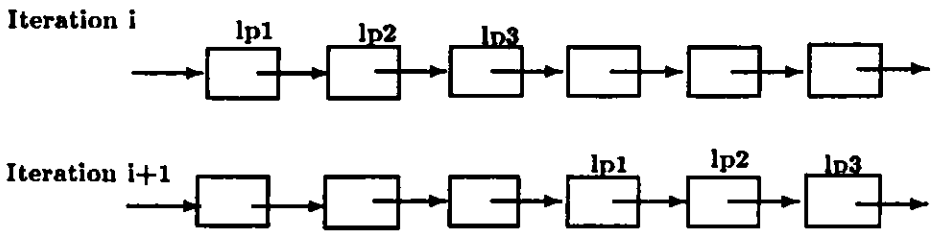


Figure 11: A loop traversal for a 3-unrolling

```
while (cond(lp))
  { body(lp);
    lp = traverse(lp);
  }
```

```
lp2 = traverse(lp);
lp3 = traverse(lp2);
while (cond(lp,lp2,lp3))
  { body(lp);
    body(lp2);
    body(lp3);
    lp = traverse(lp3);
    lp2 = traverse(lp);
    lp3 = traverse(lp2);
  }
while (cond(lp))
  { body(lp);
    lp = traverse(lp);
  }
```

(a) Pattern A and Unrolling

```
while (cond(lp))
  { new_lp = traverse(lp);
    body(lp);
    lp = new_lp;
  }
```

```
lp2 = traverse(lp);
lp3 = traverse(lp2);
while (cond(lp,lp2,lp3))
  { new_lp = traverse(lp3);
    body(lp);
    body(lp2);
    body(lp3);
    lp = new_lp;
    lp2 = traverse(lp);
    lp3 = traverse(lp2);
  }
while (cond(lp))
  { new_lp = traverse(lp);
    body(lp);
    lp = new_lp;
  }
```

(b) Pattern B and Unrolling

Figure 12: Patterns for unrolling 3 times

Detection: Each while loop is examined to see if it fits the pattern A or B. In either case, the loop must obey the following properties:

1. The condition, *cond(lp)* must be some boolean expression defined on the variable *lp* and this variable must be of **specrectype**.
2. The body of the loop must be divisible into two ***non-interfering*** sub-pieces, *body(lp)* and *traverse(lp)*. This analysis can be done by simple symbolic inspection for some loops, or it can require *path matrix* analysis for more complex loops like *reverse*. The *traverse(lp)* computation must be a traversal of a recursive field of the **specrectype**.

Unrolling: The loop is unrolled by introducing new variables *lp2, lp3, ..., lpk* of the appropriate **specrectype** and producing a pair of new loops as illustrated in figure 12. The *k*th copy of the body is created by replacing each occurrence of *lp* with *lpk*.

Conditional optimization: The naive loop unrolling creates the conditional "*cond(lp) && cond(lp2) && ... && cond(lpk)*". This can often be greatly simplified. For example, using property of speculatively traversable structures defined in section 3.1, we can simplify "*(lp != LIST_NULL) && (lp2 != LIST_NULL) && (lp3 != LIST_NULL)*" to "*(lp3 != LIST_NULL)*".

Loop body optimization: We note that in the unrolled loops there are many adjacent copies of the body of the original loop. This often provides further opportunities for traditional optimizations such as *copy elimination* and *dead code removal*. Figure 13(c) illustrates the use of *dead code removal* for the loop *last*.

Parallelization: As a final step we determine if the different copies of the body are independent. In determining this we can make use of the fact that a particular linked structure is inductive. For example, in the case of inductive structures, we can guarantee that the *k* different variables *lp, lp2, ..., lpk* refer to independent nodes.

```
next_lp = lp->next;
while (next_lp != LIST_NULL)
  { lp = next_lp;
    next_lp = next_lp->next;
  }
```

(a) *Original Loop*

```
next_lp = lp->next;
next_lp2 = next_lp->next;
next_lp3 = next_lp2->next;
while (
   (next_lp != LIST_NULL) &&
   (next_lp2 != LIST_NULL) &&
   (next_lp3 != LIST_NULL))
  { lp = next_lp;
    lp = next_lp2;
    lp = next_lp3;
    next_lp = next_lp3->next;
    next_lp2 = next_lp->next;
    next_lp3 = next_lp2->next;
  }
while (next_lp != LIST_NULL)
  { lp = next_lp;
    next_lp = next_lp->next;
  }
```

(b) *Unrolled*

```
next_lp = lp->next;
next_lp2 = next_lp->next;
next_lp3 = next_lp2->next;
while (next_lp3 != LIST_NULL)
  { lp = next_lp3;
    next_lp = next_lp3->next;
    next_lp2 = next_lp->next;
    next_lp3 = next_lp2->next;
  }
while (next_lp != LIST_NULL)
  { lp = next_lp;
    next_lp = next_lp->next;
  }
```

(c) *Unrolled and optimized*

Figure 13: Unrolling *last*

4.2 Experimental Results

In this section we present some experimental results obtained by applying our loop unrolling techniques to the six loops presented in figure 10.

First let us consider the example of unrolling the loop *last*. In figure 13 we show the original loop, the unrolled loop, and the unrolled loop after both the conditional and body have been optimized. Note that these optimizations

are straight-forward applications of the *speculatively traversable* property and *dead code elimination*. Due to the resulting reduced loop overhead and the removal of unneeded computation in the unrolled version of *last*, we would expect improved performance of the unrolled version over the original loop. This performance improvement is confirmed with the experimental figures in table 1. This experiment was performed using the original loop, and optimized unrolled loops for k equal to 2, 3, 4, 5, and 10. In each case, the transformation was performed at the C source level, and then the resulting unrolled loop was compiled using the native cc compiler with the -O optimizer option. For each unrolling we give the time in microseconds required to execute the loop for lists of length 10, 100, and 1000. In addition, the speedup is indicated in parentheses. We note excellent speedups ranging from 1.12 to 2.21 for all cases except for the 10-unrolling run on a list with 10 elements. In this case, the cost of the speculative traversal outweighs the other benefits. Also, for both architectures, we note that a 3-unrolling results in very good performance for all lengths of lists (speedups ranging from 1.21 to 1.88).

SPARC[7]

N	*Original*	*Unroll (2)*	*Unroll(3)*	*Unroll(4)*	*Unroll(10)*
10	1.72	1.50(**1.15**)	1.42(**1.21**)	1.54(**1.12**)	2.46(**0.70**)
100	15.40	11.60(**1.33**)	10.40(**1.48**)	10.00(**1.54**)	9.60(**1.60**)
1000	152.00	102.00(**1.49**)	102.00(**1.49**)	104.00(**1.46**)	88.00(**1.73**)

MIPS[8]

N	*Original*	*Unroll (2)*	*Unroll(3)*	*Unroll(4)*	*Unroll(10)*
10	2.14	1.66(**1.29**)	1.52(**1.41**)	1.67(**1.28**)	2.95(**0.73**)
100	20.23	12.42(**1.63**)	11.17(**1.81**)	10.86(**1.86**)	10.86(**1.86**)
1000	202.33	121.87(**1.66**)	107.81(**1.88**)	102.34(**1.98**)	91.40(**2.21**)

Table 1: Timings for *last*

We have also performed a complete set of experiments on each of the six loops presented in this paper. For each loop we experimented with the effect of the technique with a variety of C compilers, and for each C compiler the effect when used with or without the -O optimizing option [Hen91]. Table 2 summarizes part of those results for the case of 3-unrolling and using the native cc compiler with the -O option. This resulted in impressive speedups for all cases on the MIPS architecture (average speedups of 1.19 to 1.49), and good speedups on most cases for the SPARC architecture (average speedups of 1.03 to 1.17). By studying the code produced by the various cc compilers, we note the following reasons for this speedup: (1) reduced number of branches, (2) reduced number of instructions, and (3) better instruction scheduling due to the increased size of the block body.

	SPARC			MIPS		
	10	100	1000	10	100	1000
last	1.42(**1.21**)	10.40(**1.48**)	102.00(**1.49**)	1.52(**1.41**)	11.17(**1.81**)	107.81(**1.88**)
initialize	2.14(**1.00**)	21.00(**1.00**)	242.00(**0.96**)	1.70(**1.26**)	13.83(**1.47**)	134.37(**1.51**)
reverse	2.40(**1.13**)	22.60(**1.06**)	212.00(**1.16**)	1.77(**1.26**)	13.98(**1.45**)	136.71(**1.49**)
count	2.80(**1.06**)	23.20(**1.17**)	224.00(**1.21**)	2.95(**1.14**)	27.50(**1.19**)	274.20(**1.20**)
sum	2.16(**0.99**)	17.20(**1.08**)	174.00(**1.03**)	1.70(**1.26**)	13.75(**1.48**)	135.15(**1.51**)
find	1.78(**0.78**)	12.60(**1.21**)	122.00(**1.15**)	1.91(**0.82**)	14.14(**1.15**)	135.15(**1.20**)
average	(**1.03**)	(**1.17**)	(**1.17**)	(**1.19**)	(**1.42**)	(**1.46**)

Table 2: Speedups for six benchmarks with an unrolling of 3

4.3 Using Path Matrix Analysis to Enhance Parallelism

The observation that our loop unrolling technique lead to better instruction scheduling on RISC machines shows that techniques like this are very important for the effective exploitation of parallelism available in today's ad-

[7]SPARCstation 2, Sun release 4.1 cc compiler
[8]DECSTATION 5000, MIPS cc compiler

vanced architectures. Furthermore, using more refined alias analysis based on path matrix method, more precise dependence information can be produced to guide instruction scheduling. This is particularly important when one copy of the loop body may not have enough parallelism to match what can be supported by the target architecture. As the future generation of high-performance architectures will have substantially more parallelism at all levels, and the benefit of the proposed method will become even more significant.

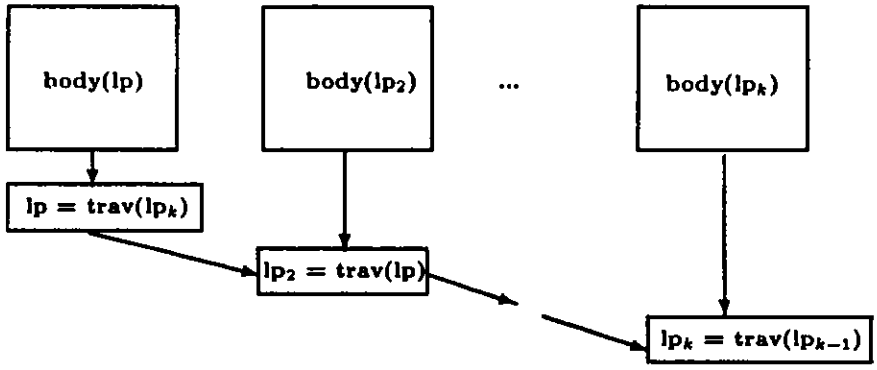


Figure 14: Dependencies in a parallel k-unrolled loop

Consider an unrolled loop of the form given in figure 12(a). If our path matrix analysis determines that each of the variables *lp*, *lp1*, ..., *lpk* refer to distinct nodes, and the body of the original loop only refers to the node plus some scalar variables, then we can build a dependency graph with the basic structure as shown in figure 14. Note that each of the statement sequences *body(lp1)*, ..., *body(lpk)* are totally independent, and the only dependencies are due to the traverse statements. Ample parallelism between iterations may be exposed which can be effectively exploited by the target architectures.

4.4 Handling Coarse-Grain Parallelism

Although we have concentrated on instruction-level parallelism, our techniques are also useful for more coarse-grain parallelism [Hen90, HN90]. For example, consider a while loop in which each iteration performs a relatively large task. In this case, we can use our analysis to determine when it is safe to speculatively traverse the structure while allocating each iteration, or group of iterations, to different parallel processes.

In addition, we can use the inductive property and associated alias analysis tools to determine when two or more recursive procedure calls may execute in parallel. For example, consider a program which operates on a tree by processing the root node, and then recursively processing the sub-trees. If our analysis can determine that the structure is in fact inductive, that is determine that the sub-trees are non-interfering, then we can allocate each recursive call to a parallel process.

However, even when we can detect when it is safe to execute different iterations or procedure calls in parallel, the effective exploitation of coarse-grain parallelism is complicated by the following problems.

Locality: Many parallel architectures have a multi-level memory hierarchy in which memory accesses to local data are considerably faster than memory accesses to non-local data. In order to exploit such architectures it is beneficial to map the data structures in such a way as to minimize non-local references. With array data structures this can often be achieved by mapping the arrays by rows, columns, blocks, or other regular mappings. However, with pointer data structures, the shape of the data structure changes dynamically, and such regular mappings are difficult to perform statically. Even if a mapping is made dynamically, small pointer updates in the structure (for example, reversing two sub-trees) may invalidate the mapping.

Size: In order to determine the grain-size of a parallel process, it is often useful to know the size or length of the data structures. For example, with arrays or vectors, one can often divide one process into approximately

equal sized sub-processes that work on equal sized pieces of the vector or array. Once again, this is difficult to do for pointer data structures. For example, if a tree data structure is not balanced, then the size of the sub-trees may vary considerably, and we cannot use the sub-division of the data structure to guide the subdivision of the process into equal sized processes.

In order to make some progress on the problems due to locality and size, we plan to pursue our approach of extending the programming language to allow the programmer to specify more properties about the data structure. For example, the programmer may actually have some encoding in the data structure that gives the length or size of the data structure. We would like to make this encoding explicit so that the compiler can make use of this information.

5 Conclusions

Current and future generation of high-performance architectures support instruction-level parallelism. To produce efficient code for such machines, a compiler must be able to analyze programs and detect opportunities for optimization and parallelization. This motivates the central theme of this paper: analyzability is an important principle for programming language design and implementation.

In this paper we focused on the analyzability issues for real-life non-scientific programs. Many such programs are being written in imperative languages like C for many different hardware platforms, and there is no sign that this trend will slow down soon. Therefore, we feel that one challenge for researchers in the areas of programming language design and implementation is to work towards solutions that: (1) are easily assimilated and adapted by the community programming in C-like languages, and (2) will lead to effective use of current and future high-performance architectures.

We illustrated the negative effect of poor alias analysis for pointer data structures, and the difficulties encountered in designing loop transformations for them. These difficulties arise because unlike data structures such as scalars and arrays, pointer data structures have dynamically changing shapes and sizes. In addition, programmers use dynamic data structures to implement a wide variety of structures including singly-linked lists, doubly-linked lists, circular lists, nested lists, binary trees, threaded trees, and graphs.

We have proposed a programming language mechanism which can be utilized to design analyable pointer data structures with two important properties : *structural inductivity* and *speculative traversability*. We illustrate how can use *path matrix analysis* to provide accurate alias analysis for structural inductive data structures. We have also described a novel while loop transformation method to aggressively exploit fine-grain parallelism for pointer data structures which are speculatively traversable and we have provided experimental results that show that the transformation results in significant performance improvement. In addition, we discussed how our approach can be used to exploit more coarse-grain parallelism, and we outlined the outstanding problems in this area.

Based on the results of the research presented in this paper, we have decided to implement our approach in McCAT – the McGill Compiler-Architecture Testbed currently being developed by our research group at McGill [HGMS91]. McCAT provides a unified approach to the development and performance analysis of compilation techniques and high-performance architectural features. This will allow us to performance experiments on a wide range of benchmark programs and architectural models and to further refine our method.

References

[ABC86] Todd Allen, Michael Burke, and Ron Cytron. A practical and powerful algorithm for subscript dependence testing. Technical report, IBM, 1986. Internal Report.

[ACK87] Randy Allen, David Callahan, and Ken Kennedy. Automatic decomposition of scientific programs for parallel execution. *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 63–76, January 1987.

[ANP89] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM TOPLAS*, 11(4):598–632, October 1989.

[Ban76] Utpal Banerjee. Data dependence in ordinary programs. Master's thesis, University of Illinois at Urbana-Champaign, 1976.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, MA, 1988.

[DH79] J.J. Dongarra and A.R. Hinds. Unrolling loops in fortran. *Software-Practice and Experience*, 9:219–226, 1979.

[Gua88] Vincent A. Guarna Jr. A technique for analyzing pointer and structure references in parallel restructuring compilers. In *Proceedings of the International Conference on Parallel Processing*, volume 2, pages 212–220, 1988.

[GYDM90] G. R. Gao, R. Yates, J. B. Dennis, and L. Mullin. An efficient monolithic array constructor. In *Proceedings of the 3rd Workshop on Languages and Compilers for Parallel Computing, Irvine, CA*, 1990. To be published by MIT Press.

[Hen90] Laurie J. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell University, January 1990.

[Hen91] Laurie J. Hendren. Experiments on while loop unrolling for pointer data structures. Technical Report ACAPS Note 29 (in preparation), McGill University, 1991.

[HGMS91] Laurie Hendren, Guang Gao, Chandrika Mukerji, and Bhama Sridharan. Introducing McCAT - The McGill Compiler-Architecture Testbed. Technical Report ACAPS Memo 27 (in preparation), McGill University, 1991.

[HN90] Laurie J. Hendren and Alex Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1), 1990.

[HWe90] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, April 1990.

[KKP+80] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Analysis and transformation of programs for parallel computation. In *Proceedings of the Fourth International Computer Software and Application Conference*, October 1980.

[Kri90] S. M. Krishnamurthy. A brief survey of papers on scheduling for pipelined processors. *SIGPLAN Notices*, 25(7):97–106, 1990.

[LH88] James R. Larus and Paul N. Hilfinger. Restructuring Lisp programs for concurrent execution. In *Proceedings of the ACM/SIGPLAN PPEALS 1988 - Parallel Programming: Experience with Applications, Languages and Systems*, pages 100–110, July 1988.

[Muk91] Chandrika Mukerji. Instruction scheduling at the RTL level. Technical Report ACAPS Note 28, McGill University, 1991.

[PW86] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[Sri91] Bhama Sridharan. Creation and transformations of the abstract syntax tree. Technical Report ACAPS Note 27, McGill University, 1991.

[Wol89] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, London and MIT Press, Cambridge, MA, 1989. In the series, Research Monographs in Parallel and Distributed Computing. Revised version of the author's Ph.D. dissertation, Published as Technical Report UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign, 1982.

[WS87] Shlomo Weiss and James E. Smith. A study of scalar compilation techniques for pipelined supercomputers. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 105–109, 1987.

[X3J90] The FORTRAN Technical Committee of ANSI X3J3, June 1990. FORTRAN 90, Draft of the International Standard.

Compile-Time Parallelization of Prolog

Hakan Millroth, Uppsala University

Previous attempts at developing parallel Prolog systems have focused on exploiting AND-parallelism, or OR-parallelism, or both. In this work we parallelize Prolog by exploiting parallelism in its fundamental control structure: recursion.

We have developed a compilation technique [2,3] in which recursion over recursive data structures is compiled to bounded iteration (for-loops) over vectors (the vectors are somewhat analogous to binding environments). The technique is based on an analysis of the unification patterns of recursive programs in Reform [5] computations. (Reform is a new inference systems for logic programming which handles recursion differently from SLD-resolution.)

This compilation technique gives a more efficient parallelization of recursive programs than is possible with a system based on SLD-resolution. In a parallel system based on SLD-resolution it takes O(N) time to spawn all N recursive calls to the program. In our model, the time complexity for getting all recursion levels into work is bounded only by the time it takes to distribute the input data of the program. This may take O(log N) time (on, for example, a machine with tree topology) or O(1) time (in case the input data is already distributed).

We discuss the application of this technique to parallelization of Prolog [4,6]. The basic idea is that parallelization takes places only across recursion levels: the recursion levels of a program, including the head unifications at each level, are computed in parallel. The sequential left-to-right depth-first backtracking scheme of Prolog is retained within recursion levels. (If needed, communicating processes are implemented by sequential co-routining using the delay primitives of Prolog.)

This approach has some appealing consequences:

1. It gives the parallel program a natural and easy-to-understand parallel reading. The programmer can write efficient parallel programs by obeying some simple rules of programming. The programmer is relieved, to a large extent, from explicitly constraining unproductive concurrency.

2. It gives a natural partitioning of the computation and its data, since nondeterminism and producer-consumer relationships are often local to the individual recursion levels. Since we execute the individual levels sequentially, the amount of nondeterminism and data dependencies within recursion levels is completely insignificant for the efficiency of parallelization.

3. There is a simple mapping of the program onto a parallel machine whose processors are organized in a ring: adjacent recursion levels are mapped to adjacent processors. The inter-processor communication on such a machine will mostly

be between neighboring processors, since it is unusual that data is passed between nonadjacent recursion levels in a logic program. We thus achieve predominantly local communication, which is crucial on a distributed memory machine.

4. The workload will automatically be spread evenly among the parallel processors, assuming that each recursion level of the program contains approximately the same amount of work. This assumption seems reasonable for many Prolog programs. Consequently there is not much need for dynamic load balancing.

Hence we have an efficient method for parallelization of well-structured Prolog programs. What to do with unstructured problems? One approach is to parallelize a well-structured metaprogram that controls the computation; cf. Foster & Taylor's scheduler-worker method [1].

References

1. I. Foster & S. Taylor. Strand: New Concepts in Parallel Programming, Prentice-Hall.
2. H. Millroth. Reforming Compilation of Logic Programs, Ph.D. Thesis, Uppsala University, 1990. (Summary to appear at Int. Logic Programming Symp., San Diego, CA., October 1991)
3. H. Millroth. Compiling Reform, (to appear in) Massively Parallel Reasoning Systems (eds. J. A. Robinson & E. E. Siebert), MIT Press, 1991.
4. H. Millroth. Efficient Parallelization of Prolog, (to appear in) Parallelization of Inference Systems (eds. B. Fronhofer & G. Wrightson), Springer-Verlag, 1991.
5. S-A Tarnlund. Reform, (to appear in) Massively Parallel Reasoning Systems (eds. J. A. Robinson & E. E. Siebert), MIT Press, 1991.
6. S-A Tarnlund, H. Millroth, J. Bevemyr, T. Lindgren & M. Veanes. The Reform Machine, in preparation.

Compiling Crystal for Massively Parallel Machines

Extended Abstract

Marina Chen Young-il Choo

Department of Computer Science
Yale University
New Haven, CT 06520
chen-marina@yale.edu choo@yale.edu

7 October 1991

1 Introduction

The Crystal project has focused on making the task of programming massively parallel machines practical while not sacrificing the efficiency of the target code. Our compilation strategy is to start from a high-level problem specification, apply a sequence of optimizations tuned for particular parallel machines, and finally generate code with explicit communication or synchronization. The computation and data are distributed based on the analysis of communication patterns in the program and the cost of communication primitives of the target machine.

Taking advantage of the fact that many algorithms exhibit natural parallelism when formulated mathematically, our approach to parallel programming is to use a purely functional language that resembles mathematical notation. The language has higher-order operators and data structures, thereby making the extraction of parallelism far simpler and allowing us to focus on the global communication issues for massively parallel machines. The simpler semantics of the language allows us to formulate a rigorous theory of program optimization that is indispensable both in the automatic analysis of communication patterns and the explicit specification of user-defined mapping strategies.

1.1 The Principle Features of Crystal

In order to develop a theory of the language that is useful in practice, the language must have clean semantics and orderly algebraic properties. The model of communication must truly reflect the physical characteristics. It must be abstract enough to be conceptualized by the programmer and simple enough to be incorporated into a compiler.

To this end, we have defined the Crystal functional language with special data types that embody locality and structural information in both the problem and the physical domains. The novel data types are *index domains*, which embody the geometric shape of data structures such as multidimensional arrays, trees, and hypercubes, and *data fields*, which generalize the notion of distributed data structures, unifying the conventional notions of arrays and functions.

Since index domains embody the shape of a data field, the geometry of the set of indices indicates the distribution of the data elements. Therefore, a mapping from one index domain to another, called an *index domain morphism*, can be used to represent the change of shape of the distributed data fields. This is at the heart of the global optimization of Crystal programs. Once a suitable morphism has been chosen, a systematic transformation of the program results in a new program with improved behavior.

The language allows different levels of abstraction: atomic functions for sequential computation on a single processor; data fields for data parallel computation; and higher-order functions for combining data parallel components. Of course, the higher-order functions can be considered atomic at the next level of the hierarchy. But such power of abstraction is obtained at the expense of target code performance—in particular, for the SIMD type of machines. Reformulating a high-order Crystal program using data fields (first order functions defined over index domains) is analogous to turning general recursive definitions into tail recursive ones. You gain efficiency at the expense of abstraction and elegance. The first order version is conceptually more complex because the parallel structure of an algorithm must be directly exposed. (For example, a FFT network in its entirety must be defined as an index domain.) The high-order version uses recursion to do the trick (for example, only the basic butterfly pattern is defined, and the FFT network is generated by way of recursion).

1.2 Programming Methodology

The process of designing parallel programs has both formal and informal aspects. The formal aspect, such as program transformation, is mechanizable. For example, once an index domain morphism is specified, the derivation of the new data field from the original definition is automatable. Note that there are no restrictions on the shape of the domain nor on the domain morphism itself, as long as it has an inverse.

The informal aspect requires insight into the behavior of the algorithm, sometimes even a lemma or two. Except for very restricted classes of problems, determining the right domain morphism requires insight.

The role of the compiler is to automate the process of finding appropriate morphisms. One finds classes of programs that are broad enough to encompass interesting and critical application areas, yet restrictive enough to allow the compiler to give a reasonable solution within a reasonable amount of time and resource.

In the general case where the compiler is limited in its capability (since it cannot prove general theorems automatically), the next best thing is to provide a language and programming environment in which the insight of the programmer can be expressed and implemented. For example, the specification of domain morphisms allows new data fields to be automatically derived. The Crystal metalanguage [9] provides such capabilities.

1.3 The Crystal Approach to Compilation

The Crystal functional language radically simplifies the data dependency analysis necessary for synthesizing parallel control structures. The interpretation of the index domains and data fields admits efficient storage management which, in conventional function language implementations, is difficult to do.

The novel compilation techniques include synthesis of parallel control structures, automatic layout and distribution of data, generation of explicit communication from program reference patterns, and global optimization between parallel program modules. Unless the programmer explicitly provides such information, these techniques are necessary for any compiler targeting distributed memory architectures. For problems that are dynamic in nature, the redistribution of data and tasks is handled by the runtime system, utilizing both the static analysis obtained at compile-time and the dynamic dependency and profile of computation gathered during the execution.

In dealing with the two related issues of minimizing communication overhead and determining data layout and load distribution, the compiler first determines the relative location of the data

structures in a virtual domain and then aggregates contiguous parts of the data structures to be mapped into a single processor so as to convert as many references in the source program as possible into local memory access. For the remaining references requiring interprocessor communication, the compiler tries to match the reference patterns with a library of aggregate communication operators and chooses the ones which minimize network congestion and overhead.

2 The Crystal Language

Briefly, the language provides various data types and operations over them, a set of constructors for defining new data types, functional abstraction for creating functions, and function application. A program is a set of possibly mutually recursive definitions. The syntax has been kept simple, with most language constructs expressed in prefix notation, except for simple arithmetic functions, which are infix, and the list and set comprehension that uses the standard set comprehension notation with keywords added.

2.1 Crystal Programs

A Crystal program consists of a set of mutually recursive *definitions* and *directives*. In the interactive version, an expression is evaluated in the standard environment augmented by the definitions. In the compiled version, input is indicated by calling the special function StdIn and output is done by defining the special function StdOut.

A *definition* has the form $a : T = e$ and binds a to the value of e evaluated in the current environment augmented simultaneously with all the other definitions. The T is the type information used by the compiler in the allocation of resources.

Expressions are inductively built up from the constants and the identifiers by function application, both prefix and infix, the primitive data structure forms, the set and list comprehension, and the conditional expression. For any type expression T, $e{:}T$ indicates that the value of e is of type T.

Given any expression $e[x]$, which may or may not contain the variable x, $\mathsf{fn}(x){:}T\{e[x]\}$ denotes a function whose value at v of type T is the value of e evaluated in the current environment with x bound to v.

An expression may also be provided with a local environment:

$$e \ \mathsf{where}\{\ d_1 \ldots d_n\ \}$$

indicates that the expression e is to be evaluated in the current environment augmented with definitions d_i.

The general form for the conditional expression is

$$\left\{\begin{array}{c} b_1 \rightarrow e_1 \\ \vdots \\ b_n \rightarrow e_n \end{array}\right\}$$

where the b_i's are Boolean expressions, known as guards, and the e_i's are expressions of the same type. The value of the conditional expression is e_k if one b_k is true and otherwise undefined. A special symbol, denoted else, represents the conjunction of the negation of the other guards.

When more than one guard is true, an arbitrary choice leads to nondeterminism, which will not be addressed here.

Let $f : T \times T \to T$ be an associative function over some data type T and let $l = \mathsf{list}\{l_1, \ldots, l_n\}$ be a list with elements from T. The operator **reduce** is defined by

$$\mathsf{reduce}(f, l) = f(l_1, f(l_2, \ldots f(l_{n-1}, l_n) \ldots))$$

The operator **scan** is defined by

$$\mathsf{scan}(f, l) = \mathsf{list}\{m_1, \ldots, m_n\}$$

where $m_1 = l_1$, and $m_{i+1} = f(m_i, l_i)$. Note that **reduce** can also be defined over sets, but **scan** cannot since it assumes an ordering of the elements.

2.2 Index Domains

An *index domain* D consists of a set of elements (called indices), a set of functions from D to D (called communication operators), a set of predicates, and the communication cost associated with each communication operator.

In essence, an index domain is a data type with communication cost associated with each function or operator. The reason for making the distinction is that index domains will usually be finite and they are used in defining distributed data structures (as functions over some index domain), rather than their elements being used as values. For example, rectangular arrays can be considered to be functions over an index domain consisting of a set of ordered pairs on a rectangular grid. Also, the elements of an index domain can be interpreted as locations in a logical or real space and time over which the parallel computation is defined. So we classify index domains into certain *kinds* (second order types) according to how they are to be interpreted (for example, as time or space coordinates).

The time coordinates can be detected by the compiler and implemented as a loop whose body may contain assignments to array elements if such a side effect can be done safely. A *time domain* can be semi-infinite and depends on the function that is being defined over it. For each execution, those domain elements that are actually generated during computation are controlled by the use of the minimalization operator. Minimalization over a semi-infinite domain corresponds to unbounded minimalization, which simulates while-loops.

Basic Index Domains

Examples of basic classes of index domains include the interval, the hypercube, and tree domains. In this paper we only nee the interval index domain.

An *interval domain*, denoted $\mathsf{interval}(m, n)$, where m and n are integers and $m \leq n$, is an index domain whose elements are the set of integers $\{m, m+1, m+2, \ldots, n\}$ with the usual integer functions and predicates. The communication operators are $\mathsf{prev} : i \mapsto i - 1$ and $\mathsf{next} : i \mapsto i + 1$, with communication cost 1. The operators **lb** and **ub** return the lower bound (m) and the upper bound (n) of the interval domain respectively. When $m \geq n$, we define the index domain to be the same except that **prev** and **next** have reversed meaning.

Index Domain Constructors

Given index domains D and E, we can construct their product $(D \times E)$, disjoint union (coproduct) $(D + E)$, and function space $D \to E$, in the usual way.

Since index domains are first class objects, it is possible to define a new index domain as the value of a recursively defined function. For example, let B be some index domain, and $\phi[A(f(x))]$ be some index domain expression. Then

$$A = \mathsf{fn}(x) : T \begin{cases} b_0(x) - B \\ b_1(x) - \phi[A(f(x))] \end{cases}$$

defines an index domain for each x in T, with suitable guards b_0 and b_1. In this way, quite complex, data-dependent index domains can be constructed.

Index Domain Morphisms

Index domain morphisms formalize the notion of transforming one index domain into another. In this paper we define a special class of known as "reshape morphisms" simply to be a bijection between two index domains.

Here are examples of some useful reshape morphisms:

An *affine morphism* is a reshape morphism that is an affine function from one product of intervals to another. Affine morphisms unify all types of loop transformations (interchange, permutation, skewing) [2, 3, 6, 22, 23], and those for deriving systolic algorithms [11, 19, 20, 8]. For example, if $D_1 = \mathsf{interval}(0,3)$ and $D_2 = \mathsf{interval}(0,6)$, then $g = \mathsf{fn}(i,j) : D_1 \times D_2\{(j,i) : D_2 \times D_1\}$ is an affine morphism that effectively performs a loop interchange.

Another example illustrates a slightly more interesting codomain E of the morphism g by taking the image of a function g'.

$$\begin{aligned} D_0 &= \mathsf{interval}(0,3) \\ D &= D_0 \times D_0 \\ E &= \mathsf{image}(D, g') \ \mathsf{where}\{\ g' = \mathsf{fn}(i,j){:}D\{(i-j, i+j)\}\ \} \\ g &= \mathsf{fn}(i,j){:}D\{(i-j, i+j){:}E\} \\ g^{-1} &= \mathsf{fn}(i,j){:}E\{((i+j)/2, (j-i)/2){:}D\} \end{aligned}$$

Whenever it is legal to apply this affine morphism to a 2-level nested loop structure – consistent with the data dependencies in the loop body [1, 2, 4, 5, 7, 22]) – a structure that is similar, but "skewed" from the original, is generated. The most common case is when elements of the inner loop can be executed in parallel but only half of the elements are active in each iteration of the outer loop. In this example, the index domain E has holes, and so guards in the loops must test whether $i + j$ and $i - j$ are even, since only these points correspond to the integral points in D.

There are numerous other forms of reshape morphisms ranging from "piece-wise affine" morphisms for more complex loop transformations [18], to those that are mutually recursive with the program (to be transformed) for dynamic data distribution.

2.3 Data Fields and Data Field Morphisms

Data fields generalize the notion of distributed data structures and recursively definable functions.

Definition. A *data field* is a function over some index domain D into some domain of values V.

Usually, V will be the integers or the floating point numbers; however, for higher-order data fields, it can be some domain of data fields. Data fields unify the notion of distributed data

structures, such as arrays, and functions. A parallel computation is specified by a set of data field definitions, which may be mutually recursive.

To illustrate the use of data fields, consider the following program segment written in some imperative language (assuming there are no other statements assigning values to A):

```
float array A(0..n,0..n);
if i=0 or j=0 then A:=e1;
for i:= 2 to n do {
   for j := 2, n do {
      A(i,j) := A(i-1,j) + A(i,j-1) } }
```

Let V be the data type of floating point numbers. In the notation of data fields, the above is written as

$$D_0\text{:domain} = \text{interval}(0, n)$$
$$D\text{:domain} = \text{prod-dom}(D_0, D_0)$$
$$a\text{:dfield}(D, V) = \text{fn}(i, j)\text{:}D\begin{Bmatrix} i = 0 \vee j = 0 \rightarrow e_1 \\ \text{else} \rightarrow a(i-1, j) + a(i, j-1) \end{Bmatrix}$$

New data fields can be derived using index domain morphisms.

Definition. A *data field morphism* induced by an index domain morphism $g : D \rightarrow E$, is a mapping

$$g^* : (E \rightarrow V) \rightarrow (D \rightarrow V) : a \mapsto a \circ g$$

where $D \rightarrow V$ and $E \rightarrow V$ are sets of data fields.

Given a data field $a : D \rightarrow V$ and a domain morphism $g : D \rightarrow E$, what we generally want is to find the new data field $\hat{a}$ such that $g^*(\hat{a}) = a$. In order to solve this equation we need the inverse of g—that is, g needs to be a reshape morphism. Then given g and g^{-1}, we can formally derive $\hat{a} = g^{-1*}(a) = a \circ g^{-1}$.

3 Program Transformation

Data fields and domain morphisms are semantic entities that are represented in a programming language. Semantically, a new data field can be defined as the composition of a data field and a domain morphism.

We assume an equational theory of the Crystal language with the usual algebraic identities and the inference rules. We can formally transform the original program into a more efficient one.

