1 of 1

CONF 9310220

LA-UR -93-3856

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36

TITLE: RUN-TIME RECOGNITION OF TASK PARALLELISM
WITHIN THE P++ PARALLEL ARRAY CLASS LIBRARY

AUTHOR(S): Rebecca Parsons and Daniel Quinlin

SUBMITTED TO: IEEE Conference on Scalable Parallel Libraries
Mississippi State University
October 6-8, 1993

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive royalty-free license to publish or reproduce the published form of this contribution or to allow others to do so, for U.S. Government purposes.

The Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy.

Los Alamos

Los Alamos National Laboratory
Los Alamos New Mexico 87545

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

for

Run-time Recognition of Task Parallelism Within the P++ Parallel Array Class Library

Rebecca Parsons

Computing and Communications Division
Los Alamos National Laboratory
Los Alamos, NM 87545

Daniel Quinlan

Computing and Communications Division
Los Alamos National Laboratory
Los Alamos, NM 87545

Abstract

This paper explores the use of a run-time system to recognize task parallelism within a C++ array class library. Run-time systems currently support data parallelism in P++, FORTRAN 90 D, and High Performance FORTRAN. But data parallelism is insufficient for many applications, including adaptive mesh refinement. Without access to both data and task parallelism such applications exhibit several orders of magnitude more message passing and poor performance [11]. In this work, a C++ array class library is used to implement deferred evaluation and run-time dependence for task parallelism recognition, to obtain task parallelism through a data flow interpretation of data parallel array statements. Performance results show that the analysis and optimizations are both efficient and practical, allowing us to consider more substantial optimizations.

1 Introduction

The use of the C++ object-oriented language permits simplified prototyping of language features. Previous work on array class libraries supported the requirements of software development for parallel numerical applications [12]. A specific target for this work has been parallel adaptive mesh refinement. Such applications use multiple grids to enhance the solution of partial differential equations.

P++ [3] is a data parallel array class library having the same interface as the commercially available serial array class library M++¹. But many applications include both data and task parallelism. Specifically, research on the parallel solution of partial differential equations using adaptive meshes and composite

Figure 1: Composite Grid using multiple grids for enhanced resolution.

grids (see Figure 1) has provided new algorithms necessitating task parallelism (the Asynchronous Fast Adaptive Composite Grid Solver AFAC and in the improved variant AFACx [18]). The requirements of a data parallel language include a run-time system, since the distribution of data is unknown to the compiler until run-time and must be allowed to change dynamically (allowing only static partitioning of predefined data can permit the exclusion of the run-time system).

The combined task and data parallelism present in AFAC, AFACx, and other algorithms are not easily exploited in the development of parallel applications. The development of parallel software is a general problem which is greatly complicated by the use of increasingly complex numerical algorithms for large scale numerical computation. An additional complexity is the often redundant implementation of numerical codes required onto different computer architectures coupled with the relatively short useful lifetimes of such parallel machines. The use of proposed data parallel languages such as HPF only partly address such require-

¹M++ is a product of Dyad Software.

ments, since they are restricted to providing simplified access to only data parallelism, and often, both data and task parallelism must be exploited.

The A++ serial array class library, written in C++, defines the serial array interface which the P++ uses and is portable. P++ is a parallel array class library which restricts itself to a data parallel programming environment. But P++ as a C++ class library is simplified in its construction and allows significant modifications to be implemented experimentally. We report on the modification of the A++ serial array class library to include the recognition of task parallelism, preliminary to its incorporation into P++. This paper contains a description of the problems associated with task parallelism recognition for adaptive mesh refinement applications and describes the performance of recognizing task parallelism in a data parallel array class library.

The serial array class library acts as the interface to an architecture independent environment for parallel applications that can exploit both data and task parallelism at run-time. The use of the C++ class library P++ for run-time support of data parallelism was previously detailed in [11]. By itself, data parallelism is insufficient for applications such as adaptive mesh refinement. To support an architecture independent environment and because task parallelism can not readily be recognized at compile time, task parallelism is supported at run-time. Specifically, execution of the array statements is deferred, for later evaluation. The user specifies a block of array code for which a list of array expressions is formed. This list is analyzed at run-time using dependence analysis, and the resulting information is used to partition the execution list into isolated tasks. The deferred evaluation, dependence analysis, and task separation are shown in Section 5 to be efficient.

Section 2 introduces both the A++ serial array class library and the P++ parallel array class library, and defines the important restrictions of the P++/A++ array interface that permit the efficient recognition of task parallelism. Section 3 motivates our requirement for task parallelism using the adaptive mesh refinement example. Section 4 details the methods used to recognize parallel tasks within serial code. Section 5 presents performance results. Section 6 describes some of the future directions of this work. Finally in Section 7 we present some conclusions from this work.

2 C++ Array Class Libraries

The ability to build C++ class libraries provides simplified access to the development of alternative and specialized languages. Our class library development has been motivated by our research on parallel numerical methods for adaptive mesh refinement. The complexities of software development for such applications have been greatly simplified by the use of the object-oriented C++ language (see [11],[12],[3]).

In [11], C++ was used to develop two class libraries to separate the issues of parallel adaptive mesh refinement into those of parallelism and adaptive mesh refinement. The P++ parallel array class library simplifies the development of codes for parallel execution of numerical applications that were developed for the serial environment using the M++ array class library. The AMR++ adaptive mesh refinement class library abstracted the requirements of general adaptive mesh refinement from the development of adaptive mesh refinement applications. The AMR++ class library is implemented using M++. To obtain the parallel adaptive mesh refinement application, the serial adaptive mesh refinement class library is recompiled with the parallel array class library. This software development strategy has permitted us to develop parallel adaptive mesh refinement software with much greater complexity than conventionally developed parallel adaptive mesh refinement applications ([13],[13]) with explicit data partitioning and an explicit message passing model.

More recent work replaces the use of M++ with a higher performance serial array class library A++. A++ was designed for portability, greater efficiency, and combined use with Fortran.

2.1 A++ Serial Array Class

A++ is a serial array class library intended for use on either serial machines or single processors of parallel distributed memory architectures. It is implemented in two layers, a machine independent layer in C++ and a machine dependent layer in any language suitable for the target machine². A++ is designed for both high efficiency and portability. Machine dependent layers have been implemented on the Cray, CM-5, and SUN Sparc permitting A++'s use on these architectures.

A++ provides arrays of from one to four dimensions. Array operators and functions operate on the

²The target machine must have a C++ compiler and the machine dependent interface must be linkable to C code.

arrays independently of the array dimension. The syntax associated with the manipulation of the arrays is very similar to FORTRAN 90 array extensions. A++ represents arrays as objects, specifically arrays of double, float, and integer. Each type of array is defined as a class; each class overloads the standard arithmetic and relational operators. Additional functions overload the standard math library functions, such as `sqrt()` and `cos()`, and so provide equivalent operations for array objects.

A++ additionally provides indexing objects that can be used with overloaded parentheses operators. The indexing objects have their own overloaded operators, as in the index manipulation required in finite difference stencils. Index objects represent the position, count, and stride associated with a view of an array object, similar to FORTRAN 90 triplets. A++ currently implements only constant stride index objects for use with structured grids. Figure 2.2 shows a simple example of the use of the A++ array class.

To handle conditional statements A++ provides a `where` construct (including `elsewhere` support). The `where` support provides a suitable interface for serial or parallel support, though `where` support has not been added to P++ yet.

2.2 P++ Parallel Array Class Library

P++ is a parallel array class library that uses A++ internally for all array manipulations. It implements the identical array interface, but permits array objects to be distributed across arbitrary collections of processors; array operations then introduce message passing to properly execute each distributed array statement. Control over partitioning is handled by an optimization manager class which is common to both the A++ and P++ class libraries. Default partitioning is to distribute the last dimension of each multidimensional array object over all the available processors. Use of a default partitioning provides a simplified strategy to obtain efficient performance from the user's recompiled application using P++ in place of A++.

P++ provides an SPMD data parallel execution of the user's serial array application. The user's application code consisting of array operations (or non-distributed scalar operations executed on each processor) is executed correctly regardless of the distribution of the data. Optimization of the distribution of the array objects is one method by which the user can optimize parallel performance. P++ also manages all message passing required for the correct execution of the overloaded array operation. Message passing is

interpreted at run-time since only at run-time is the data distribution in a dynamic application known.