For simplicity, we begin with a program consisting of one definition:

$$a = \text{fn}(x)\text{:}D\{\tau_1[a]\},$$

where $\tau_1[a]$ is an expression in x possibly containing a. Through an abuse of notation, we also use a to denote the data field defined. Next, let the reshape morphism g and its inverse be given by

$$g = \text{fn}(x)\text{:}D\{\tau_2\text{:}E\} \quad \text{and} \quad g^{-1} = \text{fn}(y)\text{:}E\{\tau_3\text{:}D\}.$$

Semantically, what we want is a data field $\hat{a}$ satisfying $\hat{a} = a \circ g^{-1}$. However, merely executing the program g^{-1} followed by a does not decrease the communication cost. What we want is a new definition of $\hat{a}$ that does not contain either a, g, or g^{-1}. A strategy for obtaining a new definition for $\hat{a}$ from the definitions of a, g, and g^{-1} is the following:

1. Using the identity $a = \hat{a} \circ g$, replace all occurrences of a with $\hat{a} \circ g$ in the definition of a.

2. Using a combination of unfoldings of g and g^{-1} and various other identities given in the theory, eliminate all occurrences of g and g^{-1} from the result of the first step. A very useful transformation turns out to be the η-abstraction, where we provide a function with dummy arguments in order to unfold it.

4 The Crystal Metalanguage

Since the program transformations used above are all mechanizable, we have defined a *metalanguage* in which these transformations can be defined, and which furthermore allows the user to experiment with other transformations [9, 24].

Meta-Crystal borrows ideas from ML [10] and 3-Lisp [21]. It consists of basic constructors and selectors for each of the constructs in Crystal and operations that manipulate programs and a set of operations for manipulating the programs, such as folding and unfolding, substitution, and normalization or beta-reduction. Using *meta-abstraction*, the reshape transformation can be defined in terms of primitive manipulations of the constructs.

5 The Crystal Compiler

The Crystal compiler consists of three major stages: the front-end, the middle-analysis, and the back-end as shown in Figure 1. The front-end builds the abstract syntax tree and other necessary data structures for an input Crystal program.

The middle-analysis consists of the traditional semantic and dependence analyses, the more novel reference pattern and domain analyses, and other source level transformations. At the heart of the compiler is a module for generating explicit communication from shared-memory program references [12, 14, 15]. Index domain alignment[13, 16] considers the problem of optimizing space allocation for arrays based on the cross-reference patterns between arrays.

The availability of massively parallel machines opens up opportunity for programs with large scale parallelism to gain tremendous performance over those that do not. We have recently obtained new results in program dependency analysis (more accurate dependency test in the presence of conditional statements [17]) and developed new loop transformation techniques [18], both resulting in more parallelism than existing techniques.

The back-end contains the code generators and the run-time systems. The code generation is done in two separate steps. In the first step the Crystal code generator produces procedure calls to the communication routine supported by the run-time system together with *Lisp code for the CM/2 or C code for the hypercube multiprocessors. The sequential code with the run-time library routines form the interface between the parallelzing compiler and the single processor (e.g., superscalar architecture) compiler, as shown in Figure 1.

In the second, code generation is done by invoking a vendor-supported compiler to compile *Lisp or C into lower-level or machine code. In the case of the Connection Machine 2, the *Lisp compiler generates PARIS instructions, which we found can be further optimized by a simple *expression compiler* that provides significant performance improvements as shown in the next section. Since then, Thinking Machines has developed a new instruction set with the so-called *slice-wise* data representation that allows far better control of the underlying hardware and thus offers much better performance. Unfortunately, TMC is not committed to supporting *Lisp targeted to the slice-wise instructions. Consequently, our current approach of *Lisp-Paris-Expression compiler will

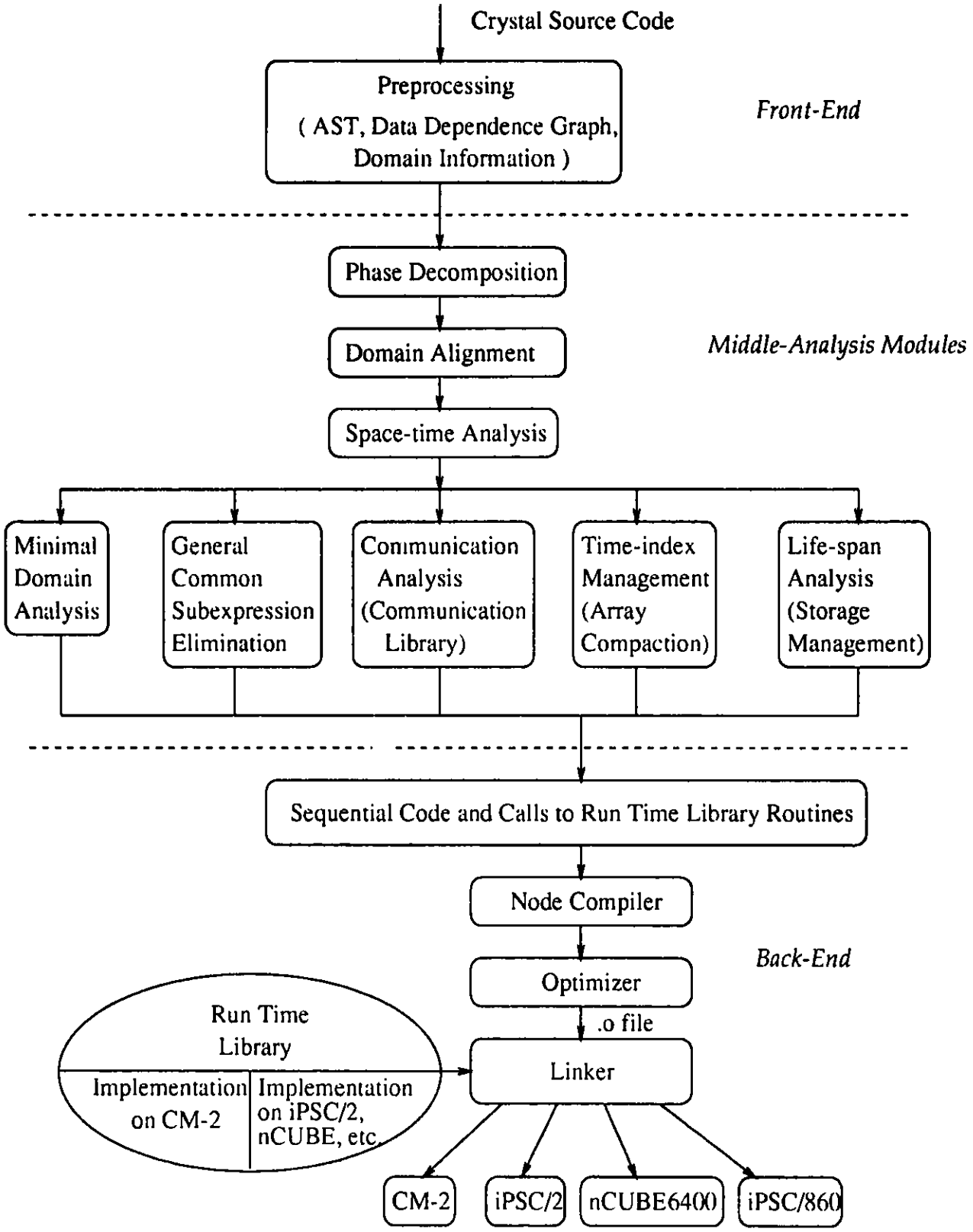


Figure 1: Structure of the Crystal Compiler.

not be used for any future development. A direct path to some kind of intermediate code will be taken instead.

Machine	#Nodes	Peak Mflops/node	Nominal Mflops/node
iPSC/2 (Yale)	64	1.10	0.65
iPSC/860 (ORNL)	128	80.00	6.00 – 10.00
nCUBE 2 (Sandia)	1024	8.00	1.00
CM-2 (Seduto, TMC)	8096 (512 FPU)	7.00	-

Table 1: The performance of parallel machines used.

6 Performance Results

Two benchmark programs, one for weather forecasting and the other for semiconductor simulation, are used to examine the performance of the compiler-generated code on three MIMD hypercube multiprocessors, namely, Intel iPSC/2, iPSC/860, and nCUBE 6400, and the SIMD Connection Machine.

The highest megaflop rates obtained using the compiler generated code on these four machines are given in Table 2 and Table 3. As a reference, we also give the peak and nominal megaflop rates of each of the four machines in Table 1.

6.1 Machine Configuration and System Software Used

The Intel iPSC/2 with the 80386 processor located at Yale has 64 processors, runs Unix V/386 3.2 for host operating system, NX 3.2 for the nodes, and provides Intel 3.2 C compiler.

The iPSC/860 machine (iPSC/2 with i860 node processors) located at Oak Ridge National Labs (ORNL) has 128 processors, but at most 64 processors are used for this experiment. This machine has the C compiler supplied by the Portland Group.

The nCUBE 6400 located at Sandia National Labs has 1K processors although only 64 processors are used for our experiment. It has a Sun 4 as its front-end and provides the NCC 3.11 C compiler.

The timing results for the Connection Machine 2 are measured on an 8K processor CM-2 where each processor has 256K bits of memory and each group of 32 processors share a 64-bit Floating Point Units. CM software version 6.0 and *LISP compiler 6.0 are used. The *Lisp compiler generates PARIS instructions which is in the so-called *field-wise* data format.

An expression compiler developed at Yale generates optimized field-wise CMIS instructions where the memory accesses are greatly reduced. The performance results with and without the expression compiler are presented for the two applications. All results on the Connection Machine are extrapolated to a full CM-2 with 64K processors, i.e., 2K 64-bit floating-point units.

6.2 Shallow Water Equation Solver

Shallow water equation is used to model the global motions of atmospheric flows in weather forecast. The algorithm is iterative, operating on a two-dimensional grids, with local computation at each grid point and data exchanges between neighboring grid points. The Crystal program for this shallow equation solver is given in the appendix.

Table 2 presents the execution time and megaflop rates of this application for different machines. In this experiment, 64 processors are used for each of the three MIMD machines. The problem size

Machine	Total Time	Computation		Communication		Mflops
	Seconds	Seconds	%	Seconds	%	
iPSC/2	85.208	80.392	94.3	6.846	5.7	24.00
iPSC/860	11.805	10.371	87.9	1.552	12.1	173.21
nCUBE 2	31.269	29.479	94.3	2.377	5.7	65.39
CM-2 (*LISP)	67.862	43.530	64.1	24.332	35.9	1840.00
CM-2 (*LISP/CMIS)	47.472	23.152	48.8	24.320	51.2	2630.00

Table 2: The performance of Shallow-water Equation Solver

used is 256K×120 (area × iteration). For the Connection Machine, a much larger problem size, 16M× 120, is used. Due to the difficulty with the node compiler of the i860 chip, we aren't able to get much beyond 2 megaflops per processor. Given the same problem size, the computation on an iPSC/860 processor is 8 times faster than that of an iPSC/2 processor while its communication capability is about 4.5 times faster, thus resulting in a higher percentage of communication overhead for the iPSC/860. Quadrapling the problem size for iPSC/860 increases the total rate to about 180 megaflops, with the percentage of communication overhead slightly higher at 13from short-message send/receive to long-message send/receive.

For this application, CM-2 has a more significant communication overhead, about 60inter-processor communication. One possible explanation for the higher percentage communication overhead for the CM result is the following: Each processor on a hypercube machine computes a sub-grid, and the data exchange involves only the boundary grid points whereas the each CM processor iterates over the virtual processors, forcing the data movement of all grid points either by actual communication between processors or by local copying within processors.

The speedup of execution time over increasing number of processors on these machines are shown in Figures 1(a) and 1(b). The problem size used is 64K×120 for hypercube machines, and 4M×120 for CM-2. As a CM-2 cannot be configured into many different sizes, the 4K processor configuration is used as the basis. Some of the points in Figure 1(b) are extrapolated from the timings on one machine size with different number of virtual processors (vp-ratios). We can do this becasue vp-ratio is the predominant factor for determining the computation time and communication time between adjacent processors.

6.3 Bohr Code

Bohr code is an application using monte-carlo method for semiconductor simulations. It contains many independent trials (nmax is the local array size in each trial) which can be done concurrently on different processors, with a global reduction summarizing the result of all trials.

Table 3 presents the execution time and megaflop rates for this code. Similar to the above, 64 nodes are used for the three MIMD machines for this benchmark. The problem size used is 4K×4 (trials × nmax) for the MIMD machines and 256K× 64 for the CM.

Since the only communication needed for this application is a global reduction at the end of program execution, communication time is almost a negligible portion of total execution time.

Machine	Total Time	Computation		Communication		Mflops
	Seconds	Seconds	%	Seconds	%	
iPSC/2	12.413	12.331	99.3	0.081	0.7	26.40
iPSC/860	3.348	3.320	99.2	0.028	0.8	97.87
nCUBE 2	8.148	8.115	99.6	0.033	0.4	40.22
CM-2 (*LISP)	45.054	43.926	97.5	1.128	2.50	2370.00
CM-2 (*LISP/CMIS)	31.460	30.350	96.5	1.110	3.53	3400.00

Table 3: The performance of Bohr code

The speedups of Bohr code on different machines are shown in Figures 1(c) and 1(d). The problem size used here is 4K×4 for the three MIMD machines, and 16K×10 for CM-2.

For obtaining speedup result on the CM-2, the same problem size must be used for all machine sizes. Due to the heavy use of memory of this applicaiton, the largest problem size can fit on a 4K-processor configuration will run on a 16K-processor machine with one virtual processor per node. Consequently, a larger machine will be useful only with a larger problem. Hence the speedup results are only given for three configurations. Again, we extrapolate the speedup results based on a single physical machine size. According to the timing result, the small percentage of the reduction time indicates the accuracy of extropolation although, in isolation, a gloabal reduction depends on the machine size as well as the problem size.

References

[1] J.R. Allen. *Dependence Analysis for Subscript Variables and Its Application to Program Transformation.* PhD thesis, Rice University, April 1983.

[2] J.R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN'84 Symposium on Compiler Construction*, pages 233–46. ACM, 1984.

[3] J.R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, 10 1987.

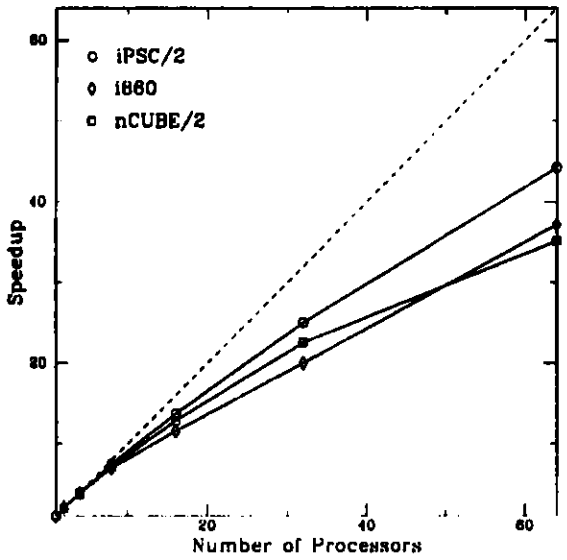
[4] U. Banerjee. Data dependence in ordinary programs. Master's thesis, University of Illinois at Urbana-Champaign, November 1976.

[5] U. Banerjee. *Speedup of Ordinary Programs.* PhD thesis, University of Illinois at Urbana-Champaign, October 1979.

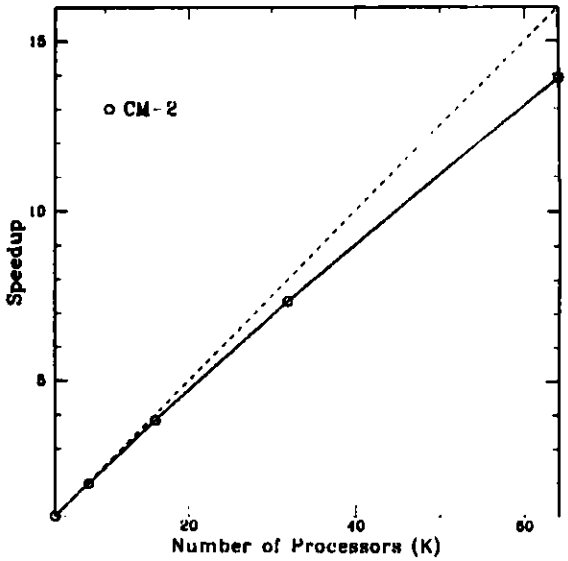
[6] U. Banerjee. A theory of loop permutation. Technical report, Intel Corporation, 1989.

[7] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86*. 1986.

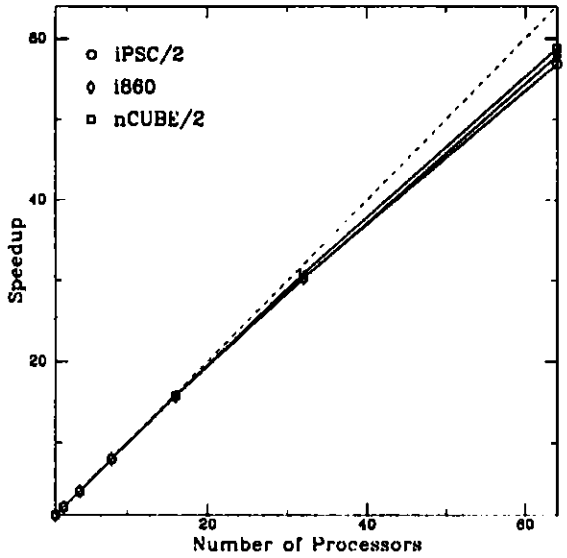
[8] Marina Chen, Young-il Choo, and Jingke Li. Compiling parallel programs by optimizing performance. *Journal of Supercomputing*, 1(2):171–207, July 1988.



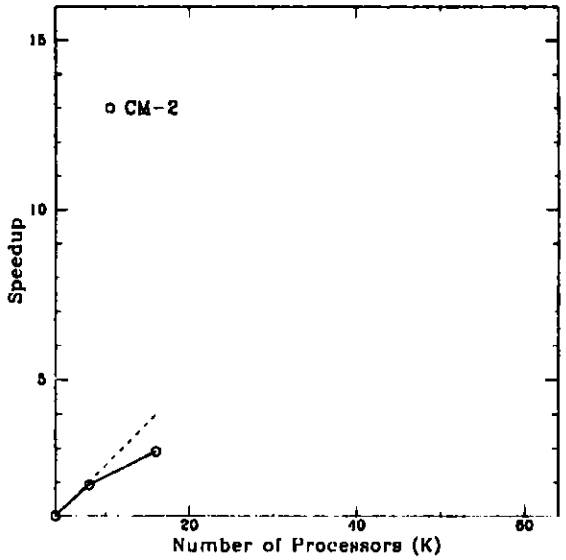
(a) Speedup of Shallow-water Code on Hypercube Machines



(b) Speedup of Shallow-water Code on CM-2



(c) Speedup of Bohr Code on Hypercube Machines



(d) Speedup of Bohr Code on CM-2

Figure 2: Speedups of Shallow-water and Bohr Applications

[9] Young-il Choo and Marina Chen. A theory of parallel-program optimization. Technical Report YALEU/DCS/TR-608, Dept. of Computer Science, Yale University, July 1988.

[10] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*. Springer-Verlag, Berlin, 1979.

[11] R. Karp, R. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–90, July 1967.

[12] Jingke Li and Marina Chen. Generating explicit communications from shared-memory program references. In *Proceedings of Supercomputing'90*, 1990.

[13] Jingke Li and Marina Chen. Index domain alignment: Minimizing cost of cross-reference between distributed arrays. In *The Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*, 1990.

[14] Jingke Li and Marina Chen. *Proceedings of the Workshop on Programming Languages and Compilers for Parallel Computing*, chapter Automating the Coordination of Interprocessor Communication. MIT Press, 1990.

[15] Jingke Li and Marina Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transaction on Parallel and Distributed Systems*, 1991.

[16] Jingke Li and Marina Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 1991.

[17] Lee-chung Lu and Marina Chen. Subdomain dependency test for massively parallelism. In *Proceedings of Supercomputing'90*, 1990.

[18] Lee-chung Lu and Marina Chen. A unified framework for systematic applications of loop transformations. Technical Report 818, Yale University, August 1990.

[19] D.I. Moldovan. On the analysis and synthesis of vlsi algorithms. *IEEE Transactions on Computers*, C-31(11):1121–26. Nov. 1982.

[20] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proceedings of 11th Annual Symposium on Computer Architecture*, pages 208–14, 1984.

[21] Brian Cantwell Smith. Reflection and semantics in lisp. In *Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, Salt Lake City, Utah, January 1984. ACM.

[22] M.J. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Dept. of Computer Science, 1982.

[23] M.J. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, Mass., 1989.

[24] Allan Yang. Design and implementation of meta-crystal: A metalanguage for parallel program optimization. Technical report, Yale University, 1989.

A New Method for Compile-Time Granularity Analysis: An Extended Abstract

X. Zhong, E. Tick, S. Duvvuru,
L. Hansen, A. V. S. Sastry and R. Sundararajan
University of Oregon

Abstract

We present a new granularity analysis scheme for concurrent logic programs. The main idea is that, instead of trying to estimate costs of goals precisely, we provide a compile-time analysis method which can efficiently and precisely estimate *relative* costs of active goals given the cost of a goal at runtime. This is achieved by estimating the cost relationship between an active goal and its subgoals at compile time, based on the call graph of the program. *Iteration parameters* are introduced to handle recursive procedures. Compared with methods in the literature, our scheme has several advantages: it is applicable to any program, it gives a more precise cost estimation than static methods, and it has lighter runtime overheads than absolute estimation methods.

1 Introduction

The importance of grain sizes of tasks in a parallel computation has been well recognized [6, 5, 7]. In practice, the overhead to execute small grain tasks in parallel may well offset the speedup gained. Therefore, it is important to estimate the costs of the execution of tasks so that at runtime, tasks can be scheduled to execute sequentially or in parallel to achieve the maximal speedup.

Granularity analysis can be done at compile time or runtime or even both [7]. The compile-time approach estimates costs by statically analyzing program structure. The program is partitioned statically and the partitioning scheme is independent of runtime parameters. Costs of most tasks, however, are not known until parameters are instantiated at runtime and therefore, the compile-time approach may result in inaccurate estimates. The runtime approach, on the other hand, delays the cost estimation until execution and can therefore make more accurate estimates. However, the overhead to estimate costs is usually too large to achieve efficient speedup, and therefore the approach is usually infeasible. The most promising approach is to try to get as

much cost estimation information as possible at compile time and make the overhead of runtime scheduling very slight. Such approach has been taken by Tick [10], Debray *et al.* [2], and King and Soper [4]. In this paper, we adopt this strategy.

A method for the granularity analysis of concurrent logic programs is proposed. Although the method can be well applied to other languages, such as functional languages, in this paper, we discuss the method only in the context of concurrent logic programs. The key observation behind this method is that task spawning in many concurrent logic program language implementations, such as Flat Guarded Horn Clauses (FGHC) [12], depends only on the *relative* costs of tasks. If the compile-time analysis can provide simple and precise cost relationships between an active goal and its subgoals, then the runtime scheduler can efficiently estimate the costs of the subgoals based on the cost of the active goal. The method achieves this by estimating, at compile time, the cost relationship based on the call graph and the introduction of iteration parameters.

2 Motivations

Compile-time granularity analysis is difficult because most of the information needed, such as size of a data structure and number of loop iterations, are not known until runtime. Sarkar [7] used a profiling method to get the frequency of recursive and nonrecursive function calls for a functional language. His method is simple and does not have runtime overheads, but can give only a rough estimate of the actual granularity.

In the logic programming community, Tick [10] first proposed a method to estimate weights of procedures by analyzing the call graph of a program. The method, as refined by Debray [1], derives the call graph of the program, and then combines procedures which are mutually recursive with each other into a single cluster (i.e., a strongly connected component in the call graph). Thus the call graph is converted into an acyclic graph. Procedures in a cluster are assigned the same weight which is the sum of the weights of the cluster's children (the weights of leaf nodes are one, by definition). This method has very low runtime overhead; however, goal weights are estimated statically and thus cannot capture the dynamic change of weights at runtime. This problem is especially severe for recursive (or mutually recursive) procedures.

As an example of the method, consider the naive-reverse procedure in Figure 1.[1] Examining the call graph, we find that the algorithm assigns a weight of one to `append/3` (it is a leaf), and a weight of two to `nrev/2` (one plus the weight of its child). Such weights are associated with *every* procedure invocation and thus cannot accurately reflect execute time.

Debray *et al.* [2] presented a compile-time method to derive costs of predicates. The cost of a predicate is assumed to depend solely on its input argument sizes. Relationships between input and output argument sizes in predicates are first derived based on so-called data dependency

[1] The clauses in the `nrev/2` program do not have guards, i.e., only head unification is responsible for commit.

```
nrev([],R) :- R=[].
nrev([H|T],R) :- nrev(T,R1), append(R1,[H],R).

append([],L,A) :- A=L.
append([H|T],L,A) :- A=[H|A1], append(T,L,A1).
```

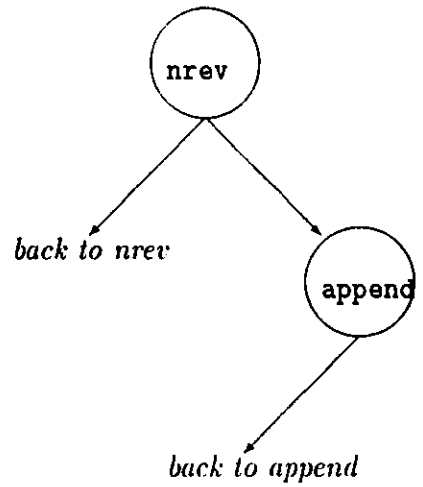


Figure 1: Naive Reverse and its Call Graph

graphs and then recurrence equations of cost functions of predicates are set up. These equations are then solved at compile time to derive closed forms (functions) for the cost of predicates and their input argument sizes, together with the closed forms (functions) between the output and input argument sizes. Such cost and argument size functions can be evaluated at runtime to estimate costs of goals. A similar approach was also proposed by King and Soper [4]. Such approaches represent a trend toward precise estimation. For `nrev/2`, Debray's method gives $\text{Cost}_{nrev}(n) = 0.5n^2 + 1.5n + 1$, where n is the size of the input argument. This function can then be inserted into the runtime scheduler. Whenever `nrev/2` is invoked, the cost function is evaluated, which obviously requires the value n, the size of its first argument. If the cost is bigger than some preselected overhead threshold, the goal is executed in parallel; otherwise, it is executed sequentially.

The method described suffers from several drawbacks (see [13] for further discussion). First, there may be considerable runtime overhead to keep track of argument sizes, which are essential for the cost estimation at runtime. Furthermore, the sizes of the initial input arguments have to be given by users or estimated by the program when the program begins to execute. Second, within the umbrella of argument sizes, different metrics may be used, e.g., list length, term depth, and the value of an integer argument. It is unclear (from [2, 4]) how to correctly choose metrics which are relevant for a given predicate. Third, the resultant recurrence equations for size relationships and cost relationships can be fairly complicated.

It is therefore worth remedying the drawbacks of the above two approaches. It is also

clear that there is a tradeoff between precise estimation and runtime overhead. In fact, Tick's approach and Debray's approach represent two extremes in the granularity estimation spectrum. Our intention here is to design a middle-of-the-spectrum method: fairly accurate estimation, applicable to any procedures, without incurring too much runtime overhead.

3 Overview of the Approach

We argue here, as in our earlier work, that it is sufficient to estimate only *relative* costs of goals. This is especially true for an on-demand runtime scheduler [8]. Therefore, it is important to capture the cost *changes* of a subgoal and a goal, but not necessarily the "absolute" granularity. Obviously the costs of subgoals of a parent goal are always less than the cost of the parent goal, and the sum of costs of the subgoals (plus some constant overhead) is equal to the cost of the parent goal. The challenging problem here is how to distribute the cost of the parent goal to its subgoals properly, especially for a recursive call. For instance, consider the naive reverse procedure `nrev/2` again. Suppose goal `nrev([1,2,3,4],R)` is invoked (i.e., clause two is invoked) and the cost of this query is given, what are the costs of `nrev([2,3,4],R1)` and `append(R1,[1],R)`?

It is clear that the correct cost distribution depends on the runtime state of the program. For example, the percentage of cost distributed to `nrev([1,2,3,4],R)` (i.e., as one of the subgoals of `nrev([1,2,3,4,5],T)` will be different from that of cost distributed to `nrev([1,2],R)`. To capture the runtime state, we introduce an *iteration parameter* to model the runtime state, and we associate an iteration parameter with every active goal. Since the cost of a goal depends solely on its entry runtime state, its cost is a function of its iteration parameter. Several intuitive heuristics are used to capture the relations between the iteration parameter of a parent goal and those of its children goals. To have a simple and efficient algorithm, only the AND/OR call graph of the program, which is slightly different from the standard call graph, is considered to obtain these iteration relationships. Such relations are then used in the derivation of recurrence equations of cost functions of an active goal and its subgoals. The recurrence equations are derived simply based on the above observation, i.e., the cost of an active goal is equal to the summation of the costs of its subgoals.

We then proceed to solve these recurrence equations for cost functions bottom up, first for the leaf nodes of the modified AND/OR call graph, which can be obtained in a similar way in Tick's modified algorithm by clustering those mutually recursive nodes together in the AND/OR call graph of the program (see Section 2). After we obtain all the cost functions, cost distribution functions are derived as follows. Suppose the cost of an active goal is given, we first solve for its iteration parameter based on the cost function derived. Once the iteration parameter is solved, costs of its subgoals, which are functions of their iteration parameters, can be derived based on

the assumption that these iteration parameters have relationships with the iteration parameter of their parent, which are given by the heuristics. This gives the cost distribution functions desired for the subgoals.

To recap, our compile-time granularity analysis procedure consists of the following steps:

1. Form the call graph of the program and cluster mutually recursive nodes of the modified AND/OR call graph.
2. Associate each procedure (node) in the call graph with an iteration parameter and use heuristics to derive the iteration parameter relations.
3. Form recurrence equations for the cost functions of goals and subgoals.
4. Proceed bottom up in the modified AND/OR call graph to derive cost functions.
5. Solve for iteration parameters and then derive cost distribution functions for each predicate.

4 Deriving Cost Relationships

4.1 Cost Functions and Their Recurrence Equations

To derive the cost relationships for a program, we use a graph G (called an AND/OR call graph) to capture the program structure. Formally, G is a triple (N, E, A), where N is a set of procedures denoted as $\{p_1, p_2, \ldots, p_n\}$ and E is a set of pair nodes such that $(p_1, p_2) \in E$ if and only if p_2 appears as one of the subgoals in one of the clauses of p_1. Notice that there might be multiple edges (p_1, p_2) because p_1 might call p_2 in multiple clauses. A is a partition of the multiple-edge set E such that (p_1, p_2) and (p_1, p_3) are in one element of A if and only if p_2 and p_3 are in the body of the same clause whose head is p_1. Intuitively, A denotes what subgoals are AND processes. After applying A to edges leaving out a node, edges are partitioned into clusters which correspond to clauses and these clauses are themselves OR processes. Figure 2 shows an example, where the OR branches are labeled with a bar, and AND branches are unmarked. Leaf facts (terminal clauses) are denoted as empty nodes.

As in [1], we modify G so that we can cluster all those recursive and mutually recursive procedures together and form a directed acyclic graph (DAG). This is achieved by traversing G and finding all strongly-connected components. In this traversing, the difference between AND and OR nodes is immaterial, and we simply discard the partition A. A procedure is recursive if and only if the procedure is in a strongly-connected component. After nodes are clustered in a strongly-connected component in G, we form a DAG G', whose nodes are those strongly-connected components of G and edges are simply the collections of the edges in G. This step can be accomplished by an efficient algorithm proposed by Tarjan [9].

```
qsort([], S) :- S=[].
qsort([M|T],S) :-
    split(T,M,S,L),
    qsort(S,SS),
    qsort(L,LS),
    append(SS,LS,S).

split([],   M,S,L) :- S=[], L=[].
split([H|T],M,S,L) :- H < M  |
    S=[H|TS], split(T,M,TS,L).
split([H|T],M,S,L) :- H >= M |
    L=[H|TL], split(T,M,S,TL).
```

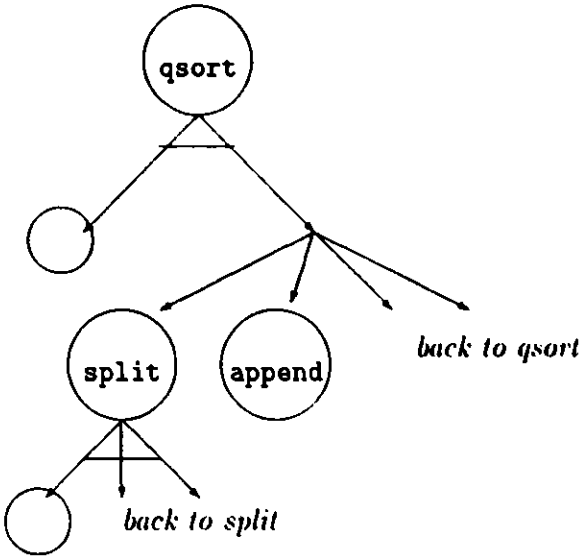


Figure 2: Quick Sort: FGHC Source Code and the AND/OR Call Graph

The cost of an active goal p is determined by two factors: its entry runtime state s during the program execution and the structure of the program. We use an integer n, called the *iteration parameter*, to approximately represent state s. Intuitively, n can be viewed as an encoding of a program runtime state. Formally, let $\mathcal{S}$ be the set of program runtime states, M be a mapping from $\mathcal{S}$ to the set of natural numbers N such that $M(s) = n$ for $s \in \mathcal{S}$. It is easy to see that the cost of p is a function of its iteration parameter n. It is also clear that the iteration parameter of a subgoal of p is a function of n. Hereafter, suppose p_{ij} is the j^{th} subgoal in the i^{th} clause of p. We use $I_{ij}(n)$ to represent the iteration parameter of p_{ij}. The problem of how to determine function I_{ij} will be discussed in Section 4.2.

To model the structure of the program, we use the AND/OR call graph G as an approximation. In other words, we ignore the attributes of the data, such as size and dependencies. We first derive recurrence equations of cost functions between a procedure p and its subgoals by looking at G. Let $\text{Cost}_p(n)$ denote the cost of p. Three cases arise in this derivation:

> **Case 1:** p is a leaf node of G which is non-recursive. This includes cases where that p is a built-in predicate. In this case, we simply assign a constant c as $\text{Cost}_p(n)$. c is the cost to execute p. For instance such cost can be chosen as the number of machine instructions in p.

For the next two cases, we consider non-leaf nodes p, with the following clauses (OR processes),

$$C_1 : p \quad :- \quad p_{11}, \ldots, p_{1n_1}.$$

$$C_2 : p \quad :- \quad p_{21}, \ldots, p_{2n_2}.$$
$$\ldots$$
$$C_k : p \quad :- \quad p_{k1}, \ldots, p_{kn_k}.$$

Let the cost of each clause be $\text{Cost}_{C_j}(n)$ for $1 \leq j \leq k$. We now distinguish whether or not p is recursive.

Case 2: p is not recursive and not mutually recursive with any other procedures. We can easily see that

$$\text{Cost}_p(n) \leq \sum_{j=1}^{k} \text{Cost}_{C_j}(n). \tag{1}$$

Conservatively, we approximate $\text{Cost}_p(n)$ as the right-hand side of the above inequality.

Case 3: p is recursive or mutually recursive. In this case, we must be careful in the approximation, since minor changes in the recurrence equations can give rise to very different estimation. This can be seen for `split` in `qsort` example in Section 2.