The interpretation of the message passing required for correct execution of the array statements is done in one of two ways depending on the degree of alignment of the array object operands for each array operation. Details of the run-time message passing models are contained in [11]. In each statement and for every partitioning of the operand's data, the message passing introduced by P++ is optimal. Deferred evaluation accumulates array operations over large sections of user code. It is expected to provide the mechanism to perform optimal message scheduling over small collections of array statements. The current interpretation looks at each array statement individually and optimizes the message passing for that statement.

Example 2.2 shows the use of P++ with the default partitioning. The code is compiled using A++ for the serial environment and with P++ for the distributed memory environment.

2.3 Restrictions on A++/P++ interface

The A++/P++ interface is similar to the FORTRAN 90 array interface, except that indexing triplets are simplified to index objects. Fundamental to the array interface is that the right-hand side operations are evaluated before any assignment to the left-hand side. This restriction is key to the parallelization of the A++/P++ array statements since the user has provided the parallelism for P++ to exploit. Since the exploitation of the data parallelism is both architecture dependent and complicated, P++ addresses these particular points, providing a simple development environment that is architecture independent.

The greatest efficiency in A++/P++ comes from the manipulation of the arrays using the array operations. The use of scalar indexing is not expected to be very efficient. However, using C++ compilers with more aggressive optimizations permits the equivalent performance of explicit indexing in FORTRAN and C using conventional array and loop construction. P++ at present does not support a message passing interpretation of scalar indexing.

3 Motivation for Task Parallelism

We separate parallel software support into two broad categories, data parallelism (e.g., HPF, FORTRAN 90 D, and current P++ work) and task parallelism (e.g., CC++ [5]). Among numerical solution methods for partial differential equations, applications

```

#include "header.h"
#ifdef PPP
#define doubleArray double_VSG_Array
#define Index VSG_Index
#endif

void Hyper (Index I, double T_Step, doubleArray & F,
            doubleArray & U_NEW, doubleArray & U_OLD)
{ // array expression:
  F      = (U_OLD * U_OLD) / 2;
  // scalar expression:
  U_NEW(0) = U_OLD(0) - T_Step * (F(1) - F(0));
  // indexed array expression:
  U_NEW(I) = U_OLD(I) - T_Step * (F(I+1) - F(I));
  // array expression:
  F      = (U_NEW * U_NEW) / 2;
  // indexed array expression:
  U_NEW(I) = 0.5 * (U_OLD(I) + U_NEW(I)) -
            0.5 * T_Step * (F(I) - F(I-1));
}

void main()
{ int N;
  double T_Step;
  scanf (&N, &T_Step);
  doubleArray U_OLD (N,N), U_NEW (N,N), F(N,N);
  // Setup data
  ...
  int Interior_Start_Position = 1;
  int Interior_Count          = N-2;
  int Interior_Stride         = 1;

  Index Interior (Interior_Start_Position,
                  Interior_Count, Interior_Stride);

  Hyper (Interior, T_Step, F, U_NEW, U_OLD);
}

```

Figure 2: C++/A++/P++ example code: MacCormack (Hyperbolic) Scheme.

using a single grid (i.e., a single set of arrays to represent a grid on which finite difference equations are solved) benefit most from data parallel language implementations.

Adaptive mesh refinement combines the use of multiple grids (e.g., a global grid and many local refinement grids, see Figure 1). Research on parallel numerical algorithms for such adaptive meshes [13] has developed methods that permit the asynchronous processing of the local refinement grid in the adaptive meshes (composite grids). Although each grid may be solved using data parallel operations, the ordering of the data parallel processing of each grid using a data parallel language remains sequential. Using only data parallelism, the performance of this approach is especially degraded when, because of the numerical algorithms, each grid is distributed over a restricted set of processors as in Figure 3. In this specific example,

Figure 3: Example of Composite grid partitioning for AFAC/AFACx algorithm.

each of the grids that form the composite grid could be solved in parallel; the data parallel processing of the grids can be performed in parallel as well.

Figure 3 shows the partitioning of 2 grids across 2 processors (A and B). This partitioning, and the analogous extension to more grids and more processors, is the most efficient one for the solution of an elliptic application on the related composite grid problem using the AFAC or AFACx algorithm [10]. The motivation for task parallelism in our current approach to adaptive mesh refinement stems from the inefficiency associated with data parallel execution of this example. Since the data parallel execution of solvers on each grid would require a sequential processing of each grid, data parallelism on each grid would be advantageous only if each grid were partitioned uniformly onto the multiprocessor system. But such a partitioning is inefficient [10] since the amount of work on each grid can not offset the cost of the parallel overhead of the message passing³. Knowledge of the task parallelism in the solution process allows any partitioning of the collection of grids, in particular the distribution in Figure 3 shown to be most efficient (see [10]).

The specification of explicit task parallelism by the user interferes with the goal of architecture independence in the development of the application code. We show in this paper that the use of an array syntax and a run-time system provides the basis for the recognition of task parallelism in the user's application code developed for the serial environment.

3.1 Why a Run-Time System

For performance reasons, run-time systems are avoided whenever possible. However, there continue to be significant analyses that one would like a compiler to do that either can not be done to an acceptable degree of precision or can not be done efficiently.

³The number of messages required for solution in such a partitioning is several orders of magnitude higher using a uniform partitioning of each grid.

Many of the problems that one would like a compiler to solve are undecidable. For others, decisions made by the program at run-time determine the solution to the problem, making it impossible for a compiler to provide a precise solution. Compilers must err on the side of conservatism to ensure the correctness of the compiled code.

Another important issue about compiler capabilities is that a compiler must account for the semantics of the entire language. In languages such as C and C++ which include pointers and objects, the level of analysis possible is quite low, since pointers allow very general reference patterns and objects generally prevent the compiler from obtaining the level of information necessary to perform even basic optimizations. The array class library, however, imposes restrictions on how array objects can be accessed and manipulated. Thus, the analysis that we perform is much simpler than that which the compiler must do. In addition, we have opportunities for optimizations that are not available to the compiler.

4 Recognition of Task Parallelism

Deferred evaluation provides the mechanism for the recognition of task parallelism. The user specifies the region of deferred evaluations, which bounds the array statements that will be eligible for parallel execution. Under deferred evaluation, a list of the array statements to be executed is formed. We perform run-time dependence analysis, in an extremely efficient manner, to determine which statements can be executed in parallel. Based on these dependences, we combine statements that depend on each other, forming regions consisting of interdependent statements, yielding regions that can be executed in parallel. Each step of the method is outlined below; see Parsons and Quinlan [16] for the details of the algorithms.

Figure 4.3 shows a sample code fragment that implements deferred evaluation for an array of grids, each containing solvers. In this case each solver is independent, and the resulting independent tasks are recognized and executed for each time step. However the construction of the execution object list and its analysis is only done once. The present user interface is sufficient for our use with the AMR++ class library.

4.1 Deferred Evaluation

The first step in the recognition of task parallelism is the accumulation of the statements that will be executed. This step is complicated by the inaccessibility

of the program source. It is simplified by the complete knowledge of control flow which is available at run-time. Within an array class, the array object's member functions are defined so that if deferred evaluation is turned on by the user, the member functions skip the processing of the array data and record the invocation in an *execution object*.

The purpose of the execution object is to record the array operation invocation. Since many occur and the order is important, they are accumulated in an ordered list. Because of the different number and types of operands that are required for the execution of the functions in the machine dependent interface, there are many different types of execution objects. Each execution object is capable of calling the correct machine dependent functions using the input parameters saved during the construction of that object. Minimal memory is required for the linked list of execution objects; tests of deferred evaluation have easily recorded 50,000+ array operations.

The overhead of deferred evaluation includes more than just the cost of building the linked list of execution objects. This subtle point originally made it unclear whether or not deferred evaluation could be implemented efficiently. However, the results in Section 5 demonstrate that the cost of deferred evaluation represents only a few percent overhead to the normal evaluation of the array objects.

The problem arises when a deferred operation had, as one of its operands, a local variable of a scope different than the scope of the deferred evaluation. The local variable would be constructed during the deferred evaluation and then be destroyed before the execution object executed. Clearly, if the execution object was executed and the operand did not exist, the execution would cause an error. The solution is to use the pointer indirection associated with the array object to extend its lifetime. The signal to extend the lifetime of a local variable is when its destructor is called and deferred evaluation is still turned on.