To be more precise, we first observe that some clauses are the "boundary clauses," that is, they serve as the termination of the recursion. The other clauses, whose bodies have some goals which are mutually recursive with p, are the only clauses which will be effective for the recursion. Without loss of generality, we assume for $j > u$, C_j are all those "mutually recursive" clauses. For a nonzero iteration parameter n (i.e., $n > 0$), we take the average costs of these clauses as an approximation:

$$\text{Cost}_p(n) = \frac{1}{k-u} \sum_{j=u+1}^{k} \text{Cost}_{C_j}(n) \tag{2}$$

and for $n = 0$, we take the sum of the costs of those "boundary clauses" as the boundary condition of $\text{Cost}_p(n)$:

$$\text{Cost}_p(0) = \sum_{j=1}^{u} \text{Cost}_{C_j}(0).$$

The above estimation only gives the relations between cost of p and those of its clauses. The cost of clause C_j can be estimated as

$$\text{Cost}_{C_j}(n) = \text{CHead}_j + \sum_{m=1}^{n_j} \text{Cost}_{p_{jm}}(I_{jm}(n)) \tag{3}$$

where CHead_j is a constant denoting the cost for head unification of clause C_j and $I_{jm}(n)$ is the iteration parameter for the m^{th} body goal. Substituting Equation 3 back into Equation 1 or 2 gives us the recurrence equations for cost functions of predicates.

4.2 Iteration Parameters

There are several intuitions behind the introduction of the iteration parameter. As we mentioned above, iteration parameter n represents an encoding of a program runtime state as a positive integer. In fact, this type of encoding has been used extensively in program verification, e.g., [3], especially in the proof of loop termination. A loop $\mathcal{L}$ terminates if and only it is possible to choose a function M which always maps the runtime states of $\mathcal{L}$ to nonnegative integers such that M monotonically decreases for each iteration of $\mathcal{L}$. Such encoding also makes it possible to solve the problem that once the cost of an active goal is given, its iteration parameter can be obtained. This parameter can be used to derive costs of its subgoals (provided the iteration-parameter functions I_m are given), which in turn give the cost distribution functions.

Admittedly, the encoding of program states may be fairly complicated. Hence, to precisely determine the iteration-parameter functions for subgoals will be complicated too. In fact, this problem is statically undecidable since this is as complicated as to precisely determine the program runtime behavior at compile time. Fortunately, in practice, most programs exhibit regular control structures that can be captured by some intuitive heuristics.

To determine the iteration-parameter functions, we first observe that there is a simple conservative rule: for a recursive body goal p, when it recursively calls itself back again, the iteration parameter must have been decreased by one (if the recursion terminates). This is similar to the loop termination argument. Therefore, as an approximation, we can use $I_m(n) = n - 1$ as a *conservative estimation* for a subgoal p_{im} which happens to be p (self-recursive). Other heuristics are listed as follows:

§1. For a body goal p_{im} whose predicate only occurs in the body once and it is not mutually recursive with p (i.e., not in a strongly-connected component of p), $I_{im}(n) = n$.

§2. If p_{im} is mutually recursive with p and its predicate only occurs once in the body, $I_{im}(n) = n - 1$.

§3. If p_{im} is mutually recursive with p and its predicate occurs l times in the body, where $l > 1$, $I_{im}(n) = n/l$ (this is integer division, i.e., the floor function).

The intuitions behind these heuristics are simple. Heuristic §1 represents the case where a goal does not invoke its parent. In almost all programs, this goal will process information supplied by the parent, thus the iteration parameter remains unmodified. Heuristic §2 is based on the previous conservative principle. Heuristic §3 is based on the intuition that the iteration is divided evenly for multiple callees. Notice for the situation in heuristic §3, we can also use our conservative principle. However, we avoid use of the conservative principle, if possible,

because the resultant estimation of $\text{Cost}_p(n)$ may be an exponential function of n, which, for most practical programs, is not correct.

These heuristics have been derived from experimentation with a number of programs, placing a premium on the *simplicity* of $I(n)$ [13]. A remaining goal of future research is to further justify these heuristics with larger programs, and derive alternatives.

4.3 An Example: Quicksort

After we have determined the iteration-parameter functions, we have a system of recurrence equations for cost functions. These system of recurrence equations can be solved in a bottom-up manner in the modified graph G'. The problem of systematically solving these recurrence equations in general is discussed in [13]. Here, we consider a complete example for the `qsort/2` program given in Figure 2.

The boundary condition for $\text{Cost}_{qsort}(n)$ is that $\text{Cost}_{qsort}(0)$ is equal to the constant execution cost d_1 of `qsort/2` clause one. The following recurrence equations are derived:

$$\begin{aligned} \text{Cost}_{qsort}(0) &= d_1 \\ \text{Cost}_{qsort}(n) &= \text{Cost}_{C_2} \end{aligned}$$

With Heuristic §2, we have

$$\text{Cost}_{C_2} = d_2 + \text{Cost}_{split}(n) + 2\text{Cost}_{qsort}(n/2)$$

where d_2 is the constant cost for the head unification of the second clause of `qsort/2`.

Similarly, the recurrence equations for $\text{Cost}_{split}(n)$ are

$$\begin{aligned} \text{Cost}_{split}(0) &= d_3 \\ \text{Cost}_{split}(n) &= (\text{Cost}_{C_2} + \text{Cost}_{C_3})/2 \end{aligned}$$

Furthermore,

$$\begin{aligned} \text{Cost}_{C_2} &= \text{Cost}_{C_3} \\ &= d_4 + \text{Cost}_{split}(n-1) \end{aligned}$$

where d_4 is the constant cost for the head unification of the second (and the third) clause of `split`. We first solve the recurrence equations for `split`, which is in the lower level in G' and and then solve the recurrence equations for `qsort`. This gives us $\text{Cost}_{split}(n) = d_3 + d_4 n$ which

can be approximated as $d_4 n$ and $\text{Cost}_{qsort}(n) = d_1 + d_2 \log n + d_4 n \log n$, which is the well known average complexity of **qsort**.

Finally, it should be noted that it is necessary to distinguish between the recursive and nonrecursive clauses here and take the average of the recursive clause costs as an approximation. If we simply take the summation of all clause costs together as the approximation of the cost function, both cost functions for **split** and **qsort** would be exponential, which are not correct. More precisely, if the summation of all costs of clauses of **split** is taken as $\text{Cost}_{split}(n)$, we will have

$$\text{Cost}_{split}(n) = d_3 + 2(d_4 + \text{Cost}_{split}(n-1))$$

The solution of $\text{Cost}_{split}(n)$ is an exponential function, which is not correct.

5 Distributing Costs

After we have derived functions of the iteration parameter for each procedure, we are now ready to derive cost distributing formulae for a given procedure and its body goals. The first step is to solve for the iteration parameter n in Equation 3 assuming that $\text{Cost}_p(n)$ is given at runtime as C_p. Assuming that clause i is invoked in runtime, we approximate $\text{Cost}_{C_i}(n)$ as C_p and solve Equation 3 for n. Let $n = F(C_p)$ be the symbolic solution, which depends on the runtime value of $\text{Cost}_p(n)$ (i.e., C_p), we can easily derive costs of its subgoals of clause i as we can simply substitute n with $F(C_p)$ in $\text{Cost}_{p_{im}}(I_{im}(n))$, which gives rise to the cost distributing functions we need to derive at compile time.

Let's reconsider the **nrev/2** procedure. The cost equations are derived as follows:

$$\begin{aligned}
\text{Cost}_{nrev}(n) &= \text{Cost}_{nrev}(n-1) + \text{Cost}_{append}(n) \\
\text{Cost}_{nrev}(0) &= c_1 \\
\text{Cost}_{append}(n) &= \text{Cost}_{append}(n-1) + C_a \\
\text{Cost}_{append}(0) &= c_2
\end{aligned}$$

We can easily derive the closed forms for these two cost functions as $\text{Cost}_{append}(n) = n \times C_a + c_2$ which can be approximated as $C_a \times n$, and $Cost_{nrev}(n) \doteq C_a \times n^2/2$. Now, given the $\text{Cost}_{nrev}(n)$ as C_r, we solve for n and have $n = \sqrt{\frac{2C_r}{C_a}}$. Hence, we have $\text{Cost}_{nrev}(n-1) = C_a(\sqrt{\frac{2C_r}{C_a}} - 1)^2/2$ and $\text{Cost}_{append}(n) = C_a\sqrt{\frac{2C_r}{C_a}}$. These are the desired cost distributing functions.

It should be pointed out that in some cases, it is not necessary to first derive the cost functions and then derive the cost distributing functions since we can simply derive the cost

distributing scheme directly from the cost recurrence equations. For example, consider the Fibonacci function, where the cost equations are

$$\begin{aligned} \text{Cost}_{fib}(n) &= C_f + 2 \times \text{Cost}_{fib}(n/2) \\ \text{Cost}_{fib}(0) &= C_1 \end{aligned}$$

Without actually deriving the cost functions of $\text{Cost}_{fib}(n)$, we can simply derive the cost distributing relationship from the first equation as $\text{Cost}_{fib}(n/2) = (\text{Cost}_{fib}(n) - C_f)/2$.

Also note that at compile time, the cost distributing functions should be simplified as much as possible to reduce the runtime overhead. It is even worthwhile sacrificing precision to get a simpler function. Therefore, a conservative approach should be used to derive the upper bound of the cost functions. In fact, we can further simplify the cost function derived in the following way. If the cost function is of a polynomial form such as $c_0 n^k + c_1 n^{k-1} + \ldots c_k$, we simplify it as $k c_0 n^k$ and if the cost function is of several exponential components such as $c_1 a^n + c_2 b^n$ where $b > a$, we simplify it as $(c_1 + c_2) b^n$. This will simplify the solution of the iteration parameter and the cost distributing function and hence simplify the evaluation of them at runtime.

5.1 Runtime Goal Management

The above cost relationship estimation is well suited for a runtime scheduler which adopts an on-demand scheduling policy (e.g., [8]), where PEs maintain a local queue for active goals and once a PE becomes idle, it requests a goal from other PEs. A simple way to distribute a goal to a requesting PE is to migrate an active goal in the queue. The scheduler should adopt a policy to decide which goal is going to be sent. It is obvious that the candidate goal should have the maximal grain size among those goals in the queue. Hence, we can use a priority queue where weights of goals are their grain sizes (or costs). The priority is that the bigger the costs are, the higher priority they get. Because the scheduler only needs to know the relative costs, we can always assume the weight of the initial goal is some fixed, big-enough number. Based on this initial cost and the cost distributing formulae derived at compile time, every time a new clause is invoked, the scheduler derives the relative costs of body goals. The body goals are then enqueued into the priority queue based on their costs.

Some bookkeeping problems arise from this approach. First, even though we can simplify the cost distributing functions at compile time to some extent, the runtime overhead may still be large, since for each procedure invocation, the scheduler has to calculate the weights of the body goals. One solution to this problem is to let the scheduler keep track of a modulo counter and when the content of the counter is not zero, the scheduler simply lets the costs of the body goals be the same as that of their parent. Once the content of the counter becomes zero, the cost-distributing functions are used. If we can choose an appropriate counting period, this

method is reasonable (one counter increment has less overhead than the evaluation of the cost estimate).

Another problem in this approach is that for long-running programs, costs may become negative, i.e., the initial weight is not large enough. Since we require only relative costs, a solution is to reset all costs (including those in the queue, and in suspended goals), when some cost becomes too small. Cost resetting requires the incremental overhead of testing to determine when to reset.

6 Conclusions and Future Work

We have proposed a new method to estimate the relative costs of procedure execution for a concurrent language. The method is similar to Tick's static scheme [10], but gives a more accurate estimation and reflects runtime weight changes. This is achieved by the introduction of an iteration parameter which is used to model recursions.

Our method is based on the idea that it is not the absolute cost, but rather the relative cost that matters for an on-demand goal scheduling policy. Our method is also amenable to implementation. First, our method can be applied to any program. Second, the resultant recurrence equations can be solved systematically. In comparison, it is unclear how to fully mechanically implement the schemes proposed in [2, 4]. Nonetheless, our method may result in an inaccurate estimation for some cases. This is because we use only the call graph to model the program structure, not the data. We admit that further static analysis of program structure such as argument-size relationships can give more precise estimations.

Future work in granularity analysis includes the development of a more systematic and precise method to solve the derived recurrence equations. It is also necessary to examine this method for practical programs, performing benchmark testing on a multiprocessor to show the utility of the method.

Acknowledgements

E. Tick was supported by an NSF Presidential Young Investigator award.

References

[1] S. K. Debray. A Remark on Tick's Algorithm for Compile-Time Granularity Analysis. Research note, Department of Computer Science, University of Arizona, June 1989.

[2] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *SIGPLAN Conference on Programming Language Design and Implementation*,

pages 174–188. ACM Press, June 1990.

[3] D. Gries. *Science of Programming.* Springer-Verlag, 1989.

[4] A. King and P. Soper. Granularity Control for Concurrent Logic Programs. In *International Computer Conference*, Turkey, 1990.

[5] B. Kruatrachue and T. Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, pages 23–32. January 1988.

[6] C. McGreary and H. Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. *Communications of the ACM*, 32:1073–1978, 1989.

[7] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* MIT Press, Cambridge MA, 1989.

[8] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *IFIP Working Conference on Parallel Processing*, pages 305–318. Pisa, North Holland, May 1988.

[9] R. E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *Regional Conference Series in Applied Mathematics.* Society for Industrial and Applied Mathematics, Philadelphia PA, 1983.

[10] E. Tick. Compile-Time Granularity Analysis of Parallel Logic Programming Languages. *New Generation Computing*, 7(2):325–337, January 1990.

[11] E. Tick. *Parallel Logic Programming.* MIT Press, Cambridge MA, 1991.

[12] K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 1, pages 140–156. MIT Press, Cambridge MA, 1987.

[13] X. Zhong, E. Tick, *et al.* Towards an Efficient Compile-Time Granularity Analysis Algorithm. Technical Report CIS-TR-91-19, University of Oregon, Department of Computer Science, September 1991.

GST: Grain-Size Transformations for Efficient Execution of Symbolic Programs

Extended Abstract

Andrew A. Chien and Wuchun Feng
achien@cs.uiuc.edu *feng@cs.uiuc.edu*
University of Illinois
Department of Computer Science
1304 W. Springfield Avenue
Urbana, IL 61801

October 1, 1991

1 Introduction

Controlling grain size is a key issue which spans programming approaches and machine architectures in parallel systems. The ability to effectively adjust program grain size, is a critical component of achieving efficient, portable parallel programming. A number of programming models have been used to express application programs with large quantities of fine-grained concurrency. Unfortunately, despite rapid improvements in processor architecture, we currently cannot exploit such fine-grained concurrency efficiently. Even if such architectures were available, portability issues motivate the development of grain-size control techniques. We are pursuing the construction of a grain-size transformer system which merges grains, increasing the execution grain size for more efficient execution.

2 Background

We are building programming systems for MIMD, distributed-memory machines. We focus on such machines not only because they represent a scalable hardware architecture, but also because they have no built-in policies for data movement. From a software perspective, they represent the cleanest slate. Their message-passing structure makes communication

explicit, forcing the software to manage it and allowing it to be optimized explicitly. Despite our focus on a particular machine organization, the techniques developed are applicable to any machines for which increased locality improves performance.

If program granularity can be adjusted over a sufficiently wide range, the programs which express fine-grained concurrency can be executed efficiently on both fine-grained and medium-grained architectures. The fine-grain programs can become the basis for portable, parallel programming on a family of distributed memory machines.

We would like to support both numeric and symbolic computing which involve complex data structures. The issue is the complexity of data structures and the prevalence of pointers, not the computational methods or even the application being solved.

We have been actively involved in the design and implementation of Concurrent Smalltalk (CST) [1] and Concurrent Aggregates (CA) [2], both concurrent object-oriented programming systems. These systems were initially developed to program the J-machine [3], a fine-grained MIMD distributed-memory machine. We describe our developments in the context of an object-based concurrent system [4]. However, our techniques should be directly applicable to most Actor languages [5, 6] and as similar to fold/unfold transformations [7] in committed-choice logic languages [8, 9]. Extension to other programming paradigms which do not bind program and data closely, such as functional programming approaches, may require a different approach.

3 An Approach to Grain-Size Transformation

The objective of grain-size transformation is to adjust the *dynamic* execution grain size to increase the efficiency of execution. We define a computation grain as the unit of work performed in response to a message arrival. Computation grains are terminated for two reasons: remote data access and synchronization. Our approach addresses both causes by constraining data placement and merging units of synchronization based on invocation relations. We constrain data placement in a process called *abstract placement*. The merging of synchronization units is termed *object coalescing*.

We assume that the program has been initially formulated to express concurrency at the object level. Consequently, transformations to increase task granularity involve attempts to productively merge objects and their invocations. Our program transformations produce an abstract data placement, a set of constraints which the run-time system must enforce for correct execution, as well as program code optimized to execute with that data placement. Together the abstract data placement and optimized code constitute a new program with a larger execution grain size.

Efficient invocation in message-passing machines requires knowledge of data placement.

In recent years, the distance between shared-memory and message-passing machines has decreased. Shared memory machines now have memory hierarchies with locality, and message-passing machines support shared address spaces. However, one important remaining difference is that message-passing machines have distinguished local and non-local access mechanisms. This implies that the most efficient forms of local and non-local invocation require distinct calling sequences in message-passing machines. Constraining data placement allows us to avoid using the expensive remote-invocation sequence in many cases. In contrast, shared memory machines use automatic data relocation, encouraging compilation to a uniform, local calling sequence.

4 Abstract Placement and Object Coalescing

Abstract placement involves the constraining of data placement in a program execution. Define the data placement constraints, C, as a set of locality sets:

$$C = \{ls_0, ls_1, ls_2, ...\}$$

Each of the locality sets, ls, contains objects. Each object is a member of exactly one locality set.

$$ls_m = \{obj_i, obj_j, obj_k, ...\}$$

Membership in the same locality set implies a data placement constraint. All objects in a locality set must be placed together[1]. Additional data placement constraints can reduce communication and linkage requirements, but for maximum benefit, such constraints should be added in accord with the invocation structure between objects. Data placement restrictions can be used to reduce the number of invocation overheads along a chain of references.

Object coalescing involves the merging of synchronization units to reduce synchronization overhead. The idea is analogous to the notion of reducing blocking in real-time systems [10]. While abstract placement can reduce communication requirements and linkage overhead for tasks, it may not increase the execution grain-size. In our model, objects are units of synchronization and hence exclusion. Invocations, even if they are local, cross these object boundaries and thus may suspend, terminating the grain.

We define a similar formalism for synchronization units. We define the set of execution objects, as distinguished from user-defined object boundaries, as a set of synchronization

[1] More precisely, it must be possible to address them in the same address space and use the local invocation mechanisms, including inlining, to couple computation amongst them.

sets. These sets are units of exclusive access with respect to program execution. Initially, each object belongs to a unique synchronization set. Object coalescing transformations join the synchronization sets, reducing synchronization overhead. Typically, object coalescing requires both objects to already be members of the same locality set. In such circumstances, the synchronization of the new unit can be achieved efficiently. Object coalescing can be used to reduce the number of synchronization operations along a path or reduce the interface concurrency of a multi-access data abstraction, such as an aggregate in CA.

Naming Issues In order to formulate issues of data placement in a dynamic storage allocation environment, we must be able to identify data structures and their interrelationships. In order to effect our grain-size transformation (placement restriction and object coalescing), we must identify their point of allocation. Acquiring the full knowledge required to identify all program data structures would require full program execution and therefore is not feasible. Instead, we have adopted a scheme based on the notion of alias graphs [11] to name objects at compile time. With alias graphs, we can identify the allocation points for root, intermediate nodes, and leaves of nested structures, providing the necessary control to implement data placement decisions and object coalescing. The advantage of alias graphs is that they can be extended or compressed to trade off compilation cost for more accurate alias information. In a grain-size transformer system, the amount of alias information (the depth of the graphs) is related to the amount of transformation we wish to do. The compilation cost increases for more aggressive modification of program granularity.

Inferring accurate aliasing information can be difficult, depending on the complexity of program structure. In Larus' Curare system for parallelizing Scheme, alias graph structures were inferred, but programmer annotations could be used to refine alias graph information. We are pursuing a similar approach for deriving object invocation and sharing relationships. Several reasons suggest that our system may be able to derive enough information to significantly transform program grain size. First, many symbolic computations use multiple layers of static structuring – data abstractions built with objects, constructive program reuse. These layers can be effectively deduced and combined. Second, repeating sequences of structures such as pairs in a list can deduced and used to increase grain size on sequence-oriented phases of computation. Third, aggregate data abstractions which involve bulk allocation such as in Concurrent Aggregates allow the expression of data-parallel operations, a natural target for grain-size adjustment [12].

Transformation Conditions Merging of locality sets and synchronization units is not done arbitrarily. In each case, we must assure appropriate conditions to avoid changing program functionality or reducing performance. While the merging of locality sets does not alter a program's functional behavior, it may affect its performance. Constraining two

sets of objects to be placed together limits their collective concurrency to that available at the local node[2]. Naturally, this limit increases in significance as the set size increases. In addition, there is a physical limitation to the memory resources at a computing node. A more relevant restriction is the desire to keep storage units small, leaving the run-time system with some opportunity to perform load balancing for computation and memory usage.

Merging synchronization units is quite tricky. Synchronization units are part of the programming model and merging them arbitrarily may cause deadlock. In order to determine when units can be merged safely, we need sharing and invocation information. We obtain this information from conventional control and data flow analysis of our programs. To deduce sharing relationships we use alias graphs as well. Static study of a number of programs leads us to believe that there are many cases in which merging can be done safely. We are currently collecting dynamic statistics to reconfirm our findings. While merging synchronization domains can reduce synchronization overhead, it also limits concurrency within the synchronization unit. Typically, the concurrency within a single unit will be limited to a single thread.

The basic idea is to allow a programmer to specify concurrency at a fine-grain, easing program construction and enhancing portability. Typically, this involves specifying small data components and expressing the concurrency in operations on them. The compiler adjusts the specified program grain-size to an execution grain size suitable for the execution engine at hand. The net effect is to increase the size of structure components and the amount of work associated with each invocation on that structure. In order to achieve a real increase in grain size, we must simultaneously deal with issues of data locality and synchronization

5 Dynamic Techniques

The program transformation techniques we have described can also be used dynamically. The approach we are pursuing here is directly analogous to the dynamic optimization techniques used by Chambers and Ungar in SELF [13, 14] to reduce invocation overhead in object-oriented systems. Instead, we are studying the use of dynamic program optimization to reduce linkage overhead due to data placement. Speculative application of these techniques must be based on knowledge of the dynamic characteristics of concurrent object-oriented programs. We are currently pursuing such a study. Data location transformations and code customization can be done without deadlock-safety concerns and will give correct execution so long as the customization is reversible. The sharing information required for

[2]Future distributed memory machines may use multiprocessor nodes, but current machines only have uniprocessor nodes.

object coalescing could be derived from reference counting, if used for storage reclamation. However, even with such information, assuring that an object will never be shared requires some analysis.

6 Summary

Grain-size adjustment is an issue which must be addressed by any portable parallel programming system. We are developing a program transformation system which constrains the placement of data and transforms control structures to increase the execution grain size of object-based concurrent programs. While it is too early to say how successful this approach will be, we are confident that a combination of automatic and programmer-aided techniques will give us the ability to transform grain sizes over a modest range. The ultimate product will be a modestly portable parallel programming system.

References

[1] W. Horwat, A. Chien, and W. Dally, "Experience with cst: programming and implementation," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pp. 101–9, ACM SIGPLAN, ACM Press, 1989.

[2] A. A. Chien and W. J. Dally, "Concurrent aggregates (ca)," in *Proceedings of Second Symposium on Principles and Practice of Parallel Programming*, ACM, March 1990.

[3] W. J. Dally, A. Chien, S. Fiske, W. Horwat, J. Keen, M. Larivee, R. Lethin, P. Nuth, S. Wills, P. Carrick, and G. Fyler, "The j-machine: a fine-grain concurrent computer," in *Information Processing 89, Proceedings of the IFIP Congress*, pp. 1147–1153, Aug. 1989.

[4] P. Wegner, "Dimensions of object-based language design," in *Proceedings of OOPSLA '87*, pp. 168–82, 1987.

[5] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems.* Cambridge, MA: MIT Press, 1986.

[6] G. Agha, "Concurrent object-oriented programming," *Communications of the Association for Computing Machinery*, vol. 33, pp. 125–41, September 1990.

[7] T. Kawamura and T. Kanamori, "Preservation of stronger equivalence in unfold/fold logic program transformation," in *Proceedings of the International Conference on Fifth Generation Systems*, (Tokyo, Japan), pp. 413–421, ICOT, 1988.

[8] V. Saraswat, K. Kahn, and J. Levy, "Janus: a step towards distributed constraint programming," in *Proceedings of the North American Conference on Logic Programming*, (Austin, Texas), October 1990.

[9] I. Foster and S. Taylor, *Strand: New Concepts in Parallel Programming.* Prentice-Hall, 1990.

[10] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–85, 1990.

[11] J. R. Larus and P. N. Hilfinger, "Detecting conflicts between structure accesses," in *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 21–33, ACM, 1988.

[12] P. Hatcher, M. Quinn, A. Lapadula, B. Seevers, R. Anderson, and R. Jones, "Data-parallel programming on mimd computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 377–383, 1991.

[13] C. Chambers and D. Ungar, "Iterative type analysis and extended message splitting," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pp. 150–60, 1990.

[14] C. Chambers and D. Ungar, "Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language," in *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pp. 146–60, 1989.

Using Domain-Specific, Abstract Parallelism

Ira Baxter and Elaine Kant
Schlumberger Laboratory for Computer Science
8311 North RR 620
Austin, Texas, 78720-0015
baxter@slcs.slb.com, kant@slcs.slb.com

Abstract

Discovery of potential parallelism in low level code is difficult, especially in the absence of problem domain knowledge. An alternative is to explicitly represent maximal potential parallelism in abstract program components. A transformation system refines a program composed of such components into a concrete program. We discuss an experimental system in which we are installing such facilities. An example refinement sequence is provided.

1 Introduction

Compiling problem-domain independent program representations for parallel architectures is often difficult because of the need to infer opportunities for parallelism. Because safe inference of parallelism must be conservative, the inferred parallelism is often considerably less than that actually available in the applications. This problem leads to a demand for tools such as E/SP [SMD+89], ParaScope [BKK+89] and MIMDizer [Cor90], which identify points of potential, but unverifiable, parallelism and query the programmer to determine a less conservative version of the truth. Inference and query-the-programmer are both methods for rediscovering the parallelism. All of this would be unnecessary if the knowledge of what was parallel at the time of program construction were not lost.

An alternative approach we are pursuing is to capture the inherent parallelism (actually, absence of execution ordering constraints) in an abstract program in a domain-specific fashion. Then a transformation system would refine not only the program but also the parallelism information into the concrete program. In this fashion both the expense of the conservative inference and the need to query the programmer are minimized.

In this paper, we give a short example of an abstract domain-specific component whose full parallelism is "refined away" (rather than rediscovered) until it is usable on a particular target machine. We also briefly motivate the need for non-tree-structured internal representations.

2 Problem Domain

SINAPSE [KDMW90] [KDMW91] is an experimental tool to synthesize mathematical modeling programs for a variety of similar applications. These have, to date, been primarily acoustic wave propagation problems, typically used to validate geophysical models for oil exploration.

SINAPSE accepts specifications of typically 20 to 50 lines, and produces C, Fortran, or Connection Machine Fortran programs that solve the differential equations related to the problem domain by using a finite differencing method. Resulting programs are typically 500 to 1500 lines in size; lines are often very dense.

A number of programs generated by SINAPSE have produced useful scientific results for Schlumberger modelers after some post-generation hand optimization. The work described here is part of research aimed at automating that optimization.

Synthesizing modeling programs requires knowledge of the wave propagation problem domain, knowledge about solution techniques for problems in that domain, general programming knowledge, and control knowledge to sequence the synthesis process. This class of program provides many opportunities for data-parallel computation [HS86]; consequently, knowledge of potential parallelism and when to use it is also useful.

3 Synthesis Process

User-specified algorithm schemas are refined by repeatedly replacing schema components with lower-level schemas or parameter values. These component replacements are taken from knowledge bases selected by, or computed directly from, the specification. Rather than being the initial abstract program, the specification simply directs the choice of schemas and parameter values.

Algorithm schemas are stated in terms of a high-level programming language called "algSinapse," which includes assignments, conventional control constructs, array and scalar computations, references to parameters, and references to other algorithm schemas.

Generic programming knowledge as well as application domain knowledge is needed to produce efficient programs. Much of the programming knowledge is in the form of algorithm refinements that expand constructs such as parallel enumeration or matrix multiplication into built-in constructs or a combination of loops and scalar operations depending on the target architecture and language. Rather than having a runtime library of special-case methods (e.g., different matrix multiplications for diagonal arrays), SINAPSE derives the special methods directly. This is accomplished by substituting representations, determined by explicitly represented properties of interest (e.g., DIAGONAL-ARRAY or SYMMETRIC), for references to values, and simplifying away unneeded operations and combining similar terms. This avoids the need to rewrite such libraries for each new target language. The approach is made feasible by the use of Mathematica [Wol91], a symbolic manipulation language as an implementation platform. There are also a number of optimizing transformations.

One of the problems of refining abstract schemas into real programs are inefficiencies introduced because of necessarily conservative analysis of the original schemas. These come about simply because schemas, while optimized maximally on an individual basis, may be more optimizable when combined.

One can resolve this problem in a number of ways, of which SINAPSE currently uses two:

- General purpose optimization techniques, and
- Special case algorithm schemas.

SINAPSE has an optimizer which moves static computations outside of loops. Abstract computations are often placed inside a loop in an originating schema simply because domain knowledge tells us they are usually loop-index dependent. Only when the expression is actually instantiated can we determine the actual loop dependency. If domain knowledge tells us that some expression is always loop independent, then it can be encoded outside the loop in the schema.

The optimizer simply moves blocks of code earlier into the computation as long as this is consistent with the data-flow constraints. This often moves code outside of loops. The moved code is placed in parallel with the earliest statement it can precede. Thus, a free side effect of running the code motioner is the conversion of unnecessary sequencing constructs into parallel execution constructs. A special mechanism detects when expressions dependent only on loop indices can be moved outside the loop. The values of these expressions will be cached in a array. More specifically, storage for the array is allocated, code to fill the array is generated outside the loop, and the cached values from the array are referenced inside the loop. We plan to add a common-subexpression eliminator.

Considerable payoff also occurs when the problem or target domain dictates certain properties of the code; one can then optimize a schema in advance of supplying it to SINAPSE, thereby avoiding the expense of dynamic optimization at program synthesis time. A price is paid for this: manual encoding of such optimized schemas at synthesizer-construction time, and conditioning the instantiation of the special case schemas on the domain property.

4 A Weak Representation for Parallelism

SINAPSE currently represents abstract programs as tree schemas containing various control constructs representing explicit classes of parallelism:

- $seq[s_1, s_2, \ldots, s_n]$
  Sequencing of state-changing constructs s_i
- $doSeq[s, j, lb, ub]$
  Iteration of statement s requiring sequential execution with index j in range $lb \ldots ub$
- $par[s_1, s_2, \ldots, s_n]$
  Arbitrary execution ordering of state-changing constructs s_i
- $doPar[s, [j_1, lb_1, ub_1], [j_2, lb_2, ub_2], \ldots, [j_n, lb_n, ub_n]]$
  Parallel execution of (possibly compound) statement s instantiated with simultaneous assignment of loop indices j_i

doPar provides much of the opportunity to generate data-parallel programs for the Connection Machine 2 (CM2), written in CM Fortran 90. As an example, the following construct:

$$doPar[A[j] = B[j] * k - C[j], [j, 1, size(A)]]$$

is converted into the Fortran 90 array statement:

$$A[1 : size(A)] = B[1 : size(A)] * k - C[1 : size(A)]$$

When the rank of a target array does not match the rank of a source, then the Fortran 90 intrinsic function:

$$\textbf{SPREAD}(value, axisNumber)$$

is generated to expand the source array along necessary axes.

SINAPSE algorithm schemas also allow the expression of computations on entire arrays, which pass through virtually unchanged to CM Fortran. SINAPSE replaces entire-array operations with *doPar* equivalents when the target is sequential Fortran 77. It is then trivial to generate corresponding sequential code for any *par* and *doPar* constructs.

An explicit concession to data parallelism used in our representation is a variant of *doPar*:

$$makeArray[f(j_1, j_2, \ldots, j_n), [j_1, lb_1, ub_1], [j_2, lb_2, ub_2], \ldots, [j_n, lb_n, ub_n]]$$

which constructs a rank n array for which each element value is defined by the function f, usually instantiated as an expression over the index variables.

While this seems to work well for pure data-parallel constructs (for SIMD target machines such as the CM2), this representation is too weak to represent more general parallelism. Consider four computations A, B, C, and D, with the requirements that A occur before C, and that B occur before C and D; the present primitives can at best express only overly-constrained versions of the requirements, thus losing the ability to explicitly represent the potential parallelism. The partial order in which finite-difference equations must be evaluated is one such example. In general, tree-structured representations cannot capture partial orders (without resorting to some kind of context-sensitivity).

5 Proposed Representation

We are considering using a variant of "Unified Computation Graphs" (UCGs) [WBS+91] to represent programs. Such graphs are based on simple data-flow graphs, with the addition of shared data, control-flow arcs (similar to program dependence graphs [FOW87]) and "exclusion constraints" between nodes. Exclusion constraints prevent two or more parallel activities from simultaneous execution, and are usually associated with access to overlapping parts of a shared data structure.

In Figure 1, we show computations as bubbles, data- and control-flow as solid arrows, and exclusion dependencies as a dashed arc with no arrows. Primitive computations in bubbles are represented as trees.

UCGs assume global shared data among computations, while pure data-flow assumes no shared data. We have added a representation for data shared among particular nodes, and show the storage shared by two computations with an enclosing dashed arc. Computations not sharing data with others are not necessarily functional; each may still have internal state. Parallel-prefix operations may be represented either by reduction operators over data aggregates, such as the Fortran 90 **SUM** operation, or explicitly via n-ary trees on explicitly represented operands. We currently do not represent pipeline parallelism or multiple simultaneous activations of each operator [AG82].

We mix notations by writing (sub)UCGs isomorphic to purely parallel (respectively sequential) constructs as their textual *par* (respectively *seq*) equivalent.

5.1 Refinements on UCGs

A refinement is a type of transformation that introduces detail (i.e., removes possible models). Several generic refinements of standard UCGs are possible:

R_{comp} Refine a computation into a sub-UCG (Figure 2).

R_{flow} Refine a data-flow carrying a complex data structure into multiple data-flows carrying parts of that structure (composing this with the previous action produces data-parallel computations when data-flow components are homogeneous).

$R_{abstract}$ Group parallelizable computations into a single computation.

R_{merge} Merge a set of parallel computations into a single computation.

$R_{serialize}$ Sequentialize a pair of parallel activities by adding control-flow arc.

$R_{sequence}$ Refine an exclusion constraint arc into a control-flow arc going in either direction.

New refinements possible because of the enriched representation:

R_{store} Refine data-flow between nodes into control-flow plus shared storage.

$R_{coalesce}$ Coalesce several shared storage regions into one.

6 Example

In this section, we sketch the refinement of a domain-specific component into good CM2 code, given our proposed representation.