There are some restrictions on the use of deferred evaluation. The assumption of deferred evaluation is that the control flow is fixed (most conditional operations in the array syntax use the A++ **where** function). Reduction operators (e.g., min and max operations) return scalar values dependent upon the data, disallowing deferred evaluation.

4.2 Dependence Analysis

To determine which statements can be executed in parallel, we examine the dependences among the statements. Since we are interested in statements only

Figure 4: Data Dependences

within the range of deferred evaluation, the control flow is fixed. In this setting, only data dependences exist. In general, a dependence between two statements specifies the relative order in which those statements must be executed for the program to give the right answer. In previous work both by Horwitz *et al.* [8] and Parsons [14], two different approaches established that any execution order respecting the dependences of a program results in the same outcome for the program.

There are three types of data dependences, shown in Figure 4: flow dependences, output dependences and anti-dependences [9]. Informally, these dependences are defined as follows:

Flow There is a flow dependence from statement A to statement B if A assigns a value to some identifier x , B uses x , and there is some path from A to B in the execution of the program that contains no other assignment to x . Thus, statement A computes a value for x that statement B uses.

Output There is an output dependence from statement A to statement B if both statements assign a value to some identifier x and there is some path from A to B in the execution of the program that contains no other assignment to x .

Anti- There is an anti-dependence from statement A to statement B if A uses some identifier x , B assigns a value to x , and there is some path in the execution of the program from A to B that includes no other assignments to x . Thus, statement A uses a value of x that must be fetched before statement B over-writes the value.

The works cited above by Horwitz and Parsons both assume an implicit renaming of identifiers that renders anti-dependences unnecessary. Renaming of scalar identifiers is quite common as an optimization since it reduces the number of dependences. However, when dealing with arrays, renaming requires too much space. Therefore, renaming is not performed and anti-dependences must be considered. There is, however, a relationship between these three types of data dependences. In the absence of control dependences, there is an anti-dependence from statement A to statement B

if there is some other statement C such that there is an output dependence from C to B and a flow dependence from C to A. For the formal statement of this relationship and its proof, see Parsons and Quinlan [16].

While this relationship does not imply that we can ignore anti-dependences when determining the sequencing of statements A, B, and C, it does imply that we can ignore anti-dependences when determining which statements are independent of one another. It is this property we exploit in the recognition of task parallelism.

4.2.1 Recognition of Dependences

Since the deferred evaluation resolves the control flow of the program, dependence analysis is quite simple. The dependence recognition algorithm uses a symbol table to record which statement holds the current assignment for an array object. For each operand reference and assignment, the array object is looked up in this symbol table, yielding the statement number of the output or flow dependence predecessor. Since hash table look-ups are, on average and for a suitably sized hash table, a constant time operation, the dependence recognition pass is linear in the number of array operations deferred.

It is important to recognize the simplifications in dependence analysis that are possible in this special context. Since the statements being considered have no control dependences, each statement can only have one output predecessor and one flow predecessor for each array used to compute the expression. In addition, dependences can only flow forward in the execution object list. As a result, there is no need to consider loop-dependent dependences [15], and we could compute dependences for objects as they are placed in the execution object list. Next, for the initial implementation, we avoid subscript analysis. Therefore, any access to an array is assumed to be dependent on the prior access to that array. Finally, the array class library semantics follows the Fortran-90 semantics, meaning that all operands of a statement are fetched, and then the operations are performed on these fetched objects. Thus, a statement can not be dependent upon itself.

4.3 Task Recognition Algorithm

Once the dependences have been recognized, we partition the statements into completely independent task regions. As we are only looking for completely independent regions, the relationship among anti-dependences with output and flow dependences allows

```

Set_Of_Tasks Task_Set;
for (int T_Step=0; T_Step < Max_T_Steps; T_Step++)
{
    // Only turn on deferred evaluation once

    if ( Optimization_Mgr::Def_Eval_ON ( Task_Set ) )
    {
        for (int i=0; i < NUMBER_OF_SOLVERS; i++)
            List_Of_Solvers [i].Solve();
    }
    Task_Set = Optimization_Mgr::Def_Eval_OFF ();
    Task_Set.Execute();
}

```

Figure 5: Example of deferred evaluation on collection of grid solvers.

us to ignore anti-dependences. We use the union-find algorithm presented in Aho *et al.* [1] to recognize the independent regions. Initially, each object in the execution object list is in the same region as all other objects relating to that statement. We combine regions containing the dependence predecessors of the object being analyzed with the region for that object, examining each object in the execution object list only once. After this pass of the execution object list finishes, the regions represent sets of objects which are interdependent. These regions are the weakly connected components of the dependence relation for the objects. Since objects in different regions are completely independent, these regions can be executed in parallel. To form the task lists, the execution object list is traversed in order once more, with task lists formed based on the region id, maintaining the relative ordering from the execution object list within the regions.

Aho *et al.* [1] present an amortized analysis of this algorithm showing that for a linear number of union and find operations, this algorithm is nearly linear. Thus, the process of finding the independent tasks from the execution object list is quite efficient, and practical even for small arrays.

5 Results

The principle results we report are for the recognition of task parallelism. To provide perspective on the results, we introduce preliminary performance results for A++ and for deferred evaluation. The A++ code is evaluated on a simple set of array statements to show that the overhead is small compared to the optimal performance potential. The results compare

A++ to the equivalent C code. The task parallelism performance results are taken from an example code which simulates the solution of many grids as part of an adaptive mesh refinement application.

5.1 A++

The performance of A++ is characterized in Figure 5.1 using two example array statements on different sized arrays and comparing the execution times to both the equivalent C code and the C code processing one operation at a time. The importance of the second *simulation* of the array class is to provide a lower bound on the timings for the array class, since the array class is forced by definition to process the array operations one at a time. The advantage for the C code is that hardware registers can be used to accumulate results internal to the loop processing. The array class is at present limited to processing whole vectors for each binary operation one at a time.⁴

5.2 Adaptive Mesh Refinement Example

In this example code we solve a shock tube problem using a second order Godunov scheme. The solver is called for each grid which is represented by a collection of arrays. Twenty grids are allocated and the solvers are called for each grid. Since the solver on each grid has its own data, the execution of each solver is independent. The purpose of this test is to obtain performance measures of the recognition of this task parallelism. Each solver contains about 160 array statements with approximately 500 array operations. Figure 4.3 shows the example code which solves the separate grids using the present A++ interface for deferred evaluation. Figure 5.4 shows the timings for the code fragment in Figure 4.3.

5.3 Deferred Evaluation

As a platform for optimization, deferred evaluation provides the required control flow and list of operations. Optimizations are then applied to the list of operations. The principle result from this work is that deferred evaluation represents a sufficiently small overhead that it can be practical to use. The performance of deferred analysis is in general independent of the array size; however, within the test code each solver

⁴Additional simple analysis can be done on the execution objects to combine binary operators into larger aggregate operators and provide improved register utilization for the array class.

Figure 6: A++ performance timings.

allocates many arrays dynamically and this has a small cost dependent on the array size.

Figure 5.4 shows that the construction of the execution object list is about 33% for length 100 arrays, 7% for length 1000 arrays, and 1% for length 10,000 arrays.

5.4 Recognition of Parallel Tasks

The recognition of task parallelism represents the fastest operation once the list of deferred operations has been built. After the operation list construction the recognition of task parallelism consists of a fixed cost independent of the array size. First, dependence analysis is done on the list of operations. Then the list is separated into independent sets. Finally, the independent sets are executed. Figure 5.4 shows the time required to build the execution object list, perform the dependence analysis, separate the list into inde-

Figure 7: Timings for deferred evaluation and recognition of task parallelism for previous code fragment.

pendent tasks, and execute the independent tasks.

6 Future Work

The current work is preliminary in nature since deferred evaluation makes many other optimizations possible. Future work will add the deferred evaluation to P++ by making it accessible through A++.