We present an equivalent Fortran 77 code fragment first, abstracted from a real modeling program, to ensure that the reader initially sees what a conventional compiler sees. A conventional compiler must determine which of these steps can be executed in parallel, knowing nothing of the intent.

```
      DO 100, ix=K+1,K+N
         DO 100, iy=K+1,K+N
100   padarray(ix,iy)=mu(ix-K,iy-K)
      <...lots of unrelated code...>
      DO 128 iy=1,N
         DO 128 ix=1,K
128   padarray(ix,iy) = padarray(K+1,iy)
      DO 129 iy=1,N
         DO 129 ix=N-K+1,N
129   padarray(ix,iy) = padarray(N-K,iy)
      DO 132 ix=1,N
         DO 130 iy=1,K
130      padarray(ix,iy) = padarray(ix,K+1)
         DO 131 iy=N-K+1,N
131      padarray(ix,iy) = padarray(ix,N-K)
```

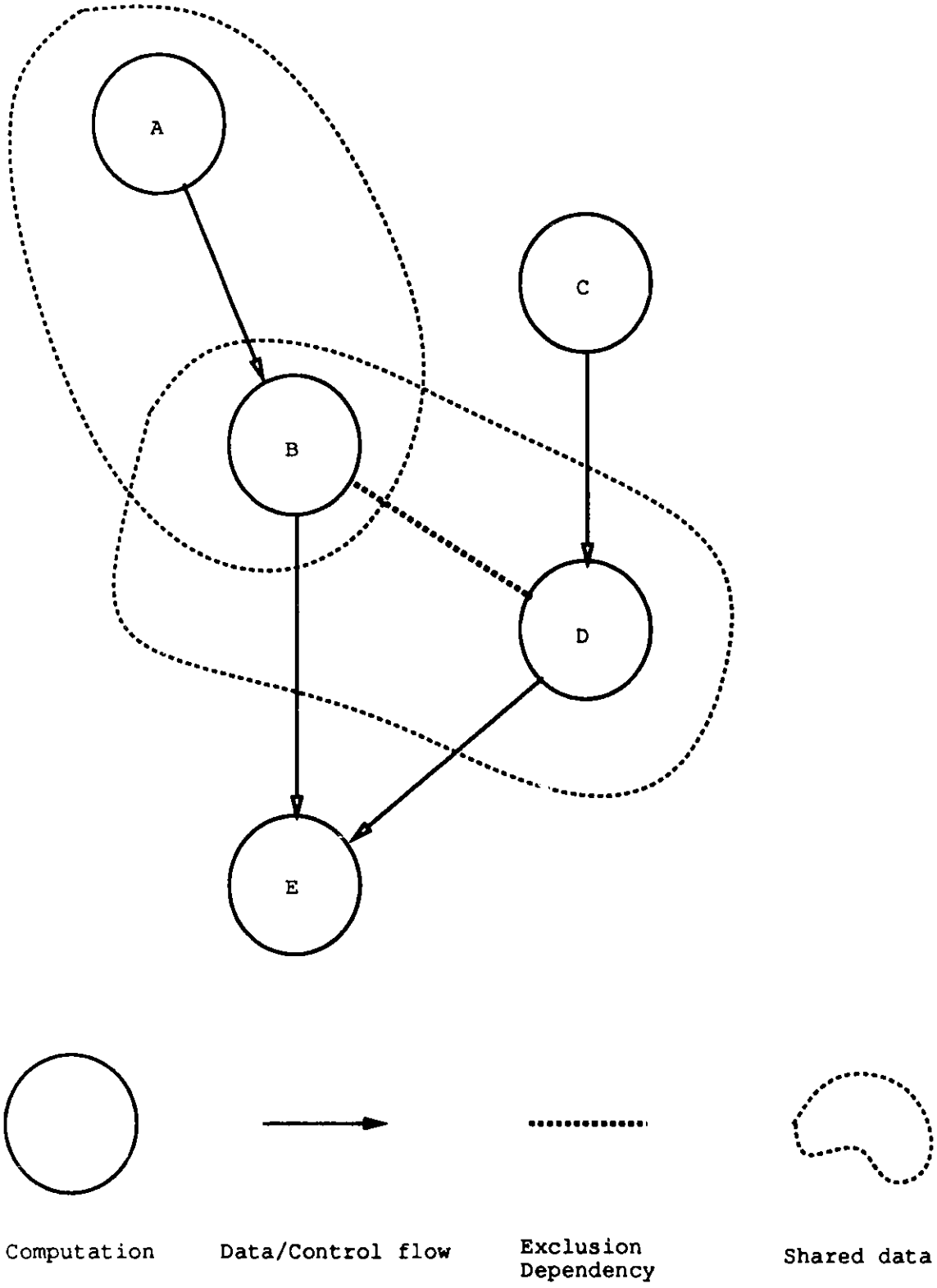


Figure 1: Representation of Parallelism

```
132  CONTINUE
```

This code actually pads an $N \times N$ array (**mu**), producing an $(N + 2 * K) \times (N + 2 * K)$ array (**padarray**) with a K-wide "taper" region along all edges. This is a common operation in modeling codes on variables representing properties of space when taper boundary conditions are used [IO81]. The intent is to fill the taper boundary areas with copies of the nearest edge of the original array; a conceptual view of this operation is provided in Figure 3. The resulting array has nine regions. The taper edges are filled with copies of the corresponding edge of the original array; symmetry leads one to the conclusion that the corner regions of the padded array must be filled with values from the closest corner of the array. Each region is defined as the set of elements selected by the cross-product of a particular range of indices; the upper, left hand corner has range $[i, 1, K], [j, 1, K]$, etc.

We give an abstract program schema defining the domain-specific notion **Pad(array)**, using case analysis to determine in which region an element resides:

```
Pad[originalarray,K] isSchema
N:=AxisSize(originalarray);
makeArray[
  case[
    1<=i<=K and 1<=j<=K: originalarray[1,1]; (* upper left corner *)
    1<=i<=K and K+1<=j<=K+N: originalarray[1,j-K+1]; (* North side *)
    1<=i<=K and K+N+1<=j<=N+2*K: originalarray[1,N]; (* upper right corner *)
    K+1<=i<=K+N and 1<=j<=K: originalarray[i-K+1,1]; (* West side *)
    K+N+1<=i<=N+2*K and 1<=j<=K: originalarray[N,1]; (* lower left corner *)
    K+N+1<=i<=N+2*K and K<=j<=K+N: originalarray[N,j-K+1]; (* South side *)
    K+N+1<=i<N and K+N+1<=j<=N+2*K: originalarray[N,N]; (* lower right *)
    K+1<=i<=K+N and K+N+1<=j<=N+2*K: originalarray[i-K+1,N]; (* East side *)
    K+1<=i<=K+N and K+1<=j<=K+N: originalarray[i-K+1,j-K+1]; (* middle *)
      ], (* case *)
          [i,1,N+2*K],[j,1,N+2*K]]
```

This schema provides for maximum (data) parallelism, every element can be computed independently, and thus the entire computation takes only $O(1)$ time on an appropriate architecture. It could be used by SINAPSE whenever padding is required; no rediscovery of parallelism is required.

However, the computation would still be unnecessarily inefficient on a SIMD machine such as the CM2, for which a data parallel operation requires all processing elements (PEs) to perform the same instruction and then synchronize. Each PE must synchronously execute the entire **case** statement body. Assuming one machine instruction for each operand and operator, each case requires about 15 instructions, so the nine cases require about 135 instruction times.

We can lower this cost by eliminating runtime evaluation of the case bounds. The cases conveniently define SIMD-compatible partitions of the instruction streams. SINAPSE assumes that a case construct precisely covers its cases, with no overlap; thus all case clauses may be executed in parallel:

```
seq[  N:=AxisSize(originalarray);
      allocate(padarray,N+2*k,N+2*k); (* creates storage for padarray *)
      par[
        doPar[padarray[i,j]:=originalarray[i-K+1,j-K+1],
                             [i,K+1,K+N],[j,K+1,K+N]]; (* middle *)
        doPar[padarray[i,j]:=originalarray[i-K+1,1],
                             [i,K+1,K+N],[j,1,K]]; (* West side *)
        doPar[padarray[i,j]:=originalarray[i-K+1,N],
                             [i,K+1,K+N],[j,K+N+1,N+2*K]]; (* East side *)
        doPar[padarray[i,j]:=originalarray[1,1],
                             [i,1,K],[j,1,K]]; (* upper left corner *)
        doPar[padarray[i,j]:=originalarray[1,j-K+1],
                             [i,1,K],[j,K+1,K+N]]; (* North side *)
```

```
    doPar[padarray[i,j]:=originalarray[1,N],
                        [i,1,K],[j,K+N+1,N+2*K]]; (* upper right corner *)
    doPar[padarray[i,j]:=originalarray[N,1],
                        [i,K+N+1,N+2*K],[j,1,K]]; (* lower left corner *)
    doPar[padarray[i,j]:=originalarray[N,j-K+1],
                        [i,K+N+1,N+2*K],[j,K,K+N]]; (* South side *)
    doPar[padarray[i,j]:=originalarray[N,N],
                        [i,K+N+1, and K+N+1,N+2*K]]; (* lower right *)
     ]; (* par *)
  padarray] (* seq *)
```

Each of these cases now maps directly onto a Fortran 90 array primitive, and each would execute in just a few instructions on a CM2. However, the CM2 has only one set of data-parallel processors, so each "parallel" case competes for the data-parallel processor resource. Static resolution of this resource contention requires serializing access to the set of data-parallel processors, and consequently 9 units of time are actually taken. This can be reduced to 5 (as in the original hand-coded fragment) by combining steps. Consider Figure 4; in stage 1, after copying the original array (1 unit), we expand the copied array along the X-axis (1 unit in both directions); in stage 2, we expand the expanded array along the Y-axis (1 unit in both directions).

To make progress towards this reduction in effort, we apply the following refinements:

- Group ($R_{abstract}$) some parallel activities, with the intention of merging them (R_{merge}), and
- Order ($R_{serialize}$) some parallel activities, to eventually ensure that certain properties are present when needed.

The result is shown in Figure 5.

The activities so grouped can be combined into a single data-parallel primitive. This is because after copying to the center, and filling east and west edges, we have entire edge rows ready to replicate vertically, as shown in Stage 2 of Figure 4. Consequently we can rewrite the three steps:

```
par[
    doPar[padarray[i,j]:=originalarray[1,1],
                        [i,1,K],[j,1,K]]; (* upper left corner *)
    doPar[padarray[i,j]:=originalarray[1,j-K+1],
                        [i,1,K],[j,K+1,K+N]]; (* North side *)
    doPar[padarray[i,j]:=originalarray[1,N],
                        [i,1,K],[j,K+N+1,N+2*K]]; (* upper right corner *)
  ] (* par *)
```

as the single step:

```
doPar[padarray[i,j]:=padarray[K,j],
                    [i,1,K],[j,1,N+2*K]]; (* upper edge *)
```

We similarly optimize the code for filling the lower edge.

On the CM2, copying from one array to another is cheap only if the copied array has the same size and alignment in memory as the target. When the source is smaller than the destination, changing the alignment, the communication costs are high (roughly 100 times slower than the aligned case!). We can do little about the cost of copying originalarray. However, we need not suffer as great a cost when filling the east and west edges; we can take advantage of the fact that an aligned copy of the east edge of the original array is present in the target array, and copy that instead. This optimization requires that we add additional computation-ordering constraints ($R_{serialize}$) to ensure that copy-original-to-center occurs before filling the east or west edges. Having accomplished that, we can rewrite:

```
doPar[padarray[i,j]:=originalarray[i-K+1,1],
                     [i,K+1,K+N],[j,1,K]]; (* West side *)
```

as:

```
doPar[padarray[i,j]:=padarray[i,K+1],
                     [i,K+1,K+N],[j,1,K]]; (* West side *)
```

Again, we can do the same for the east side, producing a computation in the form shown by Figure 6.

A final equivalence simplifies a set of parallel computations, each of which is connected to all of their descendants, into a simple sequence:

```
seq[  N:=AxisSize(originalarray);
      allocate(padarray,N+2*k,N+2*k); (* creates storage for padarray *)
      doPar[padarray[i,j]:=originalarray[i-K+1,j-K+1],
                          [i,K+1,K+N],[j,K+1,K+N]]; (* middle *)
      par[doPar[padarray[i,j]:=padarray[i,K+1],
                              [i,K+1,K+N],[j,1,K]]; (* West side *)
          doPar[padarray[i,j]:=padarray[i,K+N],
                              [i,K+1,K+N],[j,K+N+1,N+2*K]]; (* East side *)
         ];
      par[doPar[padarray[i,j]:=padarray[K+1,j],
                      [i,1,K],[j,1,N+2*K]]; (* upper edge *)
          doPar[padarray[i,j]:=padarray[K+N,j],
                       [i,K+N+1,N+2*K],[j,1,N+2*K]]; (* lower edge *)
         ];
      padarray] (* seq *)
```

At the present time, SINAPSE represents the Pad component in essentially this form, rather than refining it from a more abstract description.

At final code-generation time, we generate code in any order consistent with the partial order over the computations and produce the following CM2 Fortran 90 code:

```
C     copy original array
      padarray(K+1:K+N,K+1:K+N)=mu(1:N,1:N)
C     Fill West edge
      padarray(K+1:K+N,1:K)=SPREAD(padarray(K+1:K+N,K+1),2)
C     Fill East edge
      padarray(K+1:K+N,K+N+1:N+2*K)=SPREAD(padarray(K+1:K+N,K+N,2)
C     Fill upper boundary
      padarray(1:K,1:N)=SPREAD(padarray(K+1,1:N),1)
C     Fill lower boundary
      padarray(K+N+1:N+2*K,1:N)=SPREAD(padarray(K+N,1:N),1)
```

It is interesting to compare this to the original hand-generated code. At no point must we rediscover the parallelism from a complex, highly optimized target language code, as done by conventional compilers. The same efficiency has been achieved from an abstract specification that could be used to generate code for

multiple architectures and languages. On a MIMD machine with low communication costs, we could assign one processor per result-array element, and specialize the **case** statement for that processor at synthesis time, providing effectively unit time execution of the padding operation. For a high-communication cost MIMD machine, we might refine the original component into 9 parallel tasks with no shared storage, and a final assembly step.

Since we avoid the discovery process, we also avoid the requirement to harness the often necessary problem domain knowledge to aid this process. Conventional compilers do not have this knowledge, and thus the application programmer must be somehow brought into the process, making compilation partly manual.

7 Lessons

For the CM2, a good strategy for generating code seems to be:

- Represent computations with functional code fragments expressing data parallelism over regions (this should allow us to target other parallel architectures as well).
- Convert the functional fragments to side-effecting fragments over the same regions.
- Reduce operation count by merging parallel data-parallel operations on adjacent regions.
- Reduce communications cost by aligning data.

For data-parallel architectures with regular communication topologies, misaligned data in operations can be very expensive. A model of the communication costs would help to focus attention of the synthesis system on points needing optimization. The actual optimization can be accomplished by copying to an aligned array (creating a singly-assigned temporary variable) and allowing code motion to move the copy step to a point where the copy is only evaluated once.

8 Conclusions

We have found tree-structured representations of parallelism to be overly constraining, and are moving towards representations including partial orders. Such representations will allow us to both directly encode domain-specific components with maximal parallelism, and hence enable us to perform optimization and resource assignment based on the potential parallelism.

SINAPSE is also being enhanced in several other areas. More detailed knowledge about problem domains such as 3D ultrasonic wave propagation is being added. Solution techniques such as finite-element methods as alternatives to finite differencing are contemplated. We are currently adding programming knowledge about multiple target languages. We hope to eventually generate production quality modeling programs for parallel machines.

References

[AG82] Arvind and Kim P. Goestelow. The U-Interpreter. *Computer*, pages 42–49, February 1982.

[BKK+89] Vasanth Balasundaram, Ken Kennedy, Ulrich Kremer, Kathryn McKinley, and Jaspal Subhlok. The ParaScope Editor: An Interactive Parallel Programming Tool. In *Proceedings of Supercomputing '89*, pages 540–550. ACM Press, November 1989. ACM Order Number 415892.

[Cor90] Pacific-Sierra Research Corporation. *The MIMDizer User's Guide*. Pacific-Sierra Research Corporation, 12340 Santa Monica Blvd, Los Angeles, CA 90025, 1990.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[HS86] W. Daniel Hillis and Guy L. Steele. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1184, December 1986.

[IO81] Moshe Israeli and Steven Orszag. Approximation of Radiation Boundary Conditions. *Journal of Computational Physics*, 41:115–135, 1981.

[KDMW90] Elaine Kant, François Daube, William MacGregor, and Joseph Wald. Automated Synthesis of Finite Difference Programs. In *Symbolic Computations and Their Impact on Mechanics, PVP-Volume 205*. The American Society of Mechanical Engineers 1990, New York, NY, 1990. ISBN 0-791800598-0.

[KDMW91] Elaine Kant, François Daube, William MacGregor, and Joseph Wald. Scientific Programming by Automated Synthesis. In M. Lowry and R. McCartney, editors, *Automating Software Design*. AAAI Press, 1991. To appear.

[SMD+89] K. Sridharan, M. McShea, C. Denton, B.Eventoff, J. C. Browne, P. Newton, M. Ellis, G. Grossbard, T. Wise, and D. Clemmer. An Environment for Parallel Structuring of Fortran Programs. In E.C. Plachy and P.M. Kogge, editors, *Proceedings of 1989 International Conference on Parallel Processing*, pages 98–106, 215 Wagner Building, University Park, PA 16802, August 1989. The Penn State Press.

[WBS+91] John Werth, James C. Browne, Steve Sobek, T. J. Lee, Peter Newton, and Ravi Jain. The Interaction of the Formal and the Practical in Parallel Programming Environment Development: CODE. Technical Report TR-91-09, Department of Computer Science, University of Texas at Austin, April 1991.

[Wol91] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addision-Wesley Publishing Company, Inc., Reading, Massachusetts, 1991. Second Edition.

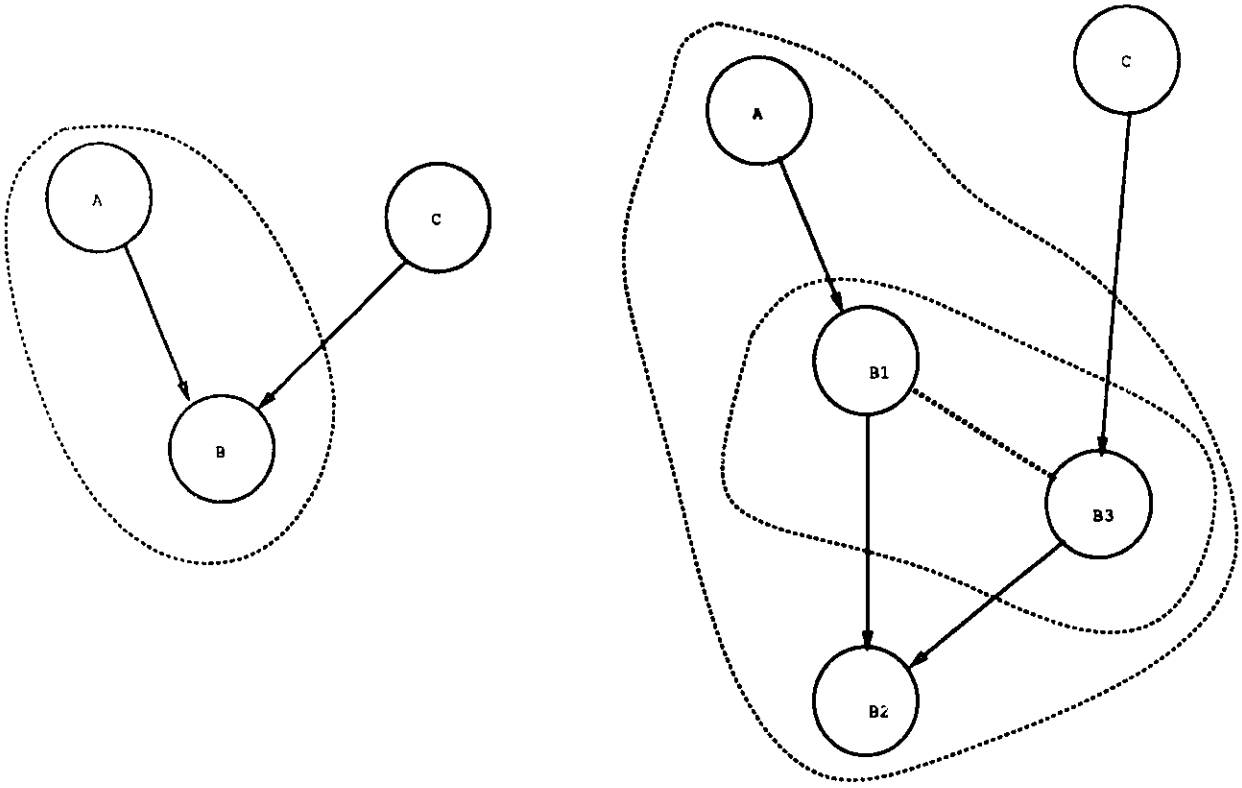


Figure 2: Refining node B

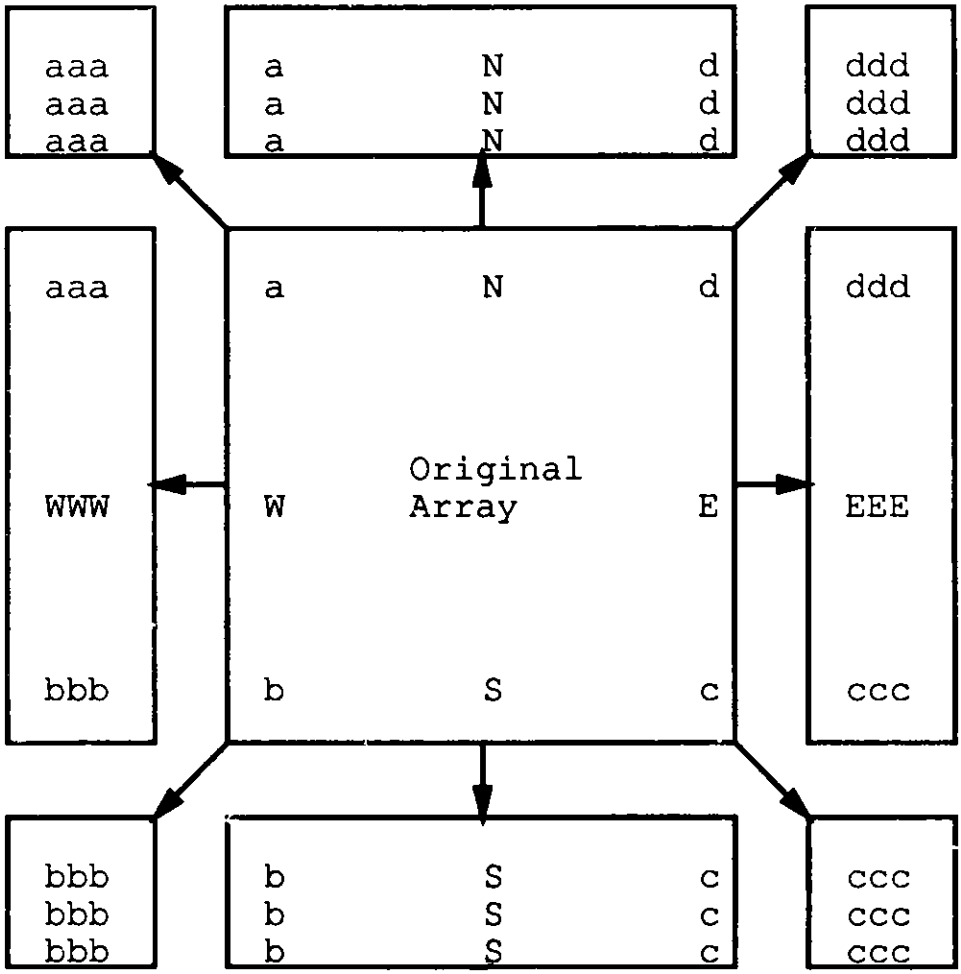
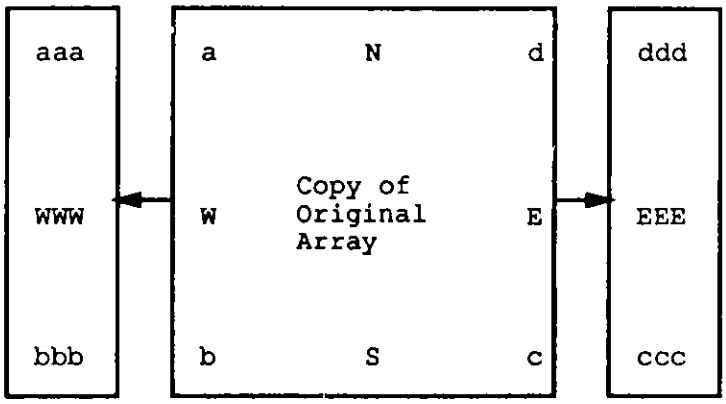
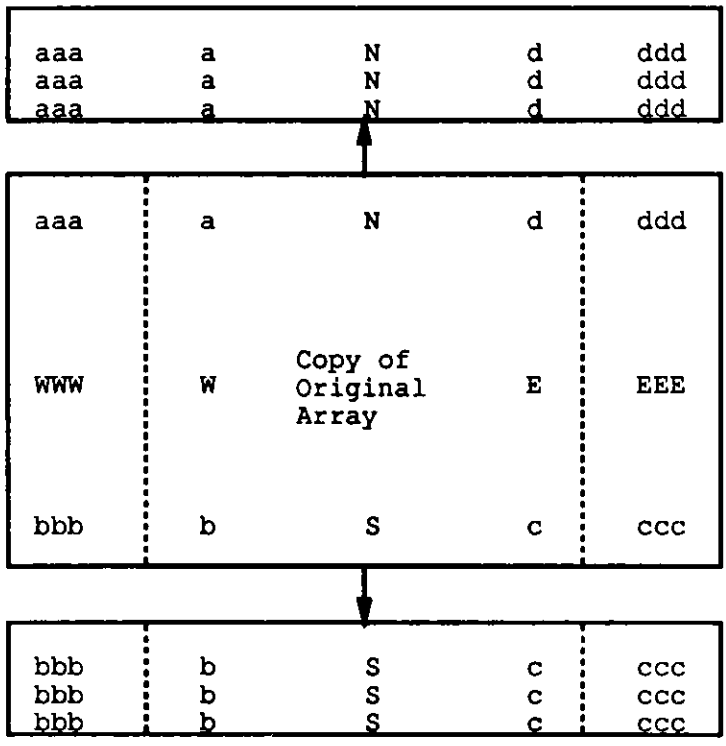


Figure 3: Padding an array by filling new boundaries with copies of edges



Stage 1



Stage 2

Figure 4: Padding operation optimized for Connection Machine

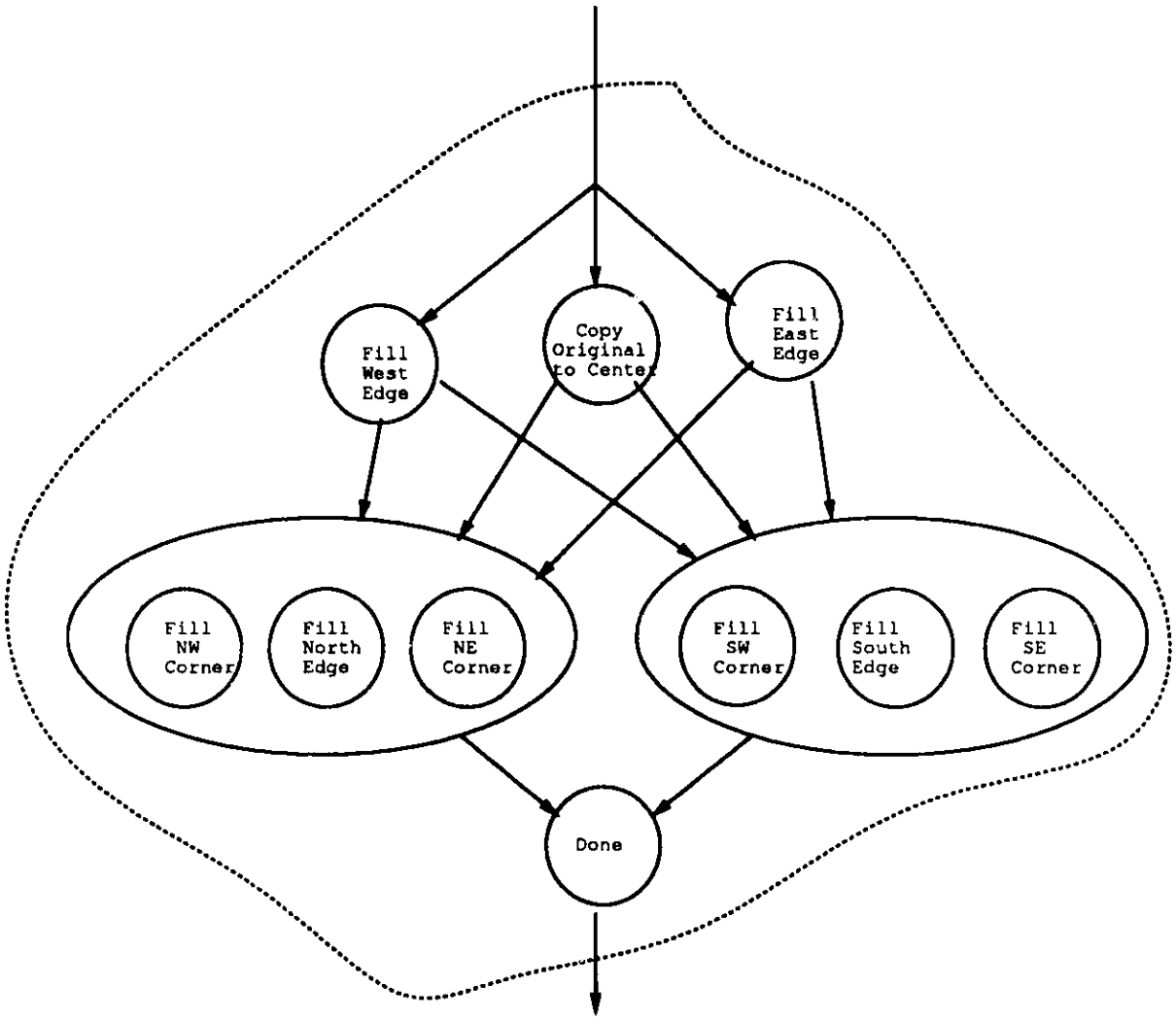


Figure 5: Refining parallel padding

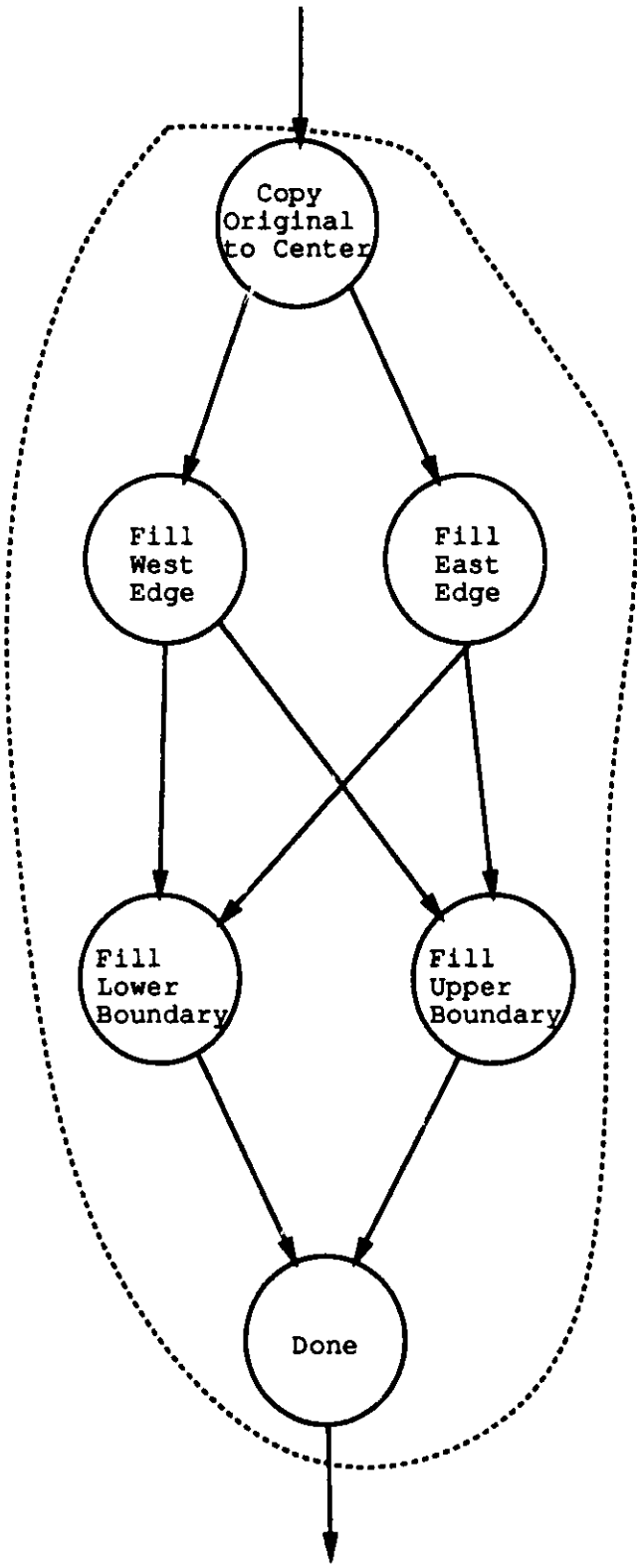


Figure 6: Final representation of padding for data parallel machine

Applying Abstract Interpretation to Indentify Vectorizable Numerical Code in Logic Programs [1]

Arvind K. Bansal[2] and Dilip S. Poduval
Department of Mathematics and Computer Science
Kent State University
Kent, OH 44242, USA
E-mail: arvind@mcs.kent.edu and poduval@mcs.kent.edu

The solution of real world problems require the efficient integration of both symbolic and quantitative computation. In recent years, logic programming paradigm has become quite popular for symbolic computation due to its nondeterministic and declarative style of programming which supports alternate solution, and the use of inherent polymorphism (multiple possibly infinite) which allows same piece of code to be used for different data types. However, the current implementation is slow, due to the extensive use of recursion, sequential data structures such as lists needed to support the declarative style of programming, and the lack of explicit declaration of monomorphic (single) types resulting into run time overhead of memory allocation. Current efforts to improve the run time efficiency falls into two categories as follows:

1. exploiting inherent control parallelism - AND parallelism, OR-parallelism, and stream parallelism - in AND-OR tree computation model which uses concurrent spawning of processes and their synchronization.

2. exploiting data parallelism on associative supercomputers to handle logic programs with large knowledge bases, and

3. Optimizing compilers based upon global data flow analysis

Approaches to incorporate parallelism in logic programs can be broadly described as follows:

1. User declared parallelism and synchronization information suitable for low level parallel logic programming etc., and

2. User transparent incorporation and integration of parallelism.

3. Compile-time global data-flow analysis to detect AND, OR, and stream parallelism in a program and to perform different types of optimizations such as identification of deterministic code and to transform different class of programs to integrate all three types of parallelism.

We are interested in the last model which forms the basis of a parallelizing and optimizing compiler in logic programs. Although parallelizing compiler seems quite promising, the execution speed of vectorizable numerical code with large data-size is much faster (1 to 2 billion floating point operations per second) on vector supercomputers such as pipelined vector supercomputers

[1]Supported in part by NSF equipment grant no. CDA 8820390
[2]Future Correspondence

or massive parallel SIMD computers, compared to 50 - 100 KLIPS (50 to 100 thousand logical inferences per second) for the current implementation of parallel logic programs. In imperative languages such as Fortran, the efficiency of the parallelizing and vectorizing compilers for scientific numerical computation has been successfully demonstrated.

A major part of the vectorizable numerical code is given by definite iteration because same set of statements are executed for every element of one or more sequences. A simple example of vectorizable numerical code (written in Fortran like language) to add two matrices is given in Example 1.

Example 1:

do 20 *I = 1* to *10*

*20 C(I) = A(I) + B(2*I + 3)*

Note that the value of the index used to access the elements in the vector *B*, increments periodically with a constant offset of 2 in each iteration step and the initial offset is 3.

Vectorization of such iterative programs replaces the innermost loop by a vector operation. In the vector notation M..N:P, M stands for the lower bound of a vector-subrange, N stands for the upper bound of the subrange, and P stands for the constant-offset in the index value to access the vector element in next step. The corresponding vector code for the above program is *C(1..10:1) = A(1..10:1) + B(5..23:2)*.

A major issue in identifying vectrizable code is the sequentiality caused due to the lack of available values for a variable, or the change of the value of a variable in the previous steps. In both the cases, the following statements have to wait for the values from the previous statements. While the absence of a value for a variable is purely a *synchronization issue*, the sequentiality caused due to change of value is due to destructive nature of variables in imperative languages. In logic programs, the absence of destructive nature of variables avoids the sequentiality caused due to the later restriction. The only sequentiality is caused due to dependence caused due to nonavailability of a value which has been identified using compile-time *producer-consumer relationship* - the first occurrence of a variable produces a variable and following occurrences consume variables - analysis. In contrast to imperative languages, the problem of vectorization in logic programs is quite different due to the presence of nondeterminism, the use of lists to simulate vectors, the lack of explicit monomorphic type declaration, the lack of support for iterative constructs, and the lack of destructive variables.

An approach to develop efficient parallelizing compiler which integrates symbolic and numeric computing under the framework of logic programming will incorporate

1. applying abstract interpretation to identify type information of the variables in various predicates,

2. applying abstract interpretation to identify vectorizable numerical code,

3. applying abstract interpretation to identify producer-consumer relationship necessary to derive data dependency,

4. transforming non-vectorizable numerical code using tail recursive programs to iterative programs which can easily be transformed to vectorizable code,

5. transform vectorizable numerical domain in logic programs to be efficiently executable code on vector supercomputers,

6. identifying deterministic code for code optimization by removing the overhead of handling multiple environment caused by multiple clauses [3].

7. identifying different types of parallelism in a logic program,

8. developing parallelizing compiler exploiting AND, OR, and stream parallelism in symbolic domain. This can be done using program transformation to existing model, or compiling the symbolic domain and non-vectorizable domain to an extended variation of warren abstract machine.

9. developing interface for parallelizing compiler in symbolic domain and vector codes on vector supercomputers

10. Extend warren abstract machine to handle non-vectorizable iteration, derived type information, along with AND, OR, and stream parallelism.

In this paper, we describe an application of abstract interpretation to identify the vectorizable numerical code. This *vector analysis* scheme derives definite iteration, the information about bound and indices, identification of vectors simulated by lists or functors, and the derivation of vector-size. The vector analysis scheme is based upon extending abstract domain from type domain to the abstract domain as vector domain. Vector domain is a superset of type domain and includes vector related information along with type related information. However, we differentiate between the two different modules, namely abstract interpretation in type domain and abstract interpretation in vector domain since the output of type domain is also used for *producer-consumer analysis*, *identifying different types of parallelism*, and identifying deterministic code - codes which have no alternative solution. The vector analysis scheme has five components, namely, generalization, vector unification, vector summarization, concretization, and iteration analysis. The first four components are used to propogate and collect the information in the vector domain and the last component uses solving a system of linear equations to derive the vector-size information, and the identification and disambiguation of index and bounds for iterative constructs simulated using tail recursive programs. We describe various schemes and explain the algorithms. We also discuss the current issues to integrate AND, OR, and stream parallelism with vectorization.

[3]Multiple environments are handled implicitly (one at a time) in Prolog by backtracking and unbinding, while, they are explicitly handled in OR-parallel implementations

Data Locality

Monica S. Lam
Computer Systems Laboratory
Stanford University, CA 94305

Previous research on parallelizing compilers for sequential imperative programming languages concentrated on the extraction of parallelism. Recent results in the area indicate that the extraction of parallelism is only the first step. The discovery of a large amount of parallelism does not necessarily translate to a gain in performance because the overhead in synchronization and communication can render parallelization ineffective. Improving data locality, thus reducing the communication overhead, will become even more important as processor speeds continue to increase much faster than communication and memory access rates. The study of parallelism must therefore be coupled with the study of locality.

While standard scalar optimizations aim to reduce the total instruction count in a program, locality optimizations rearrange the computation to reduce the cost of data accesses by taking better advantage of the memory hierarchy. We take a three-tiered approach to the problem: First, we try to group operations that use the same block of data as a unit of computation allocated to a processor. In this way, the cost of fetching the data is amortized across all the operations using the data. If this technique does not reduce the communication adequately, we try to allocate the computation and data to processors in a way that minimizes interprocessor communication. Lastly, if communication is unavoidable, we try to hide the latency of the communication by overlapping the data fetches with computation on other data.

We have been focusing on two different computation domains which require different techniques in gathering the information necessary for locality optimizations. They are dense matrix computations where a fully automatic compiler approach is feasible, and coarse-grain tasks where linguistic support is necessary.

1 Dense Matrix Computations

Our data analysis in the domain of dense matrix computations focuses on those array references whose indices can be expressed as affine functions of the loop indices. We have developed a reuse analyzer that identifies those iterations that use the same data[4]. The idea is based on finding the kernel of a matrix constructed from the index functions of an array access. This analysis provides the information useful for all the three locality optimizations described above. We have succeeded in using the information to block computations automatically, we are currently investigating the topics of better computation placement and prefetching.

To successfully convert codes in practice into their blocked version, we need to combine the basic blocking transform with other loop restructuring transforms such as loop permutation, reversal and skewing. We model these transformations of permutation, reversal and skewing and their combinations

This research was supported in part by DARPA contract N00014-87-K-0828.

as unimodular matrix transforms[5]. This approach applies not only to code whose dependences are representable as distance vectors, but also to the more general domain of direction vectors as well. This formulation reduces the locality optimization problem to finding the best matrix transform that exploits the reuse discovered in our reuse analysis.

Our algorithm can automatically block codes such as matrix multiplication, a successive over-relaxation (SOR) code, LU factorization without pivoting and Givens QR factorization. Performance evaluation reveals that blocking can improve uniprocessor workstations by a factor of 3 to 4; its impact on multiprocessor is even more significant as it reduces memory contention, permitting a near linear speed up on multiprocessor systems.

2 Coarse-Grain Parallelism

The high-level data reuse pattern, that is necessary to exploit coarse-grain parallelism effectively, is hard to extract automatically from a program. Fortunately, this level of information is well understood by the programmer of the application. We need only to provide linguistic support so that the programmer can easily convey the information to the optimizer. Existing imperative parallel programming languages generally require the programmer to express the parallelism in terms of low level control. Not only is the programming task difficult and error-prone, it is impossible to extract the high level data usage information from the program.

We have developed a parallel programming language called Jade that allows a data-oriented expression of parallelism while fully supporting the imperative programming paradigm[1, 2, 3]. Starting with a sequential program, a programmer simply augments those sections of code to be parallelized with side effect information. The Jade system automatically infers from the side effect information the allowable parallelism between these sections of code, which are also known as tasks. The Jade system finds not just static parallelism but also parallelism that can only be derived at run time. Using Jade can significantly reduce the time and effort required to develop a parallel version of an imperative application with serial semantics. Jade has been implemented as extensions to C, FORTRAN, and C++, and currently runs on the Encore Multimax, Silicon Graphics IRIS 4D/240S, and the Stanford DASH multiprocessors.

The responsibility of blocking the computation to promote data reuse falls on the programmer. Jade allows the programmer to express the side effects of an arbitrary unit of computation in terms of user-defined objects, rather than memory locations on individual read/write operations. This support of abstraction enables the programmer to use his application knowledge in choosing the right granularity of synchronization. The Jade language allows the programmer to express the information simply and directly. The programmer need not reduce the information down to low level control constructs; the system automatically coordinates the hardware resources to perform the computation correctly and concurrently. By hiding the low level details from the programmer, the program is easier to write and it is more portable. Moreover, with the side effect information specified by the programmer, the system can further optimize the program by better data and computation placement and data prefetching.

Jade is designed to be compatible with compiler optimizations. Since the language does not have any implicit communication operators, a compiler can optimize individual tasks as if they are simple sequential codes. Automatic parallelizing techniques, such as blocking, can be applied to the more regular parts of the computation, thus enabling the programmer to concentrate on the more application specific form of parallelism. By combining both language technology and compiler optimizations that focus on data locality optimizations, we hope to develop a system that can exploit parallelism in a wide

array of applications.

The research described in this talk is performed jointly with Michael Wolf, Martin Rinard, Daniel Scales and Jennifer Anderson.

References

[1] M. S. Lam and M. C. Rinard. Coarse-grain parallel programming in Jade. In *Proc. Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991.

[2] M. C. Rinard and M. S. Lam. Semantic foundations of Jade. In *Proc. 19th Annual ACM Symposium on Principles of Programming Languages*, January 1992.

[3] D. J. Scales, M. C. Rinard, M. S. Lam, and J. M. Anderson. Hierarchical concurrency in Jade. In *Fourth Workshop on Languages and Compilers for Parallel Computing*, August 1991.

[4] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[5] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, October 1991.

Compiling FP for Data-Parallel Systems (Extended Abstract)

Clifford Walinsky* Deb Banerjee†

Department of Mathematics & Computer Science
Dartmouth College
Hanover, NH 03755

Abstract

In data-parallel programming, operations are performed simultaneously on all elements of large data structures. Backus's FP functional language promotes this view. FP provides a large set of data rearrangemenet primitives, and a useful set of functional combining forms that are applied to entire data structures. We present an overview of an FP compiler that generates programs capable of exploiting data-parallelism. The FP compiler determines the effects of data rearrangement functions at compile-time, thereby avoiding creation of large intermediate data structures, and reducing communication overhead. FP and its compiler are formally specified, reducing ambiguity concerning constructs of the language and results of the compiler.

1 Introduction

Recent developments in computer design have made it possible to expoit a particular form of massive parallelism: data-parallelism [11]. Examples of data-parallel systems include the Connection Machine©[20], the ICL/DAP, the MPP [4], Blitzen [6], and ILIAC-IV.

In data-parallel systems, distinct elements of data structures are stored in distinct processors. Software data structures are then aligned most naturally to the physical configuration of processors, which is typically a multi-dimensional grid. Because processors operate synchronously, operations are performed on entire data structures. Data-parallel programming style is very different from conventional programming. In conventional languages, iteration and recursion are used to perform operations on individual elements of data structures. In data-parallel programming, operations to combine and rearrange data are applied to entire data structures, making iteration and recursion less important.

De-emphasis of iteration and recursion makes realization of efficient parallel implementations much easier. The programs in Figure 1, written in a conventional notation employing iteration, both sum all elements of a vector `V`. Sophisticated analysis is required to determine that simultaneous evaluation of the body of the first program's loop will not yield significant performance improvement (pipelined processing is more appropriate), while simultaneous evaluation is appropriate for the second program.

*clifford.walinsky@dartmouth.edu
†deb.banerjee@dartmouth.edu

```
Summation Program 1          Summation Program 2
S := 0;                      for i in 1..lg(|V|) do
for i in 0..|V|-1 do             for j in 0..|V|-1 do
    S := S + V[i]                    if j mod 2^i = 0 then
                                         V[j] := V[j] + V[j+2^(i-1)]
```

Figure 1: Summation Programs

Limiting iteration and recursion makes data-rearrangement much more important. For example, to sum corresponding elements of two vectors, the data-parallel style dictates the following steps.

1. Corresponding elements of the vectors are first paired so that corresponding elements reside within the local memory of a single processor.

2. Next, every processor adds the two elements stored in its local memory.

The first step of data-parallel vector addition involves data-rearrangement. Using FP, it has been our experience that the number of operations that rearrange data far exceeds the number needed to combine values. Without a methodology for optimizing data movement and reducing the number of intermediate data structures produced from data rearrangement, compiled programs would perform very poorly. Wadler also describes techniques for eliminating intermediate data structures in functional programs [21][22]. His optimizations are adapted for linked data structures, such as lists, and require enhancement to take advantage of vector indexing operations that occur in data-parallel programming.

A particular class of data rearrangement functions is amenable to compiler analysis and optimization. A *routing* function copies (a subset of) values from input to output in a data-independent manner. That is, the actions of a routing function are entirely dependent on the shape of the input, rather than the particular value of the input. Reversal of the order of elements in a vector is a routing function, because the ultimate destinations of elements in the vector are unaffected by their values. By contrast, a function that sorts elements of a vector is not a routing function, since destinations of elements are entirely dependent on values. The class of expressible routing functions is quite large, and may exceed the requirements of most applications.

Inherent resource limitations of data-parallel architectures also impose additional restrictions on iteration and recursion. In general these constructs are implemented most efficiently when processors are able to conduct different activities simultaneously. However, in functional programming systems, the activities of processors may involve complex stack manipulation, garbage collection, and process migration tasks. These tasks would easily overwhelm the relatively modest computational power and small local memory space of processors in data-parallel systems. In the CM-2, for example, each processor contains a 1-bit ALU, and can access at most 8K bytes of local memory.

Realizing the limitations and opportunities of data-parallel systems, the FP language [3] seems well-suited for compilation. FP has a large set of data rearrangement primitives, and a useful set of functional combining forms that encourage a style of programming where operations are applied to entire data structures. Currently, we have implemented a prototype system that translates FP to CM-Fortran [19]. Compilation employs "structure inference" to determine the form of inputs and

outputs to all expressions [23]. We apply optimizations during compilation to limit the number of intermediate data structures and amount of data movement incurred from data rearrangement [8].

Since we are currently translating FP programs to an imperative language, and since we impose many restrictions on FP programs, it may seem that our approach dilutes many of the advantages of functional programming. To the contrary, even with the restrictions we impose, FP programming provides important benefits. First, the combining forms of FP enable easy composition of program units. Our structure inference system helps to determine if function composition is meaningful, for failure of structure inference implies the presence of a program error. While composition of FP programs is extremely natural, composition of imperative language programs is relatively difficult, because name conflicts may arise, and because data dependencies may not be satisfied. To see this, consider the difficulties constructing a program to sum two vectors, `A` and `B`, by composing the two summation programs in Figure 1. The composition requires many name changes. If complex data dependencies were more prevalent between the two programs, the composition would be even harder to perform.

A second advantage in favor of FP is demonstrated by Backus [3]. FP has a well-developed algebra, enabling program improvements to proceed automatically and reliably. By contrast, improvements to imperative language programs, especially those involving global analysis, are often difficult to perform correctly.

FP is difficult to compile because programs contain no description of the data structures they are manipulating. Structure inference has been a key discovery that enables us to infer descriptions of inputs and outputs of all functions in programs. This inference technique is described in Section 2.1. Structure inference produces type information and can statically compute vector lengths. Since vector length determination is undecidable in general, we have had to restrict the class of programs suitable for structure inference.

Inferred structure information is utilized by a formally described compiler that produces low-level language programs. The compiler is described in Section 2. Encouraging preliminary performance results for the compiler are examined in Section 3.

We assume that the reader has no knowledge of the FP language. The next subsection provides an overview of FP. The final subsection of this section reviews other implementations of parallel functional programming languages. As this paper appears in a logic programming forum, we gratefully acknowledge the important contribution of Prolog in making our ideas realizable. The format of this paper is very much an extended abstract; we intentionally eliminate much detail. The reader is encouraged to examine other references for greater detail [23][8].

1.1 The Data-Parallel FP Dialect

John Backus described FP in his Turing Award lecture in 1978. Concurrent with publication of FP, Magó [15] described a massively parallel computer system, called the FPM, for evaluating FP. Development of the FPM was never completed.

An FP program is a finite collection of (possibly mutually recursive) function definitions. Every function definition describes a partial function mapping over FP data objects. Function definitions consist of primitive functions and functionals that combine functions. All functions are strict—for all functions f, $f(\perp) = \perp$, where $\perp$ denotes the error value.

Certain subexpressions within FP function definitions are designated for parallel evaluation. Currently, these subexpressions may not be recursively defined, and may not contain iteration. Parallel FP functions map over parallel FP objects, which may be scalar values, tuples (fixed-

length sequences of objects), or vectors (finite but unknown-length sequences of similarly structured objects). Vectors and tuples may be nested, producing for example vectors of tuples. A vector of n elements—$x_1, \ldots, x_n$—is denoted $[x_1, \ldots, x_n]$. A tuple of n elements—$x_1, \ldots, x_n$—is denoted $\langle x_1, \ldots, x_n \rangle$.

Table 1 describes the set of primitive parallel FP functions used in examples of this paper. Functions are combined with five special functionals—higher-order functions that take other functions as arguments and return functions over parallel FP data objects. The functionals are defined in Table 2. Due to data dependencies, Backus's original definition of the insert functional requires some multiple of n parallel time steps for evaluation on an n-element vector, even with an unlimited number of processors. The pairwise insert, by contrast, requires only $\lceil \lg n \rceil$ parallel time steps when applied to n-element vectors.

Function Name	Function Description
Primitive Computational Functions	
Addition(+)	$+(x) = \begin{cases} y_1 + y_2, & \text{if } x = \langle y_1, y_2 \rangle; \\ \bot, & \text{otherwise.} \end{cases}$
Multiplication(×)	$\times(x) = \begin{cases} y_1 \times y_2, & \text{if } x = \langle y_1, y_2 \rangle; \\ \bot, & \text{otherwise.} \end{cases}$
Primitive Routing Functions	
Tuple Selection(i_n)	$i_n(x) = \begin{cases} y_i, & \text{if } x = \langle y_1, \ldots, y_n \rangle \text{ and } 1 \leq i \leq n; \\ \bot, & \text{otherwise.} \end{cases}$
Transposition(**trans**)	$\mathbf{trans}(x) = \begin{cases} [[x_{1,1}, \ldots, x_{m,1}], \ldots, [x_{1,n}, \ldots, x_{m,n}]], \\ \quad \text{if } x = [[x_{1,1}, \ldots, x_{1,n}], \ldots, \\ \quad\quad [x_{m,1}, \ldots, x_{m,n}]]; \\ \bot, \quad \text{otherwise.} \end{cases}$
Distribute-left(**distl**)	$\mathbf{distl}(x) = \begin{cases} [\langle x, y_1 \rangle, \ldots, \langle x, y_n \rangle], \\ \quad \text{if } x = \langle x, [y_1, \ldots, y_n] \rangle; \\ \bot, \quad \text{otherwise.} \end{cases}$
Distribute-right(**distr**)	$\mathbf{distr}(x) = \begin{cases} [\langle x_1, y \rangle, \ldots, \langle x_n, y \rangle], \\ \quad \text{if } x = \langle [x_1, \ldots, x_n], y \rangle; \\ \bot, \quad \text{otherwise.} \end{cases}$
Pairing(**pair**)	$\mathbf{pair}(x) = \begin{cases} [\langle x_1, y_1 \rangle, \ldots, \langle x_n, y_n \rangle], \\ \quad \text{if } x = \langle [x_1, \ldots, x_n], [y_1, \ldots, y_n] \rangle; \\ \bot, \quad \text{otherwise.} \end{cases}$

Table 1: FP Primitives

Functional Name	Functional Description
Composition($f \circ g$)	$f \circ g(x) = f(g(x))$
Apply-all(αf)	$\alpha f(x) = \begin{cases} [f(x_1), \ldots, f(x_n)], & \text{if } x = [x_1, \ldots, x_n]; \\ \perp, & \text{otherwise.} \end{cases}$
Tupling($[f_1, \ldots, f_n]$)	$[f_1, \ldots, f_n](x) = \begin{cases} \langle f_1(x), \ldots f_n(x) \rangle, & \text{if } f_i(x) \neq \perp; \\ \perp, & \text{otherwise.} \end{cases}$
Pairwise Insert($/f$)	$/f(x) = \begin{cases} y, & \text{if } x = [y]; \\ f(\langle /f([x_1, \ldots, x_n]), /f([x_{n+1}, \ldots, x_{2n})] \rangle), & \\ \quad \text{if } x = [x_1, \ldots, x_{2n}]; & \\ \perp, & \text{otherwise.} \end{cases}$

Table 2: FP Functionals

1.2 FP Programming Examples

Backus's original matrix-multiply program is presented below.

```
def MM = αα(IP) ∘ PairUp.
def IP = /(+) ∘ α(×) ∘ pair.
def PairUp = α(distl) ∘ distr ∘ [1₂,transo2₂].
```

Function **MM** is provided a tuple of two matrices, each represented by a vector of vectors. The PairUp function forms a matrix of pairs of every row of the first matrix with every column of the second. The $\alpha\alpha$(IP) function then applies the inner-product function **IP** to each row-column pair.

We still need to show that the **MM** program specifies parallel activities that a compiler can exploit. If literally translated, the **PairUp** function could produce a large number of intermediate data structures. However, creation of these intermediate data structures is avoided with our compilation techniques. The **IP** function is evaluated simultaneously on a matrix of row-column pairs. Since each product is computed essentially in constant time, and the summation of products is performed in time proportional to the length of each row, the entire program specifies a logarithmic time (in the length of each row of the first matrix) parallel procedure.

This function definition exemplifies the style of programs suitable for data-parallel evaluation. Routing functions (**PairUp** in this example) often dominate, because they are necessary for distributing data to the appropriate functional units for computational operations. Recursion and iteration are far less important than in other functional programming languages. Some of the benefits of these constructs are subsumed by the extensive set of routing functions provided with FP, and the apply-all and pairwise-insert functionals. Certainly, recursion and higher-order functions are important programming tools; however, many useful programs can be produced without them.

1.3 Related Work Parallelizing Functional Languages

Development of parallel functional programming languages and systems has been ongoing for many years. Most systems developed and proposed so far consist of heterogeneous (MIMD) processes.

Dataflow languages [16][18] expose parallelism at the instruction level—the finest grain possible. In contrast to data-parallel languages, instructions are executed asynchronously. The Id language [18] incorporates regular data structures similar to arrays, called I-structures. Typically a computational process is initiated at each element of an I-structure, expoiting a high degree of available parallelism. Our compiler also implements regular data structures (FP's sequences) as arrays, and can spawn processes at each array element.

The HDG (Highly Distributed Graph Reduction) system [7], ParAlfl [12], and various parallel Lisp and Scheme implementations [13] have language evaluators located at each processor. In contrast to dataflow systems, these systems exploit mainly course-grained parallelism. Due to the inherent resource limitations of data-parallel systems, data-parallel implementation of HDG appears to be impractical.

ParAlfl programs contain explicit statements directing the mapping of processes to physical processors. This mapping problem is substantially reduced in data-parallel systems because processes are always mapped to distinct virtual processors, and the mapping of virtual to physical processors is performed automatically according to problem size. Much work has been conducted in the development of CM-Fortran [19], determining proper physical layout of data to reduce inter-processor communication [14]. At this early stage in its development, the Connection Machine FP compiler generates CM-Fortran programs enabling it to obtain reasonably good layout of data structures.

Mou and Hudak have developed a data-parallel functional language, called Divacon [17]. Functions designated for parallel evaluation are evaluated in a divide-and-conquer manner. With two communication functions, mirror and correspondence, PDC schemas can be compiled into efficient code suitable for evaluation on data-parallel architectures. Additional communication primitives would make programs easier to comprehend and produce, but may complicate compilation.

Blelloch presents "scan primitives" to replace memory access operations in PRAM models, resulting in more realistic performance analysis [5]. Blelloch also demonstrates how scan can simplify algorithm description. Data-parallel FP implements the insert functional that is similar, algebraically, to the scan operation.

FP functions that exploit data-parallelism are defined over entire data structures. By contrast, languages such as Crystal [9] decompose problems by specifying values of individual computational elements with recursion equations. Similarities do exist, in that user-defined routing functions are specified with "access function mappings," which are similar to recursion equations.

Many researchers have already recognized the opportunities for optimization provided by routing functions. For example, APL compilers tag arrays with information about ravel and dimension order so that evaluation of routing functions can be avoided entirely in some cases [10].

2 Compilation

Our compiler operates in three phases. In the first phase, structure inference determines the structure of inputs and outputs of all functions designated for parallel evaluation. The second phase uses structure inference information to emit an intermediate-language program. In the final phase, standard dataflow optimizations are performed on intermediate-language programs to avoid redundant data movement and data structure creation.

2.1 Structure Inference

A *structure* is an abstraction of the form of a data structure. Function $\mathcal{M}$ describes the set of values denoted by a structure. Ground structures and $\mathcal{M}$ are inductively defined as follows.

- Scalar types **int**, **real**, and **bool** are *scalar structures.* If s is a scalar structure, $\mathcal{M}[\![s]\!]$ is the set of all values representable in scalar type s.
- For $n \geq 0$, if $s_1, \ldots, s_n$ are structures, $\text{tuple}_n(s_1, \ldots, s_n)$ is a *tuple structure.*

$$\mathcal{M}[\![\text{tuple}_n(s_1, \ldots, s_n)]\!] = \{\langle x_1, \ldots, x_n \rangle | x_i \in \mathcal{M}[\![s_i]\!], \text{for } 1 \leq i \leq n\}.$$

- If s is a structure, and n is a non-negative integer, $\text{array}(n, s)$ is an n-element *array structure.*

$$\mathcal{M}[\![\text{array}(n, s)]\!] = \{[x_1, \ldots, x_n] | x_i \in \mathcal{M}[\![s]\!], \text{for } 1 \leq i \leq n\}.$$

Notice that every structure denotes a nonempty set.

If s_1 and s_2 are ground structures, $s_1 \rightarrow s_2$ is a *structure mapping* denoting the set of all partial functions from the set of values $\mathcal{M}[\![s_1]\!]$ to the set of values $\mathcal{M}[\![s_2]\!]$.

We posit the existence of disjoint, denumerable sets of *length variables* and *structure variables.* A *non-ground* structure contains length variables (and more generally, length expressions) within array structures—rather than just constant lengths, and structure variables in addition to scalar structures. Structure s' is an *instance* of structure s if there is a binding θ of non-negative integers to length variables, and structures to structure variables, such that $s' = \theta(s)$. When s_1 and s_2 are non-ground structures, $s_1 \rightarrow s_2$ is a denotation for the following class of partial functions:

$$s_1 \rightarrow s_2 = \bigcup \{\mathcal{M}[\![s_1']\!] \rightarrow \mathcal{M}[\![s_2']\!] | s_1' \rightarrow s_2' \text{ is a ground instance of } s_1 \rightarrow s_2\}.$$

The purpose of structure inference is to deduce a most general structure mapping that characterizes each function within an FP program. The inference system possesses a set of axioms, one for each primitive function, and a set of inference rules, one for each functional. When a function f is inferred to belong to the class of functions $s_1 \rightarrow s_2$, we write $f : s_1 \rightarrow s_2$. Some of the axioms and inference rules used by the inference system are listed below.

Multiplication(×)

$$\text{tuple}_2(\textbf{real}, \textbf{real}) \rightarrow \textbf{real}$$

Pairing(pair)

$$\text{tuple}_2(\text{array}(n, \alpha), \text{array}(n, \beta)) \rightarrow \text{array}(n, \text{tuple}_2(\alpha, \beta))$$

Composition(∘)

$$\frac{f : \beta \rightarrow \gamma \quad g : \alpha \rightarrow \beta}{f \circ g : \alpha \rightarrow \gamma}$$

Apply-all(α)

$$\frac{f : \alpha \rightarrow \beta}{\alpha(f) : \text{array}(n, \alpha) \rightarrow \text{array}(n, \beta)}$$

Each inference rule is conditional and specifies how deductions are to be performed. In each rule, the conclusion below the line is derivable if the premise above the line can be demonstrated. The inference rule for composition will generally require syntactic unification of the input structure for f with the output structure for g.

Example 2.1 Using the inference rules above, the following deduction tableau can be created for the function $\alpha(\times) \circ \texttt{pair}$, which occurs within `MM` (page 5).

$$\frac{\dfrac{\times : \text{tuple}_2(\textbf{real}, \textbf{real}) \rightarrow \textbf{real}}{\alpha(\times) : \text{array}(n, \text{tuple}_2(\textbf{real}, \textbf{real})) \rightarrow \text{array}(n, \textbf{real})} \qquad \begin{array}{l}\texttt{pair} : \text{tuple}_2(\text{array}(n, \textbf{real}), \text{array}(n, \textbf{real})) \\ \qquad \rightarrow \text{arr}\_\text{y}(n, \text{tuple}_2(\textbf{real},\textbf{real}))\end{array}}{\begin{array}{c}\alpha(\times) \circ \texttt{pair} : \text{tuple}_2(\text{array}(n, \textbf{real}), \text{array}(n, \textbf{real})) \\ \rightarrow \text{array}(n, \textbf{real})\end{array}}$$

2.2 Compiler Specification

The compiler emits imperative intermediate-language programs. The intermediate language has been designed to be translatable to lower-level, machine-specific languages, and to be amenable for performing dataflow optimizations.

The only construct in the intermediate language uniquely suited for data parallelism is the `for all`-loop. This construct has the form below.

```
for all l ≤ i < u do
    <statements>
```

The syntactically enclosed <statements> are evaluated simultaneously on $u - l$ processors. Processors are arrayed within a space defined by nested `for all`-loops. Each processor within an n-dimensional space is assigned a unique n-tuple of indices $(i_1, i_2, \ldots, i_n)$, where each i_j lies within a fixed numeric range. To evaluate the above `for all`-loop within n enclosing loops, each processor is assigned a unique $n+1$-tuple of indices $(i_1, i_2, \ldots, i_n, i_{n+1})$, where $l \leq i_{n+1} < u$. At each processor, index variable i will be assigned the $n+1^{st}$ element of the processor's identifying tuple.

To associate program variable names with structures, the compiler makes use of "structure-name trees." A *structure-name tree* is a structure whose leaves (scalar structures) are program variables. A structure-name tree t' is a *variant* of a structure-name tree t if there is a substitution σ of new variable names for those appearing in t such that $t' = \sigma(t)$. For example, $\text{tuple}_2(A, \text{array}(n, B))$ is a structure-name tree, possessing a variant $\text{tuple}_2(C, \text{array}(n, D))$. We use a function *variant* to construct variants from structure-name trees.

The compiler makes extensive use of a syntactic function $\mathcal{D}$. $\mathcal{D}[\![(t_1, [\vec{i}]) := (t_2, [\vec{j}])]\!]$ produces a statement to copy a data structure described by a structure-name tree t_2 to a new data structure described by t_1, which must be a variant of t_2. The index expression lists $\vec{i}$ and $\vec{j}$ contain indexing information from enclosing `for all`-loops. $\mathcal{D}$ is defined recursively below, based on the possible structure forms.

- When A and B are variable names:

  $\mathcal{D}[\![(B,[\vec{i}]):=(A,[\vec{j}])]\!] =$
  $\quad B[\vec{i}]:=A[\vec{j}].$

- $\mathcal{D}[\![(\text{tuple}_n(t'_1,\ldots,t'_n),[\vec{i}]):=(\text{tuple}_n(t_1,\ldots,t_n),[\vec{j}])]\!] =$
  $\quad \mathcal{D}[\![(t'_1,[\vec{i}]):=(t_1,[\vec{j}])]\!];\cdots;\mathcal{D}[\![(t'_n,[\vec{i}]):=(t_n,[\vec{j}])]\!].$

- $\mathcal{D}[\![(\text{array}(n,t'),[\vec{i}]):=(\text{array}(n,t),[\vec{j}])]\!] =$
  $\quad$ **for all** $0 \le i < n$ **do**
  $\quad\quad \mathcal{D}[\![(t',[\vec{i},i]):=(t,[\vec{j},i])]\!].$

The compilation function, $\mathcal{C}[\![f : t]\!]\vec{i}$, produces an intermediate-language program equivalent to FP function f; t is the structure-name tree describing f's input; and $\vec{i}$ is a list of index variables produced from enclosing **for all**-loops. $\mathcal{C}$ also returns a new structure-name tree describing its output. $\mathcal{C}$ is defined recursively, based on the form of the FP function to compile. To reduce space, we list just a few of the rules defining $\mathcal{C}$, below.

- $\mathcal{C}[\![\texttt{pair} : \text{tuple}_2(\text{array}(n,t_1),\text{array}(n,t_2))]\!]\vec{i} =$
  $\quad\langle$ **for all** $0 \le i < n$ **do**
  $\quad\quad \mathcal{D}[\![(t'_1,[\vec{i},i]):=(t_1,[\vec{i},i])]\!];$
  $\quad\quad \mathcal{D}[\![(t'_2,[\vec{i},i]):=(t_2,[\vec{i},i])]\!],$
  $\quad \text{array}(n,\text{tuple}_2(t'_1,t'_2))\rangle,$

  where $t'_1 = \mathit{variant}(t_1)$ and $t'_2 = \mathit{variant}(t_2)$.

- $\mathcal{C}[\![\times : \text{tuple}_2(A,B)]\!]\vec{i} =$
  $\quad\langle C[\vec{i}]:=A[\vec{i}] \times B[\vec{i}],$
  $\quad C\rangle,$

  where C is a new variable name.

- $\mathcal{C}[\![f \circ g : t_1]\!]\vec{i} =$
  $\quad\langle S_g; S_f,$
  $\quad t_3\rangle,$

  where $\langle S_g, t_2\rangle = \mathcal{C}[\![g : t_1]\!]\vec{i}$, and $\langle S_f, t_3\rangle = \mathcal{C}[\![f : t_2]\!]\vec{i}$.

- $\mathcal{C}[\![\alpha(f) : \text{array}(n,t)]\!]\vec{i} =$
  $\quad\langle$ **for all** $0 \le i < n$ **do**
  $\quad\quad S_f,$
  $\quad \text{array}(n,t')\rangle,$

  where $\langle S_f, t'\rangle = \mathcal{C}[\![f,t]\!](\vec{i},i)$.

Example 2.2 We continue Example 2.1 by describing the intermediate-language program produced by compiling $\alpha(\times) \circ \texttt{pair}$, which is found within the **MM** program (page 5). Recall that the inferred input structure for this function is $\text{tuple}_2(\text{array}(n,\textbf{real}),\text{array}(n,\textbf{real}))$. We construct a structure-name tree from this input structure by replacing all scalar types with unique variable names. The initial list of index variables provided to function $\mathcal{C}$ is empty, denoted by ϵ. Compilation proceeds "bottom-up," according to the following steps.

1. $\mathcal{C}[\![\texttt{pair} : \text{tuple}_2(\text{array}(n, A), \text{array}(n, B))]\!]\epsilon =$
$\langle$ **for all** $0 \leq i < n$ **do**
$C[i] := A[i]$;
$D[i] := B[i]$,
$\text{array}(n, \text{tuple}_2(C, D))\rangle$

2. $\mathcal{C}[\![\times : \text{tuple}_2(C, D)]\!]j =$
$\langle E[j] := C[j] \times D[j],$
$E\rangle$

3. $\mathcal{C}[\![\alpha(\times) : \text{array}(n, \text{tuple}_2(C, D))]\!]\epsilon =$
$\langle$ **for all** $0 \leq j < n$ **do**
$E[j] := C[j] \times D[j]$,
$\text{array}(n, E)\rangle$

4. $\mathcal{C}[\![\alpha(\times) \circ \texttt{pair} : \text{tuple}_2(\text{array}(n, A), \text{array}(n, B))]\!]\epsilon =$
$\langle$ **for all** $0 \leq i < n$ **do**
$C[i] := A[i]$;
$D[i] := B[i]$;
for all $0 \leq j < n$ **do**
$E[j] := C[j] \times D[j]$,
$\text{array}(n, E)\rangle$

2.3 Intermediate-Program Improvements

While we have successfully translated FP to an imperative intermediate language, the resulting programs can be greatly improved. For example, the final program of Example 2.2 can easily be shortened to the following more condensed form.

for all $0 \leq i < n$ **do**
$E[i] := A[i] \times B[i]$

In this section we introduce a number of transformations on intermediate-language programs that can reduce their storage requirements and improve efficiency. These transformations are based on standard program improvement techniques. They have been specialized to FP's intermediate language. Because control structures of the intermediate language are quite simple, and programs are generally single-assignment, the transformations require only local analysis within straight-line segments of statements.

We first review some standard definitions [2], adapting them to the intermediate language. An assignment statement of the form $A[\vec{i}]$:= e *defines* variable A. If a variable B appears in the right-hand side of the assignment statement, the statement *uses* B. A definition of A *reaches* to a use of A (the definition and use must be distinct statements) if there is a straight-line statement sequence from the definition to the use, and there is no intervening definition of A. Note that imperative languages generally require dataflow graphs to determine reaching definitions.

Definition Propogation If a statement $A[\vec{i}]$:= e_1 is the only definition of A that reaches to a statement $B[\vec{j}]$:= e_2, then all occurrences of $A[\vec{i}]$ in e_2 can be replaced by e_1. Instances of definition propogation are easily detected. Any definition of a variable A that reaches to a use of

A must be the only use; the compiler will not emit intervening definitions of A. A more powerful definition propogation rule can also be defined to accomodate syntactic differences between the defined variable, $A[\vec{i}]$, and its use inside e_2.

Dead-Code Elimination If a program contains no uses of a variable A, and A is not an output of the program, all definitions of A may be removed. Typically, dead code will appear after definitions have been propogated.

Loop Fusion Loop fusion can create straight-line sequences of statements amenable to definition propogation and dead-code elimination out of adjacent **for all**-loops. Two **for all**-loops with identical bounds can be incorporated into a single loop [1], as follows. Suppose there are two adjacent statements within the intermediate-language program of the following form.

```
for all l ≤ i < u do
    S1;
for all l ≤ j < u do
    S2
```

When we let S_3 be the result of replacing index j by i throughout statement S_2, the following statement is equivalent to the above statements.

```
for all l ≤ i < u do
    S1;
    S3
```

The equivalence holds as long as index i occurs nowhere within statement S_2—a requirement that can be upheld quite easily by the FP compiler in its choice of index names. Refinements to this improvement rule can be devised to accomodate differing bounds on **for all**-loops.

The dataflow optimizations described here may seem *ad hoc*, providing very little indication about the achievable performance of compiled programs. We have also developed a formal description of routing functions, using a formalism we call "access function mappings." Composition of access function mappings eliminates redundant copying. The intermediate-language improvements of this section have been selected to produce the same effects as composition of access function mappings.

3 Performance Measurements

With the improvements mentioned in Section 2.3, program **MM** (page 5) has been compiled to the intermediate-language program in Figure 2. A cursory inspection should demonstrate the close similarity between this program and one written in a more conventional language. We view the results of this program and others as successful demonstrations of the compiler. Actual performance of this program corroborates these favorable results.

Performance of the compiled **MM** program running on the CM-2 is charted in Figure 3. Times presented in this graph (in milliseconds) are averages over a number of experiments. VP-ratios indicate problem size. For each vp-ratio 2^v, the input matrix dimensions are $2^6 \times 2^{1+v}$ and $2^{1+v} \times 2^6$. The program of Figure 2 has been slightly modified, replacing the **for**-loop that computes /+ with

```
(1)   for all 0 ≤ j < n do
(2)       for all 0 ≤ i < m do
(3)           for all 0 ≤ k < p do
(4)               H[j,i,k]:=A[j,k] × B[k,i];
(5)           for x:=1 to ⌈lg n⌉ do
(6)               for all 0 ≤ k < p do
(7)                   if k mod 2^x = 0 then
(8)                       H[j,i,k]:=H[j,i,k] + H[j,i,k + 2^(x-1)];
(9)           K[j,i]:=H[j,i,0];
```

Figure 2: Compiled Version of **MM**

a system library call. Multiplying 64×512 by 512×64 matrices on 8K processors (and 256 floating point processors), yields 13.4 MFLOPS (32-bit floating point multiply instructions per second). By comparison, the CM library routine for computing matrix multiply, performed on two 128×128 matrices (giving the same number of multiplications), yields performance of 36.5 MFLOPS. We are encouraged that our performance is within a factor of three of the highly optimized system routine, although the system routine is capable of operating on much larger matrices than the program of Figure 2.

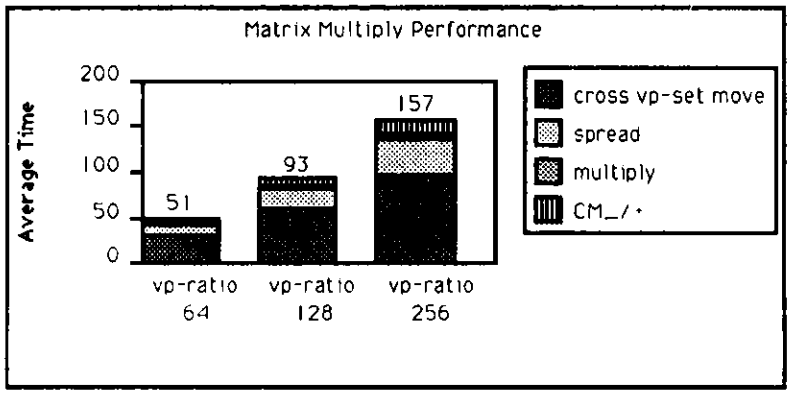


Figure 3: **MM** Performance

In addition to total times, the performance graph also breaks down individual activities within the program. The "cross vp-set move" phase sets the processor grid to three dimensions when the `for all`-loop at line (3) of the program in Figure 2 is entered. The "spread" phase copies each matrix across three dimensions. The "multiply" phase multiplies corresponding elements. Finally, the "CM_/+" phase sums across the third dimension of the product array. While communication dominates total evaluation time at all vp-ratios, this communication is essential for the algorithm.

Table 3 describes the degree of improvement in performance of a compiled version of the **MM** program before and after the improvements suggested in Section 2.3. This version of the **MM** program performs integer arithmetic, which can be slower on the CM-2 than floating-point. Problem sizes are smaller than those presented in Figure 3 because the unimproved programs exhaust memory resources long before the improved versions.

Matrix Dimensions	Times (msecs)	
	Unoptimized	Optimized
16 × 16 and 16 × 16	113	15
32 × 32 and 32 × 32	278	21
64 × 64 and 64 × 64	2,045	63

Table 3: Effects of Program Improvements

4 Summary

We have described a compiler for a dialect of Backus's FP language. The compiler generates intermediate-language programs suitable for data-parallel evaluation. The dialect of FP we have selected for compilation retains the simple algebraic properties of the original FP language. Further, we can perform structure inference on all programs written in this dialect to describe input and output structures of all program expressions. The intermediate-language is similar to conventional languages currently available for data-parallel systems. In fact our compiler was easily adapted to directly produce CM-Fortran programs. We have defined a set of program improvements that take advantage of the single-assignment nature of intermediate-language programs.

We have a number of outstanding research objectives. We end this paper outlining these goals.

Iteration and Recursion The chief limitation we currently impose on FP programs is that they should be non-recursive and non-iterative. Backus defined FP to permit no programmer-defined higher-order functions, though these restrictions are removed in FFP.

The restrictions on iteration and recursion can be partially lifted by imposing structure preservation on these constructs. For example, assuming that function f maps every structure α to another structure α, the following inference rule for iteration holds.

$$\frac{\begin{array}{c} f : \alpha \rightarrow \alpha \\ p : \alpha \rightarrow \textbf{bool} \end{array}}{\texttt{while } p \texttt{ do } f : \alpha \rightarrow \alpha}$$

If all iterative and recursively-defined functions preserve structure, the number of processes needed for function evaluation will always be fixed at the beginning of evaluation. On the other hand, more dynamic allocation of processes will result in a more expressive language, with perhaps poorer performance due to the overhead of dynamic process management.

Programmer-Defined Routing FP defines a large set of routing functions. While it is possible to claim that the existing supply of routing functions is sufficient and "natural," many situations arise in which programmer definition of new routing functions, using the existing set, is at best ungainly. The `PairUp` function clearly exposes these difficulties. We have been experimenting with a notation for programmers to directly describe routing functions, without recourse to a special set [8]. We have also enhanced the compilation function to generate code for programmer-defined routing functions.

Incorporating Machine Characteristics into Compilation On massively parallel architectures there may be different message transmission facilities within the same multi-computer system.

In the CM-2 for example, message transmission between physically adjacent processors can be up to two orders of magnitude faster than transmission between arbitrary processors. As a consequence, on the CM-2, the copy statement in line (2) below should be evaluated as a general message transmission when c is greater than some threshhold value, say 100.

```
(1)     for all 0 ≤ i < n do
(2)         B[i]:=A[i + c]
```

When c is less than the threshhold, the program above should transmit values along physically adjacent processors in c steps. The specific value of c for which different message transmission facilities should be utilized is dependent on the technology of the multi-computer system. To recognize and exploit these tradeoffs, we may need to translate the intermediate-language to a lower-level language in which communication facilities are explicity specified.

Acknowledgements

We thank Pushpa Rao for her editorial comments. Research facilities have been generously provided by Dartmouth College and the University of Victoria.

References

[1] F.E. Allen & J. Cocke, A Catalogue of Optimizing Transformations, *Design and Optimization of Compilers* (R. Rustin, ed.), Prentice-Hall, pp. 1-30, 1972.

[2] A.V. Aho, R. Sethi & J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[3] J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *CACM*, 21(8), pp. 613-641 (Aug. 1978).

[4] K.E. Batcher, Design of a Massively Parallel Processor, *IEEE Trans. Computers*, vol. C-29, pp. 836-44 (Sept. 1980).

[5] G.E. Blelloch, Scans as Primitive Parallel Operations, *IEEE Transactions on Computers*, 38(11), pp. 1526-38 (Nov. 1989).

[6] D.W. Blevins, *et al.*, Blitzen: A Highly Integrated Massively Parallel Machine, *Journal of Parallel and Distributed Computing*, no. 8, pp. 150-60 (1990).

[7] G.L. Burn, Implementing Lazy Functional Languages on Parallel Architectures, *Parallel Computers: Object-oriented, Functional, Logic* (P.C. Treleaven, ed.), John Wiley & Sons, 1990 (Chapter 5).

[8] D. Banerjee & C. Walinsky, An Optimizing Compiler for FP* — A Data-Parallel Dialect of FP, to appear in *3rd International Symposium on Parallel and Distributed Processing*, Dallas (Dec. 1991).

[9] M.C. Chen, A Parallel Language and Its Compilation to Multiprocessor Machines or VLSI, *13th Annual ACM Symposium on Principles of Programming Languages*, pp. 131-39 (Jan. 1986).

[10] L.J. Guibas & D.K. Wyatt, Compilation and Delayed Evaluation in APL, *5th Annual ACM Symposium on Principles of Programming Languages*, Tucson, AZ, pp. 1-8 (Jan. 1978).

[11] W.D. Hillis & G.L. Steele, Jr., Data Parallel Algorithms, *CACM*, 29(12), pp. 1170-83 (Dec. 1986).

[12] P. Hudak, Para-Functional Programming, *IEEE Computer*, 19(8), pp. 60-70 (Aug. 1986).

[13] Parallel Lisp: Languages and Systems, *1989 US/Japan Workshop on Parallel Lisp* (T. Ito & R.H. Halstead, Jr, eds.), in *Lecture Notes in Computer Science*, Springer-Verlag, vol. 441, 1989.

[14] K. Knobe, J.D. Lukas & G.L. Steele, Jr., Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines, *Journal of Parallel and Distributed Computing*, 8, pp. 102-18 (1990).

[15] G.A. Magó, A Network of Microprocessors to Execute Reduction Languages Parts I and II, *International Journal of Computer and Information Sciences*, vol. 8, no. 5, pp. 349-385 (1979), and vol. 8, no. 6, pp. 435-71 (1979).

[16] J.R. McGraw, The VAL Language: Description and Analysis, *ACM TOPLAS*, 4(1), pp. 44-82, (Jan. 1982).

[17] Z.G. Mou & P. Hudak, An Algebraic Model for Divide-and-Conquer and Its Parallelism, *The Journal of Supercomputing*, vol. 2, pp. 257-78 (1988).

[18] R.S. Nikhil, *ID Version 88.1 Reference Manual*, Computation Structures Group Memo 284, Laboratory for Computer Science, MIT, 1988.

[19] *CM-Fortran Reference Manual (Version 5.2-0.6)*, Thinking Machines Corporation, Cambridge, Massachussets.

[20] L.W. Tucker & G.G. Robertson, Architecture and applications of the Connection Machine, *IEEE Computer*, 21(8), pp. 26-38 (Aug. 1988).

[21] P.L. Wadler, Listlessness is Better than Laziness: Lazy Evaluation and Garbage Collection at Compile-time, *1984 ACM Symposium on Lisp and Functional Programming*, Austin, pp. 45-52 (Aug. 1984).

[22] P.L. Wadler, Listlessness is Better than Laziness II: Composing Listless Function, *1985 Workshop on Programs as Data Objects*, Copenhagen, in: *Lecture Notes in Computer Science*, Springer-Verlag, vol. 217, pp. 282-305, 1985.

[23] C. Walinsky & D. Banerjee, A Functional Programming Language Compiler for Massively Parallel Computers, *1990 ACM Conference on Lisp and Functional Programming*, Nice, pp. 131-38 (June 1990).

Improving compilation of implicit parallel programs by using runtime information

John Sargeant
Dept of Computer Science
University of Manchester
Manchester M13 9PL
England
(js@cs.man.ac.uk)

Abstract

In our quest for the "Holy Grail" of efficient implicit parallelism, we have opted for a conventional architecture with physically distributed but virtually shared memory (the EDS machine), and a Large Grain Graph Rewriting computational model. On programming languages, we hedge our bets, but the work described here uses a (strict) functional language. This set of choices eliminates many of the problems and brings those which remain into sharp focus. "All" we need to do is achieve efficient dynamic load balancing, high granularity, and good data structure locality.

We are investigating an approach whereby the program is first compiled with monitoring code inserted, and run (using relatively small data sets) to produce statistics which are fed back into the compiler which then produces optimised parallel code. For strict functional programs, the production of statistics is straightforward, although automatically using them effectively is not. The paper discusses a number of issues, including the effect of higher-order functions and other more advanced language features, and various practical problems.

This work is embryonic, but the method seems to have significant advantages over static analysis or hand annotation, and also provides extra performance information for the programmer.

1 Introduction

We are interested in hardware and software for scalable, wide-purpose, implicitly parallel systems. In such systems, the programmer is responsible for expressing parallel algorithms to solve a problem, and writing them in a language with implicit parallelism. For scalability, the machine needs to have physically distributed memory, although it may well be logically shared. Efficient mapping of the program onto the machine is the responsibility of "the system", not of the programmer. This mapping involves (at least) the following tasks:

- Detection of tasks which can correctly be executed in parallel.
- Deciding where to execute the parallel tasks (load balancing).
- Detection of tasks which are large enough to be efficiently executed in parallel (granularity control).

- Minimising overheads due to data communication (data locality control).

This will be discussed here in the context of strict (higher-order) functional programs, as these represent the simplest useful case, although we are actually interested in more general cases. In the strict functional case, it is straightforward to detect potential parallel tasks at compile time. Likewise, it is generally agreed that load balancing has to be done at runtime, and many mechanisms have been investigated. The interesting question is which part of "the system" should be responsible for granularity and data locality control. Section 3 discusses the options for this, but first I describe the environment in which this work takes place.

2 Background

2.1 The EDS project

The work described here is a (very small) part of the European Declarative System (EDS) project[3, 4, 2]. This is a large ESPRIT II project, involving ICL, Bull, Siemens and many smaller partners. The main aim of the project is to produce a parallel database machine, although there is some work on LISP and Prolog implementation.

The EDS hardware consists of conventional (SPARC) processors, each with local memory, connected by a high-bandwidth multistage switching network. Although the store is physically distributed, it is globally addressable, and the hardware is therefore well suited to our purposes.

The mainstream EDS software consists of a UNIX-like operating system, and a compilation route from ESQL (an extended version of the SQL database query language). The database uses "static" parallelism in the sense that the data is distributed according to a compiler-generated distribution function, and the computation is done where the data is. Our role in EDS is to investigate more dynamic forms of parallelism, and the work described here is part of this.

2.2 The Large Grain Graph Rewriting computational model

The basic principles of the LAGER model are as follows:

- By default, conventional stack-based serial execution takes place. At a point where parallelism can be generated, the compiler plants both serial and parallel code. At runtime, a test is done to see if the parallelism is actually needed. If not, serial execution continues.
- If parallelism is required, a packet is created which encapsulates a parallel task (called an *instance* in EDS terminology). The stack location which will receive the result of the instance is assigned a special value called a hole. The spawning task continues serially.
- When the spawned instance terminates, it fills in the hole. If the spawning process needs a value which is still a hole, it suspends until the hole is filled in, and another process is run if possible.
- Remote data accesses also cause suspension. In the EDS machine, data is copied in sectors (of 128B) and is cached by the virtual memory system. The total number of instances executed is therefore the number of parallel processes started, plus the number of suspensions due to holes or sector copies. We define the granularity to be the average number of instructions executed per instance.

A version of LAGER [10] is implemented using C and macros which expand to calls to low-level EDS parallelism handling primitives. We have a compiler, called FTC [9] which takes the FPM internal format of the Hope+ compiler [6] and produces C-LAGER code. This use of C as an assembly language is very convenient, but has the disadvantage that an inline process switch (on a hole or remote data access) involves switching a whole C runtime stack. Together with the need to save and restore all the SPARC registers, this makes such process switches uncomfortably expensive, and we therefore work quite hard to minimise the number of them.

Other features of LAGER, including a version implemented as an abstract machine code and its translation to SPARC machine code, are described in [8]

2.3 An example program

The following Hope+ program grows a balanced binary tree and then sums it.:

```
data Tree(num) == Leaf(num) ++ Node(Tree(num) X Tree(num));

dec grow : num -> Tree(num);
--- grow(0) <= Leaf(0);
--- grow(n) <= Node(grow(n-1),grow(n-1));

dec treesum : Tree(num) -> num;
--- treesum(Leaf(k)) <= k+1;
--- treesum(Node(n,m)) <= 1 + (treesum(n) + treesum(m)) div 2;

treesum(grow(12));
```

The significant part of C-LAGER code produced by the compiler is as follows: For grow:

```
Spawn(int,result_1,grow( arg1 - 1 ));
 result_2 = grow( arg1 - 1 );
 TestHole(result_1);
 result = mk_tuple3( 2, result_1, result_2 );
```

and for treesum:

```
Spawn(int,result_1,treesum( local1[1] ));
 result_2 = treesum( local1[2] );
TestHole(result_1);
 result = 1 + ( ( result_1 + result_2 ) / 2 );
```

Spawn is a macro which tests whether parallelism is required and if so creates an instance to do the parallel task. TestHole tests for a hole and causes a suspension if a hole is found. In general, if n tasks can be done in parallel, the compiler plants n-1 Spawns and leaves the last one to be done serially. This requires fairly straightforward static analysis, which also detects trivial (ie. non-recursive) functions and avoids trying to Spawn them.

To achieve good speedup for even this very simple, obviously parallel, program is surprisingly difficult. It is necessary to control granularity by ensuring that only the Spawns high up in

the execution tree actually create parallel tasks. For near-linear speedup, it is also necessary to cause the treesums to be executed where their data is, and to reconcile this with the needs of the dynamic load balancing. There are a number of traps and pitfalls, even in getting the basic mechanisms right. These issues are discussed in detail in [11].

2.4 Some experience with a parallel application program

Howard [5] wrote a relational database implementation in Hope+. The program loads a database and then allows SQL queries to be made on it. The program is quite unusual in being a substantial functional program (around 1500 lines of Hope+) which was written from the beginning with parallelism in mind. Relations are implemented as tree structures, and queries as transformations on trees. The need to generate significant parallelism from relatively small data sizes (to allow detailed simulation) meant that fairly inefficient join algorithms were used. The program is therefore not competitive with conventional SQL implementations, but is a good example of irregular manipulations on non-trivial data structures.

Poor results were obtained by just taking the code produced by the FTC compiler, because too many unnecessarily small processes were spawned. On the other hand, attempts to distribute the data and then follow it around (using hand-modified code) also produced poor results, because less parallelism was exploited and the data following and load balancing interfered with each other. In the end, the best results were obtained by compiling purely serial code, and then inserting Spawns in carefully chosen places. By this technique, reasonable speedups (up to 10 with 16 processors) were obtained. However, this performance was far from optimal. Consider the following sample figures, obtained from the EDS machine simulator [7]

PEs	Instances	Copies	Granularity	Speedup
1	1	0	27193142	1.0
2	902	576	30200	1.9
4	3541	1554	7758	3.5
8	7754	3011	3588	6.0
16	19542	5794	1457	9.0

The instances figure is the number of instances started plus the number of suspensions, as described above. The granularity is the total number of instructions executed divided by the number of instances. Speedup is simulated runtime divided into the 1-processor simulated runtime. As the number of processors is increased, the granularity falls sharply, because of increasing numbers of sector copies, and also because increasing numbers of small processes are being generated, showing that the load balancing becomes less effective. Clearly the scalability to large numbers of processors is poor.

The conclusions we drew from this exercise were:

- It's possible to get reasonable results for small numbers of processors by straightforward load balancing, provided that the instances spawned are carefully chosen.
- For more scalable results, more sophisticated distribution, including some data following, would be required.
- The program was too complex to do this by hand. A better understanding of the tradeoffs should be obtained on smaller programs first.

- In order to tackle real programs eventually, much better tools are needed. The following sections suggest what some such tools should do.

3 Approaches to the mapping problem

A mentioned above, the process of efficiently mapping an implicitly parallel program onto a parallel machine with physically distributed memory involves control of granularity and data locality. There are a number of approaches to this problem.

3.1 Static analysis

Static analysis is of limited benefit, because we are dealing with dynamic properties of programs. For instance, the sizes and shapes of computation trees are not known statically. Likewise, the sizes and shapes of the data structures produced and consumed by computations are not known statically. Neither, in general, is the pattern of sharing of data. Higher-order functions reduce still further the amount which is known at compile-time.

However, it is obviously important to obtain as much information as possible statically, and there are a number of things we can do. The easiest case is to recognise trivial (ie. non-recursive) functions as not being worth spawning. The current FTC compiler in fact does this.

A slightly more sophisticated technique is to place a partial ordering on functions, by saying $A \geq B$ if A calls B (so mutually recursive functions are equal in this ordering). For instance, consider the following fragment of a matrix multiplication program:

```
--- elemmult(val,nil,col)       <= val ;
--- elemmult(val,rh::rt,ch::ct) <= elemmult(val+(rh*ch),rt,ct) ;

--- rowmult(row,nil)  <= nil ;
--- rowmult(row,h::t) <= elemmult(0,row,h)::rowmult(row,t) ;

--- matmult(nil,b)  <= nil ;
--- matmult(h::t,b) <= rowmult(h,b)::matmult(t,b) ;
```

A compiler can determine that matmult > rowmult > elemmult and therefore probably deduce that the right thing to do is to spawn the calls of rowmult from matmult[1]

Program transformation has a role to play. For instance, the treegrow program above can be transformed into one where the data structure is consumed as soon as it it produced, and thus improve its performance by avoiding remote data access.[2] This would take away the point of the program as a benchmark, and could not be done for more complex cases such as the database example. On the other hand, it does suggest a powerful use of program transformation, in reducing the distance between the production and consumption of data structures.

Abstract interpretation may also be useful, for instance in determining whether data structures can be shared or not. However, since abstract domains are limited to a small number of discrete points, abstract interpretation, as with all these techniques, necessarily gives information about *discrete qualities* (defined or not defined, shared or not shared etc.) and we are

[1] For reasons not relevant here, it's better to spawn the smaller computation, and do the larger one serially.

[2] My thanks to David Lester for pointing this out.

interested in *continuous quantities*. The sort of thing we want to know is whether $f(x)$ is big enough to spawn, or whether $g(p, q)$ references p more frequently than q or vice-versa.

3.2 Runtime heuristics

Runtime techniques, on the other hand, have to work with very limited information to be acceptably efficient, and so are inevitably heuristic in nature.

For instance, lazy task creation[1] is a useful technique to improve granularity and reduce idle time, by avoiding premature commitment to serial execution, and favouring tasks higher in the computation tree to those lower down. However, Aharoni[12] shows that lazy task creation alone does not prevent the system from spawning many small tasks, especially for programs which have insufficient parallelism to occupy all the processors. Aharoni in turn suggests an adaptive scheme where the computation tree is locally expanded breadth-first to reveal its shape, and the generation of parallel tasks is adjusted according to the information this provides. Aharoni shows that the method adapts very quickly to changes in the nature of the computation tree, and produces some impressive results. Unfortunately, breadth-first expansion is essential to the technique, and is unacceptably inefficient compared to the depth-first stack-based evaluation used by LAGER and similar models.

In the past, we have favoured the use of runtime mechanisms where possible. However, mechanisms using simple, purely local, information tend to be ineffective, whereas using more complex, global information makes them expensive. A good example is the search for a "throttle" to control the excess parallelism which wrecked the Manchester Dataflow Machine[13]. Early proposals involved hardware to schedule individual tokens. This didn't work because the hardware didn't have enough local information to get the scheduling right. The solution we eventually adopted involved a data structure which was a fairly complete representation of the computation tree, and this would probably have been very expensive in a large-scale machine.

3.3 Annotations

The usual solution to the problem is to give the problem back to the programmer, in the form of annotations. This is currently the best available technique we know of, but there are a number of reasons for considering it undesirable as the complexity of programs, and the size of machines, increase.

- Annotations are a hassle, and require considerable skill to get right. At least that was our experience with the database program described above.
- Annotations may be machine-dependent; a small shared-store machine will have very different requirements to a large distributed-store machine. Annotations therefore need to be parameterised by machine characteristics.
- Annotations need to be introduced or modified when existing code is ported or incorporated into new programs. There is some loss of modularity, as the user of a function has to know something of its runtime characteristics.
- In, for instance, divide-and-conquer programs, the same set of function calls has very different characteristics depending on its position in the execution tree. Annotations therefore need to be parameterised by this position, or something equivalent.

It seems, then, that the inability of either static analysis and runtime heuristics to solve the problem forces us to use complex, heavily parameterised, annotations. There is one other way to find out the properties of a program: by running it.

4 Outline of the technique

The basic idea is to run the program, on relatively small data, with profiling code inserted. The statistics thus collected are analysed by an "expert system" (parameterised by the characteristics of the machine) which produces a set of recommendations which are fed back to the compiler, which then produces optimised parallel code. The problem then divides into finding suitable measures, and deciding how to use them. The first of these turns out to be straightforward, at least for the class of programs we're dealing with. The second is, of course, the hard part.

4.1 What measures should we use?

We require measures which are independent of the actual execution order of the program, and which scale sensibly with the problem size.

One important measure is the total amount of work required to execute a function, ie. the cost of execution on one processor, which we'll call C_1. This is what a standard execution profiler, such as prof, gives you. Another is the length of the critical path through the program, or equivalently the time taken if arbitrarily many processors are available, ignoring all overheads of parallel execution. This is C_∞. The ratio

$$C_1/C_\infty = \Pi$$

is known as the *average parallelism* of the program. Π is a good abstract measure of program parallelism *and is independent of the actual execution order.* It is straightforward to plant code which will accumulate these figures. Each function passes back its result, plus values for C_1 and C_∞. At a point where several expressions could be evaluated in parallel, the overall value of C_1 is the sum of the individual C_1 values, and the overall C_∞ in the maximum of the C_∞ values. Analogous measures D_1 and D_∞ can be defined for data structures.

4.2 Calculating cost functions

let $C_1(E)$ be the 1-processor cost of evaluating expression E (e.g. the number of clock cycles required), $C_\infty(E)$ be the infinite processors cost, $V(E)$ be the value returned by E, where E is one of the following:

a constant,	K
a builtin function or operator,	$op(E1 \ldots En)$
a user-defined function,	$f(E1 \ldots En)$
a conditional expression,	$if\ E0\ then\ E1\ else\ E2$

In principle, $C_1(E)$ and $C_\infty(E)$ can be calculated as follows. Constants are assumed to be created by sequential code, the cost of which can be statically determined:

$$C_1(K) = C_\infty(K) = StaticCost(K)$$

For a builtin function, C_1 is the cost of evaluating the arguments, plus the cost of executing the builtin itself. This will often be statically determined, but in general may be a function of the values of its arguments (e.g. the cost of allocating a memory cell may depend on its size):

$$C_1(op(E1 \ldots En)) = C_1(E1) + \ldots + C_1(En) + StaticCost(op, V(E1) \ldots V(En))$$

Since we are assuming strictness, the argument expressions may be evaluated in parallel, so C_∞ is the maximum of the C_∞ values for the subexpressions, plus the cost of executing the builtin (assumed to be done serially):

$$C_\infty(op(E1 \ldots En)) = max(C_\infty(E1) \ldots C_\infty(En)) + StaticCost(op, V(E1) \ldots V(En))$$

For a user-defined function call, C_1 consists of the cost of evaluating the arguments, the cost of the call itself (assumed to be the same for any function call of n arguments), and the cost of evaluating the function given the values of the arguments:

$$C_1(f(E1 \ldots En)) = C_1(E1) + \ldots + C_1(En) + CallCost(n) + C_1(f(v(E1) \ldots V(en)))$$

Again, the argument expressions can be evaluated in parallel, but all must be evaluated before the function can be called, and the call itself is assumed to be done serially (no distinction is made between the costs of serial and parallel calls in this abstract model):

$$C_\infty(f(E1 \ldots En)) = max(C_\infty(E1) \ldots C_\infty(En)) + CallCost(n) + C_\infty(f(v(E1) \ldots V(En)))$$

The C_1 cost of a conditional expression is the cost of evaluating the condition, plus the cost of evaluating the selected expression, plus the cost (assumed static) of executing the conditional itself:

$$C_1(if\ E0\ then\ E1\ else\ E2) =$$
$$C_1(E0) + StaticCost(ifthenelse) + (if(V(E0) = true)\ then\ C_1(E1)\ else\ C_1(E2))$$

Since the evaluation has to be done serially, the equation for Cinf is similar:

$$C_\infty(if\ E0\ then\ E1\ else\ E2) =$$
$$C_\infty(E0) + StaticCost(ifthenelse) + (if(V(E0) = true)\ then\ C_\infty(E1)\ else\ C_\infty(E2))$$

The equations for C_∞ reflect the influence of data dependencies on the program. When expressions can be evaluated in parallel, the maximum C_∞ value is taken. When data dependencies require serial evaluation, the C_∞ values are added. Although it is necessary to execute the program to find C_1 and C_∞, because they depend in general on some $V(E_i)$ values, the

actual execution order does not matter - it could be serial or it could be some arbitrary parallel order. For data structures without sharing (i.e. trees), D_1 and D_∞ are easy to calculate. Such a structure is one of:

a constant,	K
a tuple containing other structures,	$tuple(S1 \ldots Sn)$

It's convenient to define

$$D_1(K) = D_\infty(K) = 0$$

The D_1 cost for a tuple is the sum of the cost of the subtrees plus the space occupied by the tuple itself, including any (static) overhead:

$$D_1(tuple(S1 \ldots Sn)) = D_1(S1) + \ldots + D_1(Sn) + n + StaticCost(node)$$

for D_∞, we take the maximum and add 1 to give the critical path length:

$$D_\infty(tuple(S1 \ldots Sn)) = max(D_\infty(S1) \ldots D_\infty(Sn)) + 1$$

The inherent parallelism of a data structure can be defined by

$$D_1/D_\infty = D\Pi$$

For instance a simple list has $D\Pi = 1$, reflecting the fact that operations on it are likely to be serial, while a balanced binary tree of n levels has $D\Pi = 2^n/n$.

The definition of D_∞ also works for general acyclic graphs. Unfortunately, the definition of D_1 is dubious in the presence of sharing, since a shared substructure will be counted once for each time it is referenced. Although it's not difficult to define D_1 for a graph in terms of the total space occupied by the graph, this is expensive to calculate in general. Instead, we will stick with the above definition of D_1, and therefore of $D\Pi$, even for graphs. Apart from practical considerations, it may be reasonable to define $D\Pi$ this way, because if a structure is referenced from n places, n parallel processes can use these references to access the structure at the same time.

4.3 Accumulating raw data

The compiler plants code which accumulates values for C_1 etc., based on the above definitions. In practice, of course, the equations for a number of primitives can be combined, to give more efficient code. Each function call returns a tuple containing its actual result, plus the values of C_1 and C_∞.

Producing D_1 and D_∞ figures is similarly easy. Each heap cell is extended with two integer fields. When a tuple is created, the values of D1 and Dinf are calculated from the values stored in the substructures, according to the formulae above, and stored in turn. The values can then

be reported each time a structure is passed as an argument to, or returned as a result from, a function.

The figures produced need to be associated with the points at which functions are called, rather than the bodies, in order to distinguish between calls from different places. For instance, consider the matrix multiply example above. The call to rowmult from matmult deals with a whole row, so will consistently have a high C_1. The recursive call within rowmult itself will vary in size depending on how much of the row is left. The analysis should show that doing rowmult and matmult in parallel is a better bet than the elemmult/rowmult pair.

To avoid confusion between static and dynamic meanings of the word "call" , we will refer to the place in the program text where a function is called as a *callpoint*. The monitoring code accumulates, for each callpoint, figures such as the number of calls, the means and standard deviations of C_1 and C_∞, and of the D_1 and D_∞ figures for the parameters and the result.

For instance, the code for the body of the treegrow function looks as follows:

```
returnval* grow(arg1)
{ int result, C1=0, Cinf=0;
if (0 == arg1 )
{
result = mk_tuple2( 1, 0 );
C1   += 2; Cinf += 1;
}
   else    {
result = mk_tuple3( 2,
update( grow( arg_mon(arg1 - 1,0, &STATS[4]) ), &STATS[4] ),
update( grow( arg_mon(arg1 - 1,0, &STATS[5]) ), &STATS[5] ) );
C1   += 5 + C1_cost( &STATS[4] ) + C1_cost( &STATS[5] );
Cinf += max(2 + Cinf_cost( &STATS[4] ), 2 + Cinf_cost( &STATS[5] ));
}
C1   += 2; Cinf += 2;
return(pack(result,C1,Cinf));
} /* end of function grow */
```

The result of the function is of type *returnval*, ie a record containing the result and values for C_1 and C_∞. Within the function, these are represented as variables, which are updated at various points. The statistics are held in a global array, STATS, with an entry for each callpoint (so the first recursive call of grow is callpoint 4, and the second is callpoint 5).

A number of utility functions update the stats. *pack* simply creates a record containing the result, C_1 and C_∞. This record is used by *update* to update the stats for the callpoint. *arg_mon* updates the values of D_1 and D_∞ for an argument. *C1_cost* and *Cinf_cost* recover the appropriate values to put into the expressions for the overall values of C_1 and C_∞ for the function.

Using callpoints rather than lumping all calls to a function together clearly gives more accurate results. However, it is not necessary to stop there. In general, the behaviour of a function may depend on its position in the overall call tree. For instance, if A calls B, the behaviour of B may differ depend on whether A was called from P or from Q. However, detecting this requires more expensive monitoring, and implies a cost at "real" runtime, since if there

really was a difference the calling mechanism would have to know which route it came from. Using static callpoints seems a good compromise between complexity and accuracy, but more experience is needed to confirm this.

Another question is what are sensible values to accumulate. There are many possibilities, for instance:

Averages: Average values for C_1 and C_∞ for each callpoint of each function. Likewise, average values for D_1 and D_∞ for the data structures they consume and produces.

Standard deviations: SDs for the above measures. Of course the SD can be calculated on the fly - it's not necessary to keep all the values.

Histograms: A more accurate picture of the spread of values may be required than that provided by a simple mean and SD. This could be done by keeping histograms showing how many times the values fall within certain ranges.

Correlations: It may be useful to know how the values of C_1 and C_∞ for a function, and D_1 and D_∞ for its output, vary with the values of D_1 and D_∞ for the arguments.

Other statistics which could be useful are the number of data structure accesses done by a function and maybe the amount of data it creates (different from the output D_1 because it includes garbage, but excludes structures provided as arguments and counts shared structures only once). A function which does a high proportion of data references to computation should be sent to its data, whereas the load balancing should be free to do what it wants with a function doing little data accessing.

4.4 What should we do with them?

The most immediate benefit is that we get values of Π for the program, and for its components. This suggests a maximum number of processors on which it is sensible to run the program (especially if Π is small and varies little with data size), and identifies serial bottlenecks. This is itself very useful; it takes naive programmers some time to come to terms with Amdahl's Law.

The next easiest case is sets of callpoints which can be executed in parallel and which have consistent values of C_1 (ie. a low standard deviation). If two or more have high mean C_1, they should be executed in parallel, otherwise not. When callpoints have widely varying C_1s, (eg. in recursive divide-and-conquer programs) things are obviously more difficult. A promising technique is to attempt to generate parallel tasks for those instances of such callpoints which are higher up the dynamic call tree. This is explored in more detail in the next section.

Functions which act as net producers or consumers of data can be identified by correlating the D_1 values of their parameters and results with their C_1 values. Generally speaking, producers of large data structures should be load balanced, whereas consumers should follow their data. Functions which produce large, parallel data structures, (large D_1, small D_∞) even if they themselves are serial, can also be identified, and are candidates for enforced distribution in order to distribute the result data structure. An example of this occurs in the database program, which initially reads in the database serially, but the data structure thus created is then used in parallel.

Clearly an analyser program which works well for large programs on large machines needs to be very sophisticated. Nevertheless, the ease with which a substantial amount of information can be extracted suggests that the technique has promise.

4.5 Using information about call depth

The depth of each call can be checked in the monitoring code, and correlated with C_1. The information isn't normally available in the live code, but it would be possible to pass the call depth as an extra parameter. The overhead of this wouldn't be large, since it would only be necessary to do it for calls near the top of the call tree. However, the absolute call depth does not in general give the right information. For instance, the full version of the treegrow program has a "loop" to generate a series of trees:

```
dec itergrow : num X num -> num;
--- itergrow(ts,0) <= ts;
--- itergrow(tsize,count) <= itergrow(treesum(grow(tsize)),count-1);
```

The recursive calls within the treesum and grow functions have highly variable C_1. Counting the absolute call depth is not adequate, since the depth at which treesum and grow are called varies as itergrow recurses. However, the calls to treesum and grow from within itergrow provide good starting points. If these callpoints can be identified as roots, the computation can be distributed with very good granularity. What we really want to know, therefore, are call depths relative to nodes which are the roots of parallel subtrees. The problem therefore reduces to the problem of identifying such nodes. This is crucial, since if they can be found reliably, good granularity can be obtained without using sophisticated runtime load balancing heuristics, and indeed the load balancing in general becomes much less critical.

There are several ways in which we might attempt to detect these nodes. They represent places in which the computation "changes mode" and would therefore be expected to be related to specific callpoints, as in the example. They can probably be identified on statistical evidence (high C_1 and Π, few calls in total etc.) but in fact they can probably be found statically from inspection of the static call graph.

5 Higher-order functions

Higher-order functions are important in the functional style of programming. In general-purpose parallel programming they are arguably even more important, because it is desirable to have libraries of standard functions which manipulate data structures using parallel algorithms, and such functions are in many cases naturally higher-order.

However, such functions present a number of problems. Consider one of the simplest of such library functions, map.

```
--- map(f, nil) <= nil;
--- map(f, h::t) <= f(h)::map(f, t);
```

In principle, the call to f can be evaluated in parallel with the recursive call to map. So should it be? The writer of map cannot know, because it depends on the properties of f. It

is therefore necessary to use information about f *at runtime* to decide whether to spawn a parallel task or not. One way to implement this is to represent f as a pointer into a table which contains, as well as the code pointer, a value which indicates whether f should be evaluated serially, spawned in parallel, or made to follow its argument, etc. (In the case of a partially parameterised function, the table pointer will be part of a packet containing the argument values already collected.)

For statistics-gathering purposes, it is necessary to treat the call to f rather differently to a first-order callpoint. For some programs f may be always big enough, or always too small, but in general it will be different for different fs. It is therefore necessary to record statistics for each f. Again, this can be done by representing f as a pointer into a table, in which the figures are accumulated.

As before, there is a tradeoff between the amount of information gathered and the accuracy of the results. It is not clear, for instance, whether it is necessary to keep separate statistics for each function for each higher-order callpoint, or whether functions used this way have consistent behaviour across callpoints. Likewise, it is not obvious whether it is sensible to use information from all the calls to a function (including the first-order ones) in deciding whether to spawn it when used this way.

6 Results

One option to the FTC compiler plants code which can be run serially to generate the statistics. A very primitive version of the analyser exists. It does some granularity analysis, but currently makes no attempt to deal with data locality issues.

Currently, the software is not sufficiently stable to produce interesting results from large programs. However, the appendix shows some statistics produced from the matrix multiplication program mentioned earlier, and discusses some of the things the figures show. I do not claim that the program is beyond the wit of programmer to annotate (although there are one or two surprises). I do claim the converse, namely that most[3] annotations a programmer might write on that program can be generated automatically.

7 Open issues

7.1 The question everybody asks

How do you know that your test data is large enough, and covers all the cases? Of course this can't be answered rigorously, but intuitively the results should be robust given reasonable testing.

For a program to be worth this treatment, it must either have a large runtime on real data, or be expected to run many times. After all, 100MIP serial workstations will soon be commonplace. Conversely, it will be quite reasonable to run programs for, say, a few minutes on such workstations to produce the statistics, thereby gathering a great deal of information. It is hard to imagine that a program which runs for 10 minutes (slowed down by, say a factor of 3 by the monitoring) will behave radically differently when run for 10 hours for real.

[3] I say most rather than all, because with this very regular program there are, of course, special-purpose tricks to distribute the data evenly etc.

The statistics are additive; results from runs with different data can be combined in sensible ways. Although quantitative results are produced, they do not have to be exact. It is not necessary to test every case provided that all the substantial parts of the program are run.

For instance, in a database implementation, it would be necessary to test all of the various join algorithms implemented. But this testing would be necessary anyway, as part of the normal development of the program.

7.2 Different programming styles

The techniques described above rely on the fact that, for strict functional programs, the relevant statistics are independent of execution order. For more general programs, this is not the case. For instance, in an imperative program using locks or barrier synchronisation, the value of C_∞ can be affected by whether a process on the critical path is held up by a lock or barrier. In a lazy functional language implementation, the is a similar dependence on whether shared values are evaluated on the critical path or elsewhere. There is a further problem in this case in measuring the size of lazy data structures.

I conjecture that the techniques described above can probably be adapted to work in these cases; the nondeterminism is localised and results obtained from one evaluation order are likely to be approximately correct for others. There are, however, classes of program for which evaluation order critically affects performance, for instance in branch-and-bound algorithms and some nondeterministic logic programs. In such cases, it is necessary to have a detailed understanding of the problem to know what a "sensible" evaluation order is.

7.3 "Software engineering" and practical problems

A tool is only useful if it fits well into the overall process of producing software. There are several areas where quite a lot of work would be needed to develop a production tool.

Combination of results. There have to be sensible ways of combining results from different runs, and from different modules, preferably without explicit guidance from the user. This does not seem to present any problems in principle, since the results are essentially additive.

User interfaces. A helpful user interface is required to ease the process of going through the cycle of testing, results production, and feedback, while ensuring that all significant parts of the program are properly tested. It is also necessary to present the information extracted from the statistics in a suitably concise and meaningful way.

Robustness. It is important that small changes to a module (which move callpoints around, for instance) do not require the whole process to be done all over again. On the other hand, it is necessary to detect when a change to a module does affect the way it should be executed, and even the way *other* modules should be executed.

8 Conclusions

Large-scale implicit parallelism is difficult. We need the best tools we can produce. It is unclear at present exactly what tools are required, and which parts of the "system" should be

responsible for the vaious parts of the mapping problem. However, this paper has suggested a new labour-saving device.

It is too early to be confident of how effective the approach will be, but the quantity of interesting, quantitative, information which can easily be produced is promising.

9 Acknowledgements

The Lager computational model and its C implementation are largely the work of Ian Watson. When this paper says "we" it usually means "Ian and I". Various past members of the EDS and Flagship projects contributed to the software, notably Paul Watson, Nigel Paver and Mark Greenberg. The work required to produce the FTC compiler was greatly reduced by being able to use the Imperial College Hope+ front end.

References

[1] **Lazy task creation: a technique for increasing the granularity of parallel programs** E. Mohr, D.A. Kranz & R. H. Halstead ACM Conference on Lisp and Functional Programming, Nice, France, June 1990.

[2] **A process and memory model for a parallel distributed-memory machine** P. Istaverinos, L. Borrmann ConPar 90, LNCS 457, pp 479-488

[3] **EDS Hardware Architecture**, M. Ward, P. Townsend, G. Watzlawik ConPar 90, LNCS 457, pp 816-827

[4] **Design and simulation of a multistage interconnection network**, R. Holzner, S. Tonmann, ConPar 90, LNCS 457, pp 385-396

[5] **A Relational Database Management System in a Pure Functional Language**, C. Howard, MSc.dissertation, University of Manchester, 1990.

[6] **Hope+**, N. Perry, Report IC/FPR/LANG/2.5.1/7, Imperial College, London, 1988

[7] **EDS Parallel Machine Simulator**, N.C. Paver, EDS report, EDS.UD.3I.M001, University of Manchester, 1990

[8] **Large Grain Graph Reduction on a RISC Architecture**, N.C. Paver, MSc. dissertation, University of Manchester, 1989.

[9] **FTC: the FPM to Lager-C Compiler**, J. Sargeant, EDS report EDS.UD.3I.M002, University of Manchester, 1990

[10] **C-Lager Definition**, I. Watson, EDS report EDS.UD.3I.M004, University of Manchester, 1990

[11] **Some experiments in controlling the dynamic behaviour of parallel functional programs** J. Sargeant, I. Watson, Proc. Workshop on the Parallel Implementation of Functional Languages, Southampton, June 1991, Southampton University tech. report CSTR 91-07, pp 103-121.

[12] **A strategy for the run-time management of fine-grain parallelism**, G. Aharoni, Y. Farber, A. Barak, Proc. Workshop on the Parallel Implementation of FUnctional Languages, Southampton, June 1991, Southampton University tech. report CSTR 91-07, pp 227-245.

[13] **Control of parallelism in the Manchester Dataflow Machine**, C.A. Ruggiero, J Sargeant, Third International Conference on Functional Programming Languages and Computer Architecture, September 1987, LNCS 274.

A Results for the matrix multiplication program

The following is the full version of the matrix multiplication program, with callpoints (as numbered by the compiler) indicated in curlies. Of course this is not claimed to be an efficient matrix multiplication program.

```
dec elemmult: num X list(num) X list(num) -> num ;
--- elemmult(val,nil,col)       <= val ;
--- elemmult(val,rh::rt,ch::ct) <= {6}elemmult(val+(rh*ch),rt,ct) ;

dec rowmult: list(num) X list(list(num)) -> list(num) ;
--- rowmult(row,nil)  <= nil ;
--- rowmult(row,h::t) <= {21}elemmult(0,row,h)::{22}rowmult(row,t) ;

dec matmult: list(list(num)) X list(list(num)) -> list(list(num)) ;
--- matmult(nil,b)  <= nil ;
--- matmult(h::t,b) <= {15}rowmult(h,b)::{16}matmult(t,b) ;

dec kroneker: num X num -> num ;
--- kroneker(i,j) <= if (i = j) then 1 else 0 ;

dec position: num X num -> num ;
--- position(i,j) <= 100*i + j ;

dec genrowpos: num X num X num -> list(num) ;
--- genrowpos(i,j,n) <= if (j > n) then nil
                        else {13}position(i,j)::{14}genrowpos(i,j+1,n) ;

dec genmatpos: num X num -> list(list(num)) ;
--- genmatpos(i,n) <= if (i > n) then nil
                      else {9}genrowpos(i,1,n)::{10}genmatpos(i+1,n) ;

dec genrowkron: num X num X num -> list(num) ;
--- genrowkron(i,j,n) <= if (j > n) then nil
                         else {11}kroneker(i,j)::{12}genrowkron(i,j+1,n) ;

dec genmatkron: num X num -> list(list(num)) ;
--- genmatkron(i,n) <= if (i > n) then nil
                       else {7}genrowkron(i,1,n)::{8}genmatkron(i+1,n) ;
```

```
dec printlist: list(num) -> list(char);
--- printlist(nil) <= nil;
--- printlist(h::t) <= {17}numtostr(h) <> ", " <> {18}printlist(t);

dec printmat: list(list(num)) -> list(char);
--- printmat(nil) <= nil;
--- printmat(h::t) <= {19}printlist(h) <> "\n" <> {20}printmat(t);

let n == 20
  in let m1 == {1}genmatpos(1,n)
  in let m2 == {2}genmatkron(1,n)
 in {5}termout( {4}printmat({3}matmult(m1,m2)) <> "\n" );
```

The compiler works out the sets of callpoints which can be executed in parallel, which are:

```
[ 1 2 ] [ 3 ] [ 4 ] [ 5 ] [ 6 ] [ 7 8 ] [ 9 10 ] [ 11 ] [ 12 ] [ 13 ]
[ 14 ] [ 15 16 ] [ 17 18 ] [ 19 20 ] [ 21 22 ]
```

Sets of 1 callpoint are not interesting (unless we want to do forced data distribution). Sets of more than 2 occur in some programs (eg. doing matrix multiplication with quadtrees rather than lists). For the pairs, on this 20*20 data size, we get the following:

```
1: "genmatpos from toplevel" one
C1 = 2942, Cinf = 161, D1 = 840, Dinf = 40

2: "genmatkron from toplevel" one
C1 = 2542, Cinf = 161, D1 = 840, Dinf = 40
```

The first line gives the callpoint number, identifies the callee and caller, and gives the number of calls. In general, the maximum, and mean values of the various stats, along with a crude histogram are given, although when, as here, there is only one call, a more compact format is used. The D_1 and D_∞ figures for the result are always given, those for arguments are given when they are non-zero (ie. for structured arguments only).

The units in which C1 and Cinf are measured are currently pretty arbitrary, and the system is totally untuned. Very roughly, a C1 in 4 figures certainly justifies spawning a task, one in 2 figures usually does not. These two functions are, of course, only called once and are big enough to be worth doing in parallel. Notice the large, parallel, data structures which are produced (the matrices, of course).

```
7: "genrowkron from genmatkron" 20
! stat      max      mean        SD    <100  100-1K   1K-10K   >10K
C1          122       122         0       0      20        0      0
Cinf         82        82         0      20       0        0      0
D1           40        40         0      20       0        0      0
Dinf         20        20         0      20       0        0      0

8: "genmatkron from genmatkron" 20
! stat      max      mean        SD    <100  100-1K   1K-10K   >10K
C1         2415      1208       732       1       7       12      0
```

```
Cinf        157       115        33         5        15         0         0
D1          798       399       242         2        17         0         0
Dinf         39        28         8        19         0         0         0

9: "genrowpos from genmatpos" 20
! stat      max      mean        SD      <100   100-1K   1K-10K    >10K
C1          142       142         0         0        20         0         0
Cinf         82        82         0        20         0         0         0
D1           40        40         0        20         0         0         0
Dinf         20        20         0        20         0         0         0

10: "genmatpos from genmatpos" 20
! stat      max      mean        SD      <100   100-1K   1K-10K    >10K
C1         2795      1398       847         1         6        13         0
Cinf        157       115        33         5        15         0         0
D1          798       399       242         2        17         0         0
Dinf         39        28         8        19         0         0         0
```

These two pairs are very similar. In each case, the smaller one of the pair (genrowcron or genrowpos) is rather small. However, we might well want to distribute the output data structure. Another argument for spawning them is that, since these functions produce data but don't consume it, they will run to completion once spawned, rather than process switching on remote data accesses.

Callpoints !1,12 and 13,14 are not candidates for parallel execution, because the kronecker and position functions can statically be seen to be trivial. For what it's worth, the statistics confirm this:

```
11: "kroneker from genrowkron" 400
! stat      max      mean        SD      <100   100-1K   1K-10K    >10K
C1            2         2         0       400        0         0         0
Cinf          2         2         0       400        0         0         0
D1            0         0         0         0        0         0         0
Dinf          0         0         0         0        0         0         0
```

We now come to the big ones:

```
15: "rowmult from matmult" 20
! stat      max      mean        SD      <100   100-1K   1K-10K    >10K
C1         3322      3322         0         0        0        20         0
Cinf        161       161         0         0       20         0         0
D1           40        40         0        20        0         0         0
Dinf         20        20         0        20        0         0         0
arg 0
D1           40        40         0        20        0         0         0
Dinf         20        20         0        20        0         0         0
arg 1
D1          840       840         0         0       20         0         0
Dinf         40        40         0        20        0         0         0

16: "matmult from matmult" 20
```

! stat	max	mean	SD	<100	100-1K	1K-10K	>10K
C1	63196	31599	0	1	0	3	16
Cinf	218	181	44	1	19	0	0
D1	798	399	242	2	17	0	0
Dinf	39	28	8	19	0	0	0
arg 0							
D1	798	399	242	2	17	0	0
Dinf	39	28	8	19	0	0	0
arg 1							
D1	840	840	0	0	20	0	0
Dinf	40	40	0	20	0	0	0

This is clearly the key pair to parallelise. It is also clear from the sizes of the arguments that some data following may be required, and in particular, that a good plan is for rowmult to follow its second argument.

17: "numtostr from printlist" 400

! stat	max	mean	SD	<100	100-1K	1K-10K	>10K
C1	59	53	6	400	0	0	0
Cinf	27	24	2	400	0	0	0
D1	4	4	0	400	0	0	0
Dinf	2	2	0	400	0	0	0

18: "printlist from printlist" 400

! stat	max	mean	SD	<100	100-1K	1K-10K	>10K
C1	1237	563	348	40	316	44	0
Cinf	85	53	19	400	0	0	0
D1	6	5	1	380	0	0	0
Dinf	3	2	0	380	0	0	0
arg 0							
D1	38	19	11	380	0	0	0
Dinf	19	9	5	380	0	0	0

numtostr is a standard function which converts a number to a string for printing. Not surprisingly, it doesn't take long and we can forget this one.

19: "printlist from printmat" 20

! stat	max	mean	SD	<100	100-1K	1K-10K	>10K
C1	1302	1185	129	0	0	20	0
Cinf	88	85	2	20	0	0	0
D1	6	6	0	20	0	0	0
Dinf	3	3	0	20	0	0	0
arg 0							
D1	40	40	0	20	0	0	0
Dinf	20	20	0	20	0	0	0

20: "printmat from printmat" 20

! stat	max	mean	SD	<100	100-1K	1K-10K	>10K
C1	22774	11960	6970	1	0	7	12
Cinf	146	113	30	4	16	0	0
D1	10	8	2	19	0	0	0

Dinf	5	4	1	19	0	0	0
arg 0							
D1	798	399	242	2	17	0	0
Dinf	39	28	8	19	0	0	0

This one may come as a serious shock to the programmer. In fact, the program spends almost as much time printing the result matrix as it does doing the actual multiplication. The reason is that, since the implementation does not have sharing analysis, the repeated appends in printmat produce many copies of the array. In this case, they are largely done in parallel, but very often production of the answers in this fashion can be a serious serial bottleneck in otherwise parallel programs.

Anyway, taking these figures at face value, they suggest that spawning is appropriate, and also suggest data sharing, since the output data structures are much smaller than the arguments.

21: "elemmult from rowmult" 400

! stat	max	mean	SD	<100	100-1K	1K-10K	>10K
C1	161	161	0	0	400	0	0
Cinf	101	101	0	0	400	0	0
D1	0	0	0	0	0	0	0
Dinf	0	0	0	0	0	0	0
arg 1							
D1	40	40	0	400	0	0	0
Dinf	20	20	0	400	0	0	0
arg 2							
D1	40	40	0	400	0	0	0
Dinf	20	20	0	400	0	0	0

22: "rowmult from rowmult" 400

! stat	max	mean	SD	<100	100-1K	1K-10K	>10K
C1	3156	1579	957	20	120	260	0
Cinf	158	124	32	20	380	0	0
D1	38	19	11	380	0	0	0
Dinf	19	9	5	380	0	0	0
arg 0							
D1	40	40	0	400	0	0	0
Dinf	20	20	0	400	0	0	0
arg 1							
D1	798	399	242	40	340	0	0
Dinf	39	28	8	380	0	0	0

This final case is similar in structure to the matmult/rowmult case, but less clearcut. Should we try to parallelise just one level of the multiplication, or two? The answer is, in part, that it depends on the machine. The hope is that an "expert system" with knowledge of the machine parameters, should be able to make this decision better than a programmer trying to decide how to annotate these callpoints.

Generalized Iteration Space and the Parallelization of Symbolic Programs (Extended Abstract)

Luddy Harrison

October 15, 1991

Abstract

A large body of literature has developed concerning the automatic parallelization of numerical programs, and a quite separate literature has developed concerning the parallelization of symbolic programs. Because many symbolic programs make heavy use of array data and iterative constructs, in addition to more "symbolic" language features like pointers and recursion, it is desirable to fuse these bodies of work so that results developed for numerical programs can be applied to symbolic ones, and generalized so that they apply to the variety of language constructs encountered in symbolic computations. In this paper is described a framework, called *generalized iteration space*, that allows one to unify dependence analysis of array computations with dependence analysis of pointer computations. It is shown that subscripted array accesses as well as pointer dereferences can be seen as linear functions of generalized iteration space. We are applying this framework to the automatic parallelization of C and Lisp programs in two parallelizing compilers at CSRD, called Parcel [Har89] and Miprac [HA89].

1 Dependence Analysis of Numerical Programs

A few, quite simple ideas form the basis for virtually all dependence testing methods for numerical programs. (By numerical programs is meant those that consist mainly of loops that manipulate arrays of numbers.)

We begin with a definition of dependence, which is ordinarily something like this:

> A *dependence* exists between two memory accesses if both of them access the same location in memory and at least one of them modifies the location.

When the location accessed has a simple, unambiguous name like **x** then the dependence test is trivial. A dependence obviously exists between two statements like

```
x = 10
  ⋮
y = x
```

by the definition.

When the location accessed is an array element the dependence test is subtler. If the accesses occur in a loop, we have a situation like this:

```
do i = 1 to 100
   a[f( i)] = R          S1
        ⋮
   L = a[g( i)]          S2
```

The question is, are there values $1 \leq i, i' \leq 100$ such that $f(i) = g(i)$? Suppose that i and i' are found that satisfy this equation. Then there is a dependence between these statements. We might be more precise and attach a *direction* to the dependence. This direction is simply the sign of $i' - i$. We say that the dependence has direction ">","<", or "=" according to whether $i' - i$ is positive, negative, or zero.

We may have several loops surrounding the references, in which case i and i' become vectors $\bar{i}$ and $\bar{i'}$ of index variables, one index variable per loop

surrounding each reference. $f(i)$ and $g(\bar{i}')$ are then expressions involving several index variables rather than just one.

Now, if f and g are linear functions of $\bar{i}$ and $\bar{i}'$, then linear (or integer) programming can be used to decide if $f(\bar{i}) = g(\bar{i}')$ has a solution. If this is unacceptably expensive, then an approximation to linear programming, like Banerjee's test [Ban79], can be applied instead (at the cost of some precision).

This has been a most dreadful compression of a very lively area of research. For a complete treatment, the reader is urged to see [Ban86], [ZC90], [PW86], [TIF86], [Wol82].

2 Iteration Space = Time

Consider a program that consists of a single nest of n `do` loops. A vector $\bar{i}$ that denotes a particular setting of the index variables of the loops, defines a point in a multidimensional space (n dimensions), called the *iteration space*. For example, if the index variables, from outermost to innermost, are `i1`, `i2` and `i3`, then $\bar{i} = \langle 2, 7, 3 \rangle$ would be the point corresponding to the second iteration of the outer loop, the seventh iteration of the middle loop, and the third iteration of the inner loop. We could identify $\bar{i}$ as a point in *time* during the execution of the loop. If `S` is a statement in the inner loop, then $\mathtt{S}_{\langle 2,7,3 \rangle}$ is the instance of `S` (a particular execution of the statement `S`) that occurs at time $\langle 2, 7, 3 \rangle$. The iteration space is totally ordered, so that it makes sense to say that $\langle 2, 7, 3 \rangle$ is earlier than $\langle 2, 8, 1 \rangle$.

This way of marking time is well defined independently of the iteration variables `i1`, `i2` and `i3`. That is, it would still make sense to speak of the point in time $\bar{i} = \langle 2, 7, 3 \rangle$ even if there were no variables `i1` = 2, `i3` = 7 and `i3` = 3 visible to the programmer. $\bar{i}$ would be the point in time at which control had entered the header of the inner loop for the third time, after having entered the header of the middle loop for the seventh time, after having entered the outer loop for the second time. Note how the nesting of the loops is built into the iteration vector: each time the outer loop is entered, the counters of the inner loops are reset to zero, like the digits of an odometer.

Looked at in this light, $f(\bar{i})$ and $g(\bar{i}')$ are functions of time (the iteration space) rather than of variables in the program. The dependence test answers this question: are there times at which `a`$[f(\bar{i})]$ and `a`$[g(\bar{i}')]$ are the same

memory location? The research on dependence testing of numerical programs to date lets us say this:

> When the memory locations that a program accesses are a linear function of time, then its dependences may be decided accurately at compile-time.

3 What Makes a Symbolic Program Non-numerical?

In Section 1 the theory of dependence testing of numerical programs was summarized. What is it about symbolic programs that makes it difficult to apply this theory directly to them? Apparently symbolic programs don't use so much a *different* set of programming language constructs, as a *larger* set. Cursory examination of the source code of a compiler or a Unix utility or a computer algebra package will turn up plenty of `do` loops and arrays. However, there is more variety: data is organized in linked structures and arrays (and in mixtures of the two) and `while` loops and procedure calling (including recursion) are used to control the execution.

We would be making great progress toward parallelizing symbolic programs if we could make a similar statement for them as for numerical programs; namely that we can analyze their dependences accurately when the locations they access are a linear function of time. To accomplish this, we apparently need two things: a notion of time that pertains to recursive procedures and `while` loops as well as to `do` loops, and a way of looking at the locations accessed by pointer dereferencing as functions of time (and often, we hope, linear functions of time), analogously to the way we viewed subscripts into arrays.

4 Generalizing the notion of time to arbitrary control structures

As was pointed out in Section 2, an iteration space is well-defined quite apart from the index variables of a loop nest. We may look at our iteration vector $\vec{i}$ as simply a *count* of the number of times control has passed into

each loop header (since the last time control exited that loop altogether). Just as easily, in a program with procedure calls and `while` loops, we may speak of a vector $\bar{j}$ where each element of $\bar{j}$ is associated with each procedure entrance and `while` loop header, and the count associated with each control flow point is incremented when control passes into that point. In this way, we define a natural generalization of the iteration space, and thus a natural generalization of our notion of time.

By recursion a loop may become nested within itself, or the outer loop in a nest may become nested within the inner loop (by a call to the procedure containing the loop nest, from the body of the inner loop). We must therefore take care to define this generalization of iteration space in a way that preserves the old definition but handles the new situation appropriately. Suppose we look at a `do` loop as a tail-recursive procedure `L`. There would then be two points in the program text where `L` is called: one outside of `L` (to initiate the loop) and one in the body of `L` (to invoke the next iteration). When control exits the final iteration of `L` it causes all the iterations to exit (in reverse order) up to the last call to `L` from the outside. If we make the rule that the count in $\bar{j}$ associated with `L` equals the number of active instances of the body of `L`, then it is easy to see that the iteration vector $\bar{j}$ for a simple nest of `do` loops is built exactly as $\bar{i}$ was before. To verify this, let the nest have three tail-recursive procedures, `L1`, `L2` and `L3`. There is a call to `L3` (from the outside) in the body of `L2`, and another in the body of `L3` itself, and likewise for `L2` and `L1`. The critical point is this: every time n iterations of `L3` execute, they cause the the count associated with `L3` to be incremented until it reaches n. When the n^{th} iteration exits, the count is decremented by 1 (there is one fewer activations of `L3`), and likewise for the $n - 1^{\text{th}}$ iteration, and so on until the call to `L3` from the outside is exited, at which point the count associated with `L3` is reset to zero. We are then ready for a fresh instance of `L3`, in the next iteration of the surrounding loop (`L2`).

We've established that our new iteration vector $\bar{j}$ is built, in the case of a simple nest of `do` loops, in a way that preserves the standard definition of iteration space. However, $\bar{j}$ is well-defined for any point in time during the execution of a program that consist of procedures that call one another (whether they represent loop bodies or are ordinary procedures). This is a most useful fact, which can be turned into an interprocedural dependence test for a large class of programs.

Before moving on, let us make two observations about this generalized

iteration space. First, consider this program:

```
L1:     do i = 1 to 100
           a[2i + k] = b[3i + k]
```

If this is our program in its entirety, then the corresponding iteration vector $\bar{j}$ has one element corresponding to the iterations of `L1`. The subscript `a[2i + k]` is linear in $\bar{j}$ (since `k` is fixed). However, suppose that `L1` is instead in the body of a procedure `f` and that elsewhere in the program appears the code

```
L2:     do k = 1 to 100
                call f
```

where `k` is the location in `L1` and `L2`. Now $\bar{j}$ has three elements, for `L2`, `f` and `L1` respectively. The subscript `a[2i + k]` is still linear in $\bar{j}$. If $\bar{j} = \langle 4, 1, 7 \rangle$ for example, then `i` $= \langle 0 \cdot 4 + 0 \cdot 1 + 1 \cdot 7 \rangle$ and `2i` $= \langle 0 \cdot 4 + 0 \cdot 1 + 2 \cdot 7 \rangle$ and `2i + k` $= \langle 1 \cdot 4 + 0 \cdot 1 + 2 \cdot 7 \rangle$.

Indeed, the program might be such that `f` is recursive, and that the subscript expression is a function of the control flow through `f` as well. In this way, we can reason about dependences for larger and larger units of control flow in the program.

The point of this example is to show that interprocedural dependence testing is greatly simplified when all of the memory locations accessed by the program are referred to a single standard of time (the generalized iteration space).

The second observation is this: the points in the generalized iteration space as we have defined it do not name unique points in time during the program execution. For example, it may happen that the body of `L2` contains two calls to `f`:

```
L2:     do k = 1 to 100
                call f
                call f
```

in which case the iteration vector $\bar{j} = \langle 4, 1, 7 \rangle$ could occur more than once. This is not a problem, as long as we are mindful of the fact, but it is also easy to remedy this by using *call sites* rather than procedures as the control flow points that represented by elements of the iteration vector $\bar{j}$. See also [Har89] for a different notion of interprocedural time, that gives rise to unique names for points during the execution.

5 Generalized Induction Variables

A do loop has two aspects: repetitive control flow and an index variable that is a linear function of the control flow. This linkage is most convenient, because it means that subscript functions which are a linear function of index variables are also a linear function of the iteration space. Indeed, when we normalize do loops so that their index variables run from 1 to an upper bound by a step of 1, we are making this fact explicit. It is the fact that they are functions of a single iteration space that makes it possible for us to compare different subscript functions and test for their overlap.

When our program contains **while** loops and procedure calls, the relationship between the variables of the program and its iteration space is not so manifest; we must work to establish that certain variables are linear in the control-flow of the program. Space does not permit a description here, but it is quite easy to develop a flow analysis (an abstract interpretation if you like) that expresses variables and other quantities in a program as linear functions of the generalized iteration space, and experimentation reveals that this is quite robust in practice. Let us take it for granted, then, that we can establish with good accuracy when an arithmetic variable **x** is a linear function of $\vec{j}$, and what constant coefficients must be attached to each element of $\vec{j}$ to give the value of **x** at any program point.

With such a result we obtain at once an interprocedural dependence test for array accesses, in the setting of recursive procedures and **while** loops. For example, the program above that has L2, f and L1 could just as easily have been written as three tail-recursive procedures by the the programmer. Once we establish (by flow analysis) that $\mathtt{k} = \langle 1,0,0\rangle \cdot \vec{j}$ and $\mathtt{i} = \langle 0,0,1\rangle \cdot \vec{j}$ and therefore that $2\mathtt{i} + \mathtt{k} = \langle 1,0,2\rangle \cdot \vec{j}$, then testing for dependences between "iterations" of L1 or L2 is straightforward. This is already an important step toward parallelizing symbolic programs effectively, because it allows us to apply many of the techniques developed for Fortran programs in the setting of more general control-flow constructs.

6 Birthdates of Dynamically Allocated Data

Recall that in section3 it was claimed that we needed two things to apply the (conventional) theory of dependence testing in the symbolic setting: a

generalized notion of iteration space, and a way of looking at any memory access (whether array or pointer) as a function of that iteration space. At this point we've established the new iteration space and claimed that it is easy to express ordinary induction variables and linear combinations of them as linear functions of this generalized iteration space. It remains for us to show a correspondence between an arbitrary memory access and this iteration space.

Suppose that an object is allocated (as by `malloc`) at a point S in the text of a program, and at time $\bar{i}$ (the *birthdate* of the object). We may identify $\langle S, \bar{i} \rangle$ as the address of the object; it distinguishes the object from all others (let's assume that $\bar{i}$ is unique during the execution of the program). Now suppose that $\langle S, \bar{i} \rangle$ is but one of many objects allocated at the program point S during the execution (that is, there are other instances of S), and that each such object points to another in a chain. Let $\langle S, \bar{i}' \rangle$ be the successor of $\langle S, \bar{i} \rangle$ in this chain. If $i' - i$ is a constant vector (for all pairs of objects $\langle S, \bar{i} \rangle$ and $\langle S, \bar{i}' \rangle$ that are neighbors in the chain), then the memory locations of this chain are a linear function of time. This will happen, for example, if the $\langle S, \bar{i} \rangle$'s form a linked list or doubly linked list constructed by the iterations of a loop or tail-recursive procedure, and a similar pattern would arise if the $\langle S, \bar{i} \rangle$'s formed a tree built by a recursive procedure or even by several mutually recursive procedures.

Now, $\bar{i}$ and $\bar{i}'$ are two points in time during the execution of the program, and it might not be useful or practical to compute them at compile time, for there may be an infinitude of such vectors. (Similarly, it is not ordinarily useful or practical to manipulate the individual iterations of a nest of do loops in conventional dependence testing.)

However, if $\bar{i}' - \bar{i}$ is a constant vector, it is very reasonable to compute its value during compilation. We could do it this way: when our object is allocated at point S, attach to it the vector $\bar{j} = \bar{0}$ (all zeroes). Then, as the object undergoes movements of control flow (that is, as procedures are called and returned from) we increment and decrement the elements of $\bar{j}$ so that the net effect of these movements are recorded in $\bar{j}$. When we reach S again and allocate another object, and link it to the object to which we attached $\bar{j}$, we will have $\bar{j} = \bar{i}' - \bar{i}$. That is, $\bar{j}$ will be the time between the birthdate of the two objects, by its construction. Thus we can compute $\bar{i}' - \bar{i}$ (provided that it is a constant vector; that is, provided that it is independent of the choice of $\bar{i}$ and $\bar{i}'$) with computing any $\bar{i}$ or $\bar{i}'$.

7 Dependence Testing over Birthdates

By the foregoing method we may obtain a program analysis that attaches to every pointer, the difference in time between the birthdate of the object that contains the pointer and the object to which it points.

Now consider a traversal during which the data structure created above is modified. Suppose it has this form:

```
F:        ⋮
      ptr->info = 0
   ptr = ptr->link
      call F
```

and consider the variable `ptr`. Suppose that its value is $\langle S, \bar{i_0} \rangle$ at the beginning of `L`. After the first iteration of `L`, `ptr` becomes $\langle S, \bar{i_0} + \bar{j}$, and after the second iteration, $\langle S, \bar{i_0} + 2 \cdot \bar{j}$, and so on. By the same means that we may recognize when an arithmetic value is a linear function of the iteration space, we may recognize that `ptr` is a linear function of the iteration space, and thus that the memory accesses `ptr->info = 0` are a linear function of the iteration space. In this case we have a natural dependence test, namely, to show that there is no $\bar{j} \neq \bar{j'}$ such that $\bar{i_0} + \bar{j} = \bar{i_0} + \bar{j'}$. This is precisely the sort of equation that arose when we were testing for dependences among subscripts, and can be solved by exactly the same means (linear or integer programming or an approximation thereto). It is straightforward to generalize this idea to nests of loops and recursive procedures operating over linked data.

In the example I've chosen the traversal of the data structure is isomorphic to its construction (that is, the control flow during the construction is isomorphic to the control flow during its traversal) but the dependence test would apply if the traversal were

```
F:        ⋮
      ptr->info = 0
ptr = ptr->ptr->link
      call F
```

In this case the equation to be satisfied would be $\bar{i_0} + 2 \cdot \bar{j} = \bar{i_0} + 2 \cdot \bar{j'}$ subject to $\bar{j} \neq \bar{j'}$. Here, the $2 \cdot \bar{j'}$ comes from the addition of j' twice (once as each `link` is crossed).

There is therefore no requirement that the construction and traversal be isomorphic, only that they be linear in time as we have defined it.

8 Mixed Array and Pointer Dependence Testing

If it is true as claimed, that symbolic programs make heavy use of arrays and conventional iteration in addition to recursion and linked structures, then it is important to observe that both subscripted arrays and pointers are viewed in this framework as functions of a single iteration space. Suppose we see two accesses like `foo->a[2i+7]` and `bar->a[3j+2i]`. Suppose further that we obtain equations d_1 and d_2 for the birthdates of `foo` and `bar` as we did for `ptr` above, and that we obtain equations s_1 and s_2 for the subscripts `2i+7` and `3j+2i` as we did for the subscript `2i+k` above. Then there is a dependence between these references only if there is a solution to

$$d_1(\bar{i}) = d_2(\bar{i'}) \text{ and } s_1(\bar{i}) = s_2(\bar{i'}).$$

That is, the two equations must be satisfied by a single pair of time points $\bar{i}$ and $\bar{i'}$.

9 Related Work

It has been the goal of this paper to discuss the relationship between dependence testing in numerical programs and the dependence testing of symbolic programs used in Miprac. A complete comparison of the dependence analysis used in Miprac to other methods for analyzing symbolic programs (e.g., [LH88],[HN90],[Gua87],[CWZ90]) can't be given here since the dependence analysis algorithm used in Miprac has not been presented completely here. The interested reader may obtain [Har91] which contains a lengthy comparison of Miprac's methods to other research in the parallelization of symbolic programs.

10 Conclusion

It is possible to generalize the notion of iteration space and linear memory access so that they apply usefully to the dependence testing of symbolic programs. We have constructed one experimental compiler (Parcel) that applied this framework to Scheme programs with good success, and are currently building a more powerful and flexible system (Miprac) that applies this framework to C, Fortran, and Common Lisp programs.

References

[Ban79] Uptal D. Banerjee. *Speedup of Ordinary Programs.* PhD thesis, University of Illinois at Urbana-Champaign, October 1979.

[Ban86] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer, Boston, MA, 1986.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, 1990.

[Gua87] Vincent Antony Guarna. Analysis of c programs for parallelization in the presence of pointers. Technical Report 695, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1987.

[HA89] W.L. Harrison III and Z. Ammarguellat. The design of parallelizers for symbolic and numeric programs. In Takayasu Ito and Robert Halstead, editors, *Parallel Lisp: Languages and Systems (US/Japan Workshop on Parallel Lisp, Sendai, Japan, June 1989)*, pages 235–253. Springer-Verlag, June 1989. Lecture Notes in Computer Science #441.

[Har89] W.L. Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation: an International Journal*, 2(3/4):179–396, 1989.

[Har91] W.L. Harrison III. Pointers, procedures and parallelization. Technical Report (Work In Progress), Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, October 1991.

[HN90] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, January 1990.

[LH88] J. Larus and P. N. Hilfinger. Restructuring lisp programs for concurrent execution (summary). In *Conference Record of the ACM SIGPLAN Symposium on Parallel Programming*, 1988.

[PW86] D.A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12), December 1986.

[TIF86] Remi J. Triolet, Francois Irigoin, and Paul Feautrier. Direct parallelization of call statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 176--185, 1986.

[Wol82] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, October 1982.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Supercomputers.* Frontier Series. ACM Press, 1990.

Dataflow Analysis of Concurrent Logic Languages

Ian Foster, Argonne National Laboratory
Will Winsborough, Penn State University

We present a framework for the definition and verification of static analyses of concurrent logic programming languages. We illustrate the framework by constructing an analysis to recognize consumers that receive data structures that have no other consumer. Such information enables a compile-time decision to reuse data-structure storage when the consumer is done with it. The principal feature of the analyses constructed using our approach is that they need not simulate directly the numerous possible interleavings of process reductions that can occur during execution of a parallel program. This feature is intended to make our analyses affordable.

Compiler Support for the Refinement and Composition of Process Structures

Ian Foster, Argonne National Laboratory

We present recent work concerned with the specification of spatial organization in parallel programs. First, we describe linguistic constructs (defined as extensions to the parallel programming language PCN) that allow specifications for the logic and physical layout of a parallel program to be developed simultaneously, in the same stepwise refinement process. These constructs (described in the attached paper) provide a natural framework for the composition of programs defined in terms of specialized topologies, such as an FFT on a linear array. They also allow us to untangle the problems of specifying layout and program logic: the layout hierarchy can be modified independently of program logic to alter resource allocation decisions, changing performance but not correctness.

Second, we describe the compiler techniques used to implement these constructs on distributed memory computers. Source-to-source transformations are used to translate programs augmented with the mapping constructs into simpler programs augmented with calls to mapping libraries. These mapping libraries encode expertise about optimal embeddings of complex topologies in particular physical machines. A novel aspect of the compiler is its use of a programmable transformation system called Program Transformation Notation (PTN). This provides a metalanguage for specifying program transformations.

On the Refinement and Composition of Process Structures

Ian Foster
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA

1 Introduction

Refinement, *hierarchy*, *composition*, and *reuse* are four related ideas that together form a basis for good program design [15, 18]. These ideas are well-understood and frequently applied in sequential programming. Unfortunately, although the ideas are in principle directly applicable in parallel programming [2], in practice they are rarely used when designing programs for large parallel computers. In particular, it is rare to find genuinely reusable libraries for MIMD parallel computers.

We believe that this situation is explained in part by the fact that components of parallel programs must often be regarded as having an extent in space as well as in time. Concurrent programming notations typically allow the specification of temporal organization (e.g., do A and B concurrently, then do C), but not spatial organization (e.g., do A in half of the machine, B in the other half, and C everywhere). Hence, the spatial organization or *layout* of a program must be managed in an ad-hoc manner, often by writing code to manipulate machine addresses. Programs expressed in such low-level terms must be modified to execute in a different configuration or in a subset of the available processors, hence, they are not easily integrated into larger programs.

We address this problem by extending the parallel program design process to allow the logic and layout of a program to be developed in a single refinement process. Recall that in the stepwise refinement methodology, a problem is successively decomposed into subproblems in order to untangle seemingly interdependent aspects of the design. Each refinement step generates code defining subprograms introduced in previous decompositions. In order to permit the refinement of spatial organization, we reify the topology in which a program executes. A program executes initially in a topology corresponding to the underlying physical machine. At each refinement step, the programmer has the option of associating refined topologies with subprograms introduced by refinement. A refined topology is obtained by applying a specified remapping operation to the current topology. This operation may, for example, alter the spatial organization of the nodes in the topology or restrict subsequent execution to a subset of these nodes. In this way, a hierarchy of *virtual topologies* is developed, complimenting the hierarchical structure of the program logic.

This approach permits a methodical development of efficient spatial organizations in machines for which the physical organization of computations is important. This is the case, for example, in networks and parallel computers with non-uniform memory access times. However, we claim that the underlying concepts have deeper significance: they provide a natural framework for the composition (and hence reuse) of parallel programs on *any* MIMD parallel computer. They permit programs defined in terms of specialized topologies (e.g., an FFT on a linear array, a reduction operator on a mesh, or a self-scheduling structure [5]) to be integrated painlessly into larger programs. Topologies also provide a natural context for the integration of data-parallel operations into a MIMD framework: subsets of available computational resources can be defined, and workers replicated within subsets

to perform computations. Finally, the approach allows us to untangle the problems of specifying mapping and program logic: the mapping hierarchy can be modified independently of program logic to alter layout or resource allocation decisions, changing performance but not correctness.

To permit a practical vehicle for experimentation with these ideas, we have defined and implemented a set of extensions to the concurrent programming notation PCN [3]. The extensions include annotations for specifying refinement of topologies and replication of computations within topologies. The utility of replication is enhanced by a linguistic construct called the *port*, a distributed data structure used to establish communication links between processes executing within a virtual topology. The implementation is integrated with an existing PCN compiler for multicomputers, multiprocessors, and networks [8]. The extended language has been applied to a variety of programming problems, and has proved particularly useful in a project developing and evaluating algorithms for use in numerical simulations of climate (e.g., [6]). The design and implementation of such algorithms is complicated by a spherical problem domain and widespread use of implicit methods with complex communication structure. The use of virtual topologies has simplified the specification of algorithms and permitted the reuse of core components.

The work reported in this paper provides for the first time a truly general framework for the specification of spatial organization of programs. In addition, it significantly extends the range of parallel programming problems to which refinement and composition techniques can usefully be applied. In particular, it makes it possible to define genuinely reusable libraries for MIMD parallel computers.

Virtual topologies are often used to provide a more convenient view of a parallel computer (e.g., [10, 17]). However, these mappings typically apply to the whole computer and hence do not simplify composition. Our use of ports is analogous to the use of pins to compose packages in VLSI [13]. The term port has also been used to describe constructs used to establish interprocess communications in ensembles [9]. A similar idea underlies Browne *et al.*'s CODE environment, which allows the definition of "software chips" connected by "pins" [1]. However, these proposals include no notion of virtual topologies or of replication of ports over a restricted extent, and hence are more limited in their applicability. Hudak and Kelly define functional notations for specifying process mapping and interconnection [11, 12]. However, these proposals do not support hierarchical structures, replication, or the composition of process networks.

The rest of this paper is structured as follows. In Section 2, we introduce the notion of topology and define useful properties of topologies. In Sections 3 and 4, we describe an integration of topologies into a programming notation, and demonstrate the use of the notation in examples. In Sections 5 and 6, we outline implementation techniques and contrast our ideas with other related proposals.

2 Topologies

We first define a topology, the map functions used to construct new topologies, and certain useful properties of topologies.

Topology. A topology is a collection of computing sites or *nodes*. We represent a topology X by a $< D, T >$ pair where the *domain* $X.D$ is a vector of node names and the *type* $X.T$ indicates how the nodes are organized in space. For example:

X_0 = <<sun1, sun2, sun3>, array(3)> (a set of workstations)
X_1 = <<p0,...,p527>, mesh({17,32})> (a 528-node mesh)

We refer to topologies such as these, in which the domain contains physical names, as *physical topologies*. It is also possible to define *virtual topologies*, in which the domain is a vector of integer indices into the domain of another physical or virtual topology.

Maps. A map function M takes a topology X and generates a new *virtual topology* X', in which the domain $X'.D$ is a vector of integer indices into $X.D$. Maps can transform topologies in a variety of ways. We distinguish three general classes of transformation:

Reshaping (or aliasing): The domain of the new topology is reordered and/or its type is changed. For example, M_a reorganizes the nodes in a topology X (which must be an 8-node array) as a 4×2 mesh:

$$\begin{array}{lll} M_a & : & X \rightarrow X', \\ & \text{if} & X.T = \text{array}(8), \\ & \text{where} & X' = << 0, ..., 7 >, \text{mesh}(\{4,2\}) >. \end{array}$$

The $0, ..., 7$ in $X'.D$ are the indices of the eight nodes in X.

Restriction: The new topology includes only a subset of the nodes in the parent topology. For example, M_b and M_c define different embeddings of a 2×2 submesh in a 4×2 mesh:

$$\begin{array}{lll} M_b & : & X \rightarrow X', \quad M_c : X \rightarrow X'', \\ & \text{if} & X.T = \text{mesh}(\{4,2\}), \\ & \text{where} & X' = << 0,1,2,3 >, \text{mesh}(\{2,2\}) >, \text{ and} \\ & & X'' = << 0,2,4,6 >, \text{mesh}(\{2,2\}) >. \end{array}$$

Expansion: The new topology embeds more than one node in each node of the parent topology. For example, M_d embeds an 8-node array in a 2×2 mesh by creating two array nodes in each node:

$$\begin{array}{lll} M_d & : & X \rightarrow X', \\ & \text{if} & X.T = \text{mesh}(\{2,2\}), \\ & \text{where} & X' = << 0,0,1,1,2,2,3,3 >, \text{array}(8) >. \end{array}$$

Maps can be composed. We say that a topology U, obtained by the application of the composition of a series of map functions to a topology X, is *derived from* X. For example, the topology $B =<< 0,0,1,1,2,2,3,3 >, \text{array}(8) >$ can be derived from a topology $A =<< d_0, ..., d_7 >, \text{array}(8) >$ by application of the maps $M_d \circ M_b \circ M_a$ or $M_d \circ M_c \circ M_a$.

Properties of Topologies. In the following, let X be a topology and let $\mathcal{U}$ represent the topologies U_1, ..., U_k, all derived from X.

1. $\texttt{indices}(X) = \{0, ..., \mid D \mid\}$.
2. $\texttt{nodes}(X, I) = \{X.D_i \mid i \in I\}$
3. (Extent) Consider a topology V derived from a topology X by the composition of maps M_1, ..., M_n. Name the intermediate topologies $X_1, ..., X_{n-1}$, where $X_i = M_i(X_{i-1})$, $0 < i \leq n$, with $X_0 = X$ and $X_n = V$. Define $I_i = \texttt{nodes}(X_i, I_{i+1})$, $1 \leq i < n$, and $I_n = \texttt{indices}(V)$. Define the *extent* of V in X as:

   $$\texttt{extent}(V, T) = \texttt{nodes}(X, I_1)$$

   Informally, the extent of V in X is the set of nodes in X that contain nodes in V.
4. $\mathcal{U}$ are *disjoint* in X if $\texttt{extent}(U_1, X) \cap ... \cap \texttt{extent}(U_k, X) = \phi$.
5. $\mathcal{U}$ *cover* X if $\texttt{extent}(U_1, X) \cup ... \cup \texttt{extent}(U_k, X) = \texttt{indices}(X)$.
6. $\mathcal{U}$ are *1-to-1* in X if $\texttt{extent}(U_1, X) \cup ... \cup \texttt{extent}(U_k, X)$, when constructed as a multiset, has no duplicate elements.
7. $\mathcal{U}$ are *co-extensive* in X if $\texttt{extent}(U_1, X) = ... = \texttt{extent}(U_k, X) \subseteq \texttt{indices}(X)$.

3 Integration in a Programming Notation

We are interested in concurrent programming notations in which refinement and composition are supported in a useful manner, and in which it is possible to achieve a separation of concerns between program logic and process mapping. Strand [7] and PCN [3] both meet these requirements. We choose to work with PCN here.

The Notation. A PCN solution to a programming problem is a set of programs, each with the general form:

```
name(arg1,...,argk)
declaration1, ..., declarationm;
{ op prog1, ..., progn}
```

where $k, m \geq 0$, $n > 0$ and op is one of "||", ";", or "?", indicating that the program calls $\texttt{prog}_1$, ..., $\texttt{prog}_n$ are to be executed concurrently, in sequence, or as a set of guarded commands, respectively. The program calls may invoke either other PCN programs or procedures in sequential languages such as Fortran and C.

Program calls composed with the parallel operator || interact by reading and writing shared single-assignment variables. As in Strand and related dataflow languages, these variables are initially undefined, can be written once, and once written cannot be modified. An attempt to read an undefined variable suspends until a value is provided.

PCN is supported by a compiler and run time system which ensure that program calls in parallel compositions execute correctly wherever they are located. In particular, it is possible to map all program calls to a single processor or to randomly selected processors.

Program development in PCN typically proceeds via a sequence of refinement steps. For example, when specifying a hierarchical manager/worker scheduler, we may indicate in a first program that the scheduler is structured as a manager and two subschedulers:

```
scheduler()                                   /* Program (1) */
{|| manager(s1,s2),
    subscheduler(s1),
    subscheduler(s2)
}
```

This program defines three concurrent program calls (which we may think of as processes): a **manager** and two **subschedulers**, connected by shared variables **s1** and **s2**. We may then refine the definition of **subscheduler** to indicate that it consists of a submanager and some number of workers. Further refinement steps provide definitions for the worker and manager programs.

Refining Topologies. Code such as Program (1) specifies the creation of sets of processes but does not indicate how these processes are to be organized in a parallel computer. Experience suggests that this decision is in general sufficiently difficult to warrant programmer intervention. We introduce linguistic constructs which permit the programmer to develop a specification for the spatial organization of a program.

We assume that every program executes within a context: a $< X, i >$ pair, where X is a topology and $i \in \texttt{indices}(X)$ represents the node in X on which the program is executing. By default, new program calls in program definitions introduced during refinement execute in the same context as their parent program. However, the programmer also has the option of specifying that subprograms should execute in a new context, obtained from the parent context by the application of a *relocation*, *remap*, or *replication* operator.

Relocation: A relocation operation causes a program to execute on a different node within the same topology. The new node is a function of the current context:

$$\text{Relocate} : <X, i> \rightarrow <X, R(X, i)>,$$

where R is the relocation operator.

We represent relocation in PCN by the infix operator `@`. For example, if the current context is an array, we may locate the `manager` program on the 0th node by writing:

```
manager(s1,s2)@0
```

Remapping: A remapping operation causes a program call to execute within a new topology derived from the current context's topology by the application of a map function M (Section 2):

$$\text{Remap} : <X, i> \rightarrow <M(X), 0> .$$

The remapping may reshape, restrict, and/or expand the current topology. We represent remapping in PCN by the infix operator `in`. For example, if the map `subarray(i,I)` yields the `i`th of `I` disjoint subarrays in an array topology, then to locate the two `subscheduler` programs in disjoint subarrays, we write:

```
scheduler()                                   /* Program (2) */
{|| manager(s1,s2)@0,
    subscheduler(s1) in subarray(0,2),
    subscheduler(s2) in subarray(1,2)
}
```

Replication: A replication operation invokes a program call on each node of the current context's topology:

$$\text{Replicate} : < X, i > \rightarrow < X, 0 >, ..., < X, |X.D| - 1 > .$$

We represent replication in PCN by quantification over the indices of the topology. For example, we may specify that a subscheduler is to comprise one submanager (on node 0) and a number of workers (one per other node) as follows:

```
subscheduler(s)
{|| i over 0..nodes()-1 : i==0 -> submanager(s,...)@0,
                          i!=0 -> worker(...)@'i'
}
```

As this program shows, replication allows us to define *self-sizing* programs that adapt automatically to available resources: `subscheduler` populates whatever topology is specified in the calling program. This permits resource allocation decisions to be decoupled from problem solving logic. For example, removing the `in` annotations from Program (2) changes the behavior of the subprograms (both execute throughout the entire array) but does not require changes to `subscheduler`.

Self-sizing can also be used to control the amount of internal concurrency in a processor: a remapping is used to expand the number of nodes in a topology by some chosen factor. This provides more processes per processor, hence permitting overlapping of computation and communication.

Adaptive Mapping. The functions `topology()` and `size()` allow programs to define layouts that are specialized to particular topologies.

`topology()` : Returns a {*type*, *size*} term representing the current context's topology.

`nodes()` : Returns the number of nodes in the current context's topology.

For example, we may have implemented efficient embeddings of a program for mesh and array topologies. We also want our program to execute (perhaps suboptimally) in any other topology. Hence, we specify that our application should utilize the mesh embedding in a mesh topology; any other topology is remapped to an array:

```
program()
{ ? topology() ?= mesh(_) -> spawn_in_mesh(),
    default -> spawn_in_array() in array
}
```

The concepts presented in this section permit a methodical development of declarative specifications for resource allocation decisions, and permit these decisions to be decoupled from problem-solving logic. In addition, they make it possible to specify the composition of parallel programs, even when the programs being composed are defined in terms of alternative topologies. For example, libraries containing a parallel FFT defined in terms of an array topology and a broadcast defined in terms of a mesh can be composed with a simple remapping. This is possible because layout within individual programs is specified relative to the current context rather than in absolute terms.

4 Communication in Process Structures

It is often necessary to establish communication channels between groups of processes when defining or composing process structures. The shared single-assignment variable provides a powerful mechanism for specifying communication and synchronization between individual processes. We define a linguistic construct called the *port* that provides the same functionality for process structures.

A port is a distributed data structure with a specified number of items in each node of the current topology. Each item is an ordinary single-assignment variable and can be used for communication and synchronization in the usual way. A port is declared with a `port` declaration as a 1-dimensional array with size determined by the declaration and the underlying topology. For example, we might write `port p[2];` to declare a distributed port structure `p` with two entries, `p[2*i]` and `p[2*i+1]`, on each node ($0 \leq i <$ `nodes()`) of the underlying topology.

We illustrate the use of ports to establish communication channels between co-extensive structures and within a process structure created by replication.

Co-extensive Structures. Consider the problem of developing a parallel PDE solver using domain decomposition techniques. At each step, we must perform a global reduction across all domains to find the maximum allowable time step, exchange boundary values between neighboring domains, and advance the solution in each domain.

An elegant solution to this programming problem is obtained by defining the solver as the composition of two simpler process structures: one for performing global reductions (`reduce`) and one for performing nearest-neighbor communications (`solve`). The `reduce` structure is a spanning tree with one leaf in each node of the current topology; each leaf expects to receive a stream of values from some other (anonymous) process, and responds to each such value by returning the result of the global reduction. The `solve` structure is a set of solvers, one per node of the current topology; each solver expects to be able to send values to an (anonymous) reducer and receive reduced values.

We require a mechanism for specifying that when composing `reduce` and `solve`, a communication channel should be established between the reducer/solver process pair that is created on each node. This is achieved by using a port. Neglecting for the moment the need to establish internal communications and process structure within `reduce` and `solve`, and assuming an array topology, we write:

```
pde_solver()
```

```
port p[1];
{|| reduce(p), solve(p)}

reduce(p)
port p[1];
{|| i over 0..nodes()-1 : reducer(p[i])@'i'}

solve(p)
port p[1];
{|| i over 0..nodes()-1 : solver(p[i])@'i'}
```

The pde_solver is defined as the composition of **reduce** and **solve**. The port **p** has one entry **p[i]** on each node of the current topology. Both **reduce** and **solve** replicate subprograms (**reducer** and **solver**) throughout the current topology. The **reducer** and **solver** processes created on the ith node are both given **p[i]** as an argument. This single assignment variable provides the required connection.

The role of ports in this example can be explained by an analogy with VLSI design. Think of the **solve** and **reduce** structures as VLSI cells. Each cell comprises some internal nodes and communication structure, plus a set of pins for connecting to other cells. The port construct provides a mechanism for specifying how pins in different cells are to be connected. In this case, each port in **solve** is connected to the corresponding port in **reduce**.

Replicated Structures. Ports can also be used to establish communications between nodes in replicated structures. For example, **solver** requires internal communication channels for the exchange of information between neighboring subdomains. If the solver employs one-dimensional domain decomposition and periodic boundary conditions, then the necessary internal communication links (two input and two output channels in each process) can be established by defining two additional ports, **l** and **r**:

```
solve(p)
port p[1], l[1], r[1];
{|| i over 0..nodes()-1 : solver(p[i],r[i],r[(i+1)%nodes()],l[i],l[(i-1)%nodes()])@'i'}
```

The internal structure of **reduce** can be developed in a similar way.

A port can also be used in **subscheduler** to establish communications between **workers** and the **submanager**. A port **s** is declared and each worker is assigned its local port **s[i]** as its communication channel to the submanager. The submanager itself is given the entire port as an argument.

```
subscheduler(...)
port s[1];
{|| i over 0..nodes()-1 : i==0 -> submanager(s)@0,
                          i!=0 -> worker(s[i])@'i'
}
```

5 Implementation Notes

The language extensions described in this paper are being implemented using source-to-source transformation techniques. PCN programs augmented with the additional constructs defined in previous sections are transformed to pure PCN and linked with libraries implementing embeddings. The transformations are specified with a programmable transformation system called Program Transformation Notation (PTN). To date, relocation and remapping have been implemented and applied to several geophysical modeling problems. Implementation of the replication constructs is ongoing.

The implementation is based on a small set of simple ideas. Topologies are represented at run-time by PCN process structures. Relocation within a topology is achieved by message passing within the process structure representing the topology. Messages are terms representing relocated program calls (e.g., the term `{"f",x}` represents the program call `f(x)`) and are interpreted as requests to initiate execution of the specified program call. This implementation of process migration is possible because the PCN run-time system provides access to variables (e.g., `x`) regardless of process location. Remapping is achieved by spawning a new process structure on top of the process structure representing the current topology. The spawning of the new structure is achieved by using relocation operations. Replication is implemented in a similar way.

It is important to understand that the techniques described in this paper do not impose any overhead on program execution. Process structures laid out using virtual topologies execute directly on the underlying hardware without any layers of run-time interpretation. Some overhead *is* incurred when laying out processes, as each virtual topology in a hierarchy must be created. However, this overhead can be avoided when necessary by providing libraries that embed virtual topologies directly in the underlying physical topology. This produces a "standalone program", so called because it invokes no mapping code at run time.

6 Related Work

Virtual Machines. Taylor proposes the *virtual machine* as a means of achieving architecture independence, scalability, and programming convenience on parallel computers [17]. This construct encourages the programmer to view a computer as an infinite computing surface with interconnections forming a linear array, mesh, etc. Computations are spawned on this surface by recursively-defined programs annotated with Logo-like mapping constructs (e.g., `@fwd`, `@bwd` in a linear array) [14, 16]. These annotations inspired the current proposal's relocation operators.

Virtual machines have proved extremely useful in several parallel programming systems [17, 7]. However, although the view of a computer as an infinite surface provides scalability, it makes it difficult to achieve an optimal granularity on a particular computer. In contrast, the current proposal permits precise fitting to the size of a particular machine. In addition, the notions of refinement and composition are absent from Taylor's work.

Ensembles. Griswold *et al.* propose an abstraction called an ensemble as a means of organizing data, computation, and communication in distributed memory computers [9]. They define mechanisms for mapping data and code to processors and linking ports in different processors to create a communication network. In this respect, an ensemble is much like a physical topology. In addition, different phases of a computation can employ different ensembles, providing a limited form of remapping. However, the ensemble concept does not support hierarchy, restriction, expansion, or relocation. No implementation has been reported.

CODE. Browne *et al.* propose a *calculus of composition* for parallel program components [1]. This allows programmers to define operators specifying interconnections between components called *software chips*. Connections express data dependencies and mutual exclusion dependencies rather than communication channels; nevertheless, there are many similarities between these connections and the port construct described in the present proposal. However, Browne *et al.*'s calculus is very abstract and they do not show how these ideas might be integrated into a programming notation. The notion of virtual topology is entirely absent.

Functional Programming. Hudak's *para-functional programming* permits programmers to control mapping by means of annotations on expressions [11]. As annotations can be arbitrary expres-

sions, some degree of abstraction is presumably possible. However, the notation does not admit notions of hierarchy, replication, or composition of process networks.

In Kelly's Caliban system, programmers can associate **moreover** clauses with function definitions to provide a declarative specification of processes and expected communication channels [12]. Constructs such as pipelines and meshs can be defined and reused. However, the goal of Kelly's proposal is not to specify process mapping but to provide information about expected communication patterns, for use by a compiler. No implementation has been reported.

Concurrent Aggregates. Chien and Dally describe a concurrent object-oriented language called Concurrent Aggregates (CA) [4]. They seek to remove the sequential bottleneck associated with message-passing in object-oriented languages by allowing the definition of homogeneous collections of objects called *aggregates*. A run-time system routes messages addressed to an aggregate to one of its members. In common with the current proposal, CA allows the definition of concurrent structures which can then be composed with other structures to build a concurrent program. However, issues associated with spatial organization of such structures are not addressed.

7 Conclusions

We have defined a framework within which the familiar ideas of refinement, hierarchy, composition, and reuse can be applied to the development of parallel programs. The key idea underlying this framework is the use of spatial layout as an organizational principle: a context in which to specify replication, composition, communication, and other important issues. We show that it is possible for a programming notation to support a refinement methodology in which a program's logic and spatial organization are developed concurrently. Refinement then produces a hierarchy of virtual topologies, complimentary to the hierarchical organization of the parallel program.

The approach has a number of important benefits. First, the ability to embed virtual topologies inside other virtual or physical topologies (by remapping, restriction, or expansion) makes it possible to build complex programs by the composition of simpler programs. This is the case even if subprograms are defined in terms of different topologies. Second, design decisions concerning resource allocation and spatial organization can be decoupled from program logic, as such decisions are expressed in the composition that combines subprograms, not the subprograms themselves. Third, virtual topologies allow the underlying physical topology to be either hidden or made visible at various levels of abstraction. This makes it possible both to achieve portability and to write programs that exploit aspects of the underlying hardware. Fourth, virtual topologies provide a convenient context for embedding various data-parallel operations in a MIMD context. For example, it is straightforward to define programs that perform reductions, broadcasts, or SIMD/SPMD computations within a topology.

The concepts and techniques presented in this paper form the basis for a larger project developing a template-based parallel programming environment. A template is a reusable component implementing a parallel program structure such as a domain decomposition strategy, parallel transform, or load balancing algorithm. Virtual topologies provide an elegant framework in which to discuss the definition, reuse, and composition of templates.

Acknowledgments

The development of these ideas benefited from discussions with K.M. Chandy. Thanks to Steve Hammond for his help in implementation.

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

References

[1] Browne, J., Werth, J., and Lee, T., Intersection of parallel structuring and reuse of software components, *Proc. Intl Conf. on Parallel Processing*, Penn State Press, 1989.

[2] Chandy, C., and Misra, J., *Parallel Program Design*, Addison-Wesley, 1989.

[3] Chandy, C., and Taylor, S., *An Introduction to Parallel Programming*, Jones and Bartlett, 1991.

[4] Chien, A., and Dally, W., Concurrent Aggregates, *Proc. ACM Symp. on Principles and Practice of Parallel Programming*, 1990, 187–196.

[5] Foster, I., Automatic generation of self-scheduling programs, *IEEE Trans. on Parallel and Distributed Systems*, Jan. 1991.

[6] Foster, I., Gropp, W., and Stevens, R., The parallel scalability of the spectral transform method, *Mon. Wea. Rev.*, March 1992.

[7] Foster, I., and Taylor, S., *Strand: New Concepts in Parallel Programming*, Prentice Hall, Englewood Cliffs, N.J., 1989.

[8] Foster, I., and Taylor, S., A compiler approach to concurrent program refinement (in preparation).

[9] Griswold, W., Harrison, G., Notkin, D., and Snyder, L., Port ensembles: A communication abstraction for nonshared memory parallel programming, *Proc. Intl Conf. on Parallel Processing*, Penn State Press, 1990.

[10] Ho, C.-T. and Johnsson, L., On the embedding of arbitrary meshes in Boolean cubes with expansion two dilation two, *Proc. Intl Conf. on Parallel Processing*, Penn State Press, 188–191, 1987.

[11] Hudak, P., Para-functional programming, *IEEE Computer*, 60–70, Aug 1986.

[12] Kelly, P., *Functional Programming for Loosely-Coupled Multiprocessors*, MIT Press, 1989.

[13] Mead, C., and Conway, L., *Introduction to VLSI Systems*, Addison Wesley, 1980.

[14] Pappert, S., *Mindstorms: Children, Computers, and Powerful Ideas*, Basic Books, New York, N.Y., 1980.

[15] Parnas, D., On the criteria to be used in decomposing systems into modules, *CACM* 15(12), 1053–1058, 1972.

[16] Shapiro, E., Systolic programming: a paradigm for parallel processing, *Proc. Intl Conf. on 5th Generation Computer Systems*, Tokyo, 458–71, North Holland.

[17] Taylor, S., *Parallel Logic Programming Techniques*, Prentice Hall, Englewood Cliffs, N.J., 1989.

[18] Wirth, N., Program development by stepwise refinement, *CACM*, 14, 1971, 221–227.

List of Contributors

K. Aida, Waseda University, Tokyo, Japan
Deb Banerjee, Dartmouth College, Hanover, New Hampshire
Arvind Bansal, Kent State University, Kent, Ohio
Ira Baxter, Schlumberger Laboratory for Computer Science, Austin, Texas
Marina Chen, Yale University, New Haven, Connecticut
Andrew A. Chien, University of Illinois, Urbana, Illinois
Young-il Choo, Yale University, New Haven, Connecticut
S. Duvvuru, University of Oregon, Eugene, Oregon
Wuchun Feng, University of Illinois, Urbana, Illinois
Ian Foster, Argonne National Laboratory, Argonne, Illinois
Guang R. Gao, McGill University, Canada
L. Hansen, University of Oregon, Eugene, Oregon
William Ludwell Harrison III, University of Illinois, Urbana, Illinois
Laurie Hendren, McGill University, Canada
Seema Hiranandani, Rice University, Houston, Texas
H. Honda, Waseda University, Tokyo, Japan
Elaine Kant, Schlumberger Laboratory for Computer Science, Austin, Texas
Hironori Kasahara, Waseda University, Tokyo, Japan
Ken Kennedy, Rice University, Houston, Texas
Carl Kesselman, California Institute of Technology, Pasadena, California
Charles Koelbel, Rice University, Houston, Texas
Ulrich Kremer, Rice University, Houston, Texas
Monica S. Lam, Stanford University, Stanford, California
Steve Lucco, University of California at Berkeley, Berkeley, California
Hakan Millroth, Uppsala University, Sweden
Masao Morita, Mitsubishi Research Institute, Japan
S. Narita, Waseda University, Tokyo, Japan
M. Okamoto, Waseda University, Tokyo, Japan
Dilip S. Poduval, Kent State University, Kent, Ohio
John Sargeant, University of Manchester, Manchester, U.K.
A. V. S. Sastry, University of Oregon, Eugene, Oregon
Oliver Sharp, University of California at Berkeley, Berkeley, California
R. Sundararajan, University of Oregon, Eugene, Oregon
Evan Tick, University of Oregon, Eugene, Oregon
Chau-Wen Tseng, Rice University, Houston, Texas
Kazunori Ueda, ICOT, Japan
Clifford Walinsky, Dartmouth College, Hanover, New Hampshire
Will Winsborough, Pennsylvania State University
X. Zhong, University of Oregon, Eugene, Oregon

Distribution for ANL-91/34

Internal:

J. M. Beumer (50)
F. Y. Fradin
I. Foster (30)
H. G. Kaper
G. W. Pieper
R. Stevens
D. P. Weber
C. L. Wilkinson

ANL Patent Department
ANL Contract File
TIS Files (3)

External:

DOE-OSTI, for distribution per UC-405 (58)
ANL Libraries
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
W. W. Bledsoe, The University of Texas, Austin
P. Concus, Lawrence Berkeley Laboratory
E. F. Infante, University of Minnesota
M. J. O'Donnell, University of Chicago
D. O'Leary, University of Maryland
R. O'Malley, Rensselaer Polytechnic Institute
M. H. Schultz, Yale University
K. Aida, Waseda University, Tokyo, Japan
D. Banerjee, Dartmouth College
A. Bansal, Kent State University
I. Baxter, Schlumberger Laboratory for Computer Science
J. Cavallini, Department of Energy - Energy Research
M. Chen, Yale University
A. A. Chien, University of Illinois, Urbana
Y. Choo, Yale University
S. Duvvuru, University of Oregon
W. Feng, University of Illinois, Urbana
G. R. Gao, McGill University
L. Hansen, University of Oregon
W. L. Harrison III, University of Illinois, Urbana
L. Hendren, McGill University
S. Hiranandani, Rice University
H. Honda, Waseda University, Tokyo, Japan
F. Howes, Department of Energy - Energy Research
E. Kant, Schlumberger Laboratory for Computer Science
H. Kasahara, Waseda University, Tokyo, Japan
K. Kennedy, Rice University

C. Kesselman, California Institute of Technology
T. Kitchens, Department of Energy - Energy Research
C. Koelbel, Rice University
U. Kremer, Rice University
M. S. Lam, Stanford University
S. Lucco, University of California at Berkeley
H. Millroth, Uppsala University, Sweden
M. Morita, Mitsubishi Research Institute, Japan
S. Narita, Waseda University, Tokyo, Japan
D. Nelson, Department of Energy - Energy Research
M. Okamoto, Waseda University, Tokyo, Japan
D. S. Poduval, Kent State University
J. Sargeant, University of Manchester, U.K.
A. V. S. Sastry, University of Oregon
O. Sharp, University of California at Berkeley
R. Sundararajan, University of Oregon
E. Tick, University of Oregon
C-W. Tseng, Rice University
K. Ueda, ICOT, Japan
C. Walinsky, Dartmouth College
W. Winsborough, Pennsylvania State University
X. Zhong, University of Oregon