6.1 Semantic Completeness of Dependence Information

Once the dependence information for the execution object list is available, many other optimizations become possible since the objects and the dependences among the objects completely characterize the meaning of the program. Taken together, the different dependences characterize the flow of data, the utilization of memory resources, the patterns of memory utilization, the sequencing constraints, and many other facets of the performance of a program. The information requirements of many optimizations can easily be stated in terms of dependences. The specific program text is not needed to perform optimizations, nor is it necessary to perform optimizations on program text. Instead, these optimizations can alter the execution object list and the dependences between these objects. This dynamic use of dependence information provides an opportunity to exploit information only

available at run-time to improve the performance of the program.

6.2 Future Optimizations

The purpose of this work has been to show that deferred evaluation is not an expensive sort of operation in an array class library. The advantages are in the sophisticated optimizations that are made possible once deferred evaluation is used. Specifically the optimizations come in three areas: optimization over large sets of array statements (100 - 1,000,000+) for recognition of independent parallel tasks, thus exploiting task parallelism in code developed for the serial environment; optimization over medium sized sets of array statements (approx. 10-100) for optimized message scheduling in the distributed memory environment; and optimization over small sets of array statements (1-10) for optimized temporary handling across statements, for chaining of operations on vector architectures, and for improved register utilization within low-level array operations.

6.3 Compiler and Language Implications

While the system in its current form is quite useful in applications such as adaptive mesh refinement, additional work on the techniques will make it more widely applicable. There are several questions that should be considered in this regard: What information could the run-time analysis exploit that could be provided by a compiler? What features of this technique could be incorporated into a sub-language? Could this technique be improved by using a compiler and/or pre-processor specifically designed for this sub-language?

The last two questions provide two avenues that we plan to explore immediately. There are restrictions placed on the range of deferred evaluation that result directly from the implementation of these features within a C++ array class library. Deferred evaluation is currently not possible for scalar statements since the class library is not invoked in this context. Thus, there are many instances where deferred evaluation could be useful but it can not be exploited. If instead, we had some language support for these mechanisms, restrictions like these could be lifted or at least lessened. This area is one we plan to address with colleagues working in the area of building C++ compilation systems targeted towards scientific computation.

7 Conclusions

The purpose of this work has been to show that deferred evaluation, run-time dependence analysis and task recognition are not expensive operations in an efficient array class library. The work has been done in A++ as a proof of the concept, and to demonstrate what will be added to P++, permitting P++ to exploit both data and task parallelism and allowing the run-time system to search for the task parallelism in the user's serial application code.

The deferred evaluation pays for itself even in a serial environment, although the difference is small. Deferred evaluation even on large arrays costs very little (approximately 2-5% of the cost of a single execution of the equivalent code). The execution of the deferred evaluation list of operations is about 7% less expensive than the equivalent array class operations and recoups most of the approximately 10% overhead of the array operations.

The cost of the dependence analysis and the separation of the deferred evaluation list of operations into independent tasks is sufficiently small as to be less than 1% of the cost of the single time step for array length 1000. Given the fixed cost of such analysis, which is also independent of the number of time steps, additional analysis could be easily done without significant cost.

Having shown that deferred evaluation can be done with only a few percent overhead and that the overhead can be quickly recouped in the subsequent repeated execution of the execution object list, we expect significant opportunities for much more complex optimizations such as memory hierarchy management for RISC microprocessors increasingly found in current lines of parallel computers.

8 Acknowledgements

We wish to thank Jeff Saltzman and Joseph H. Fasel for their support and advice throughout this work. This work was performed under the auspices of the U.S. Department of Energy under contract #W-7405-ENG-36.

References

- [1] Aho A. V. and Hopcroft, J.E. and Ullman, J.D.: *The Design and Analysis of Computer Algorithms*; Addison Wesley, 1986.

- [2] Angus, I. G. and Thompkins, W. T.: *Data Storage, Concurrency, and Portability: An Object Oriented Approach to Fluid Dynamics*; Fourth Conference on Hypercubes, Concurrent Computers, and Applications, 1989.
- [3] Balsara, D., Lemke, M., Quinlan, D.: AMR++, a C++ Object Oriented Class Library for Parallel Adaptive Mesh Refinement Fluid Dynamics Applications. Proceeding of the American Society of Mechanical Engineers, Winter Annual Meeting, Anaheim, CA, Symposium on Adaptive, Multi-level and Hierarchical Computational Strategies, November 8-13, 1992.
- [4] Berryman, H.; Saltz, J.; Scroggs, J.: *Execution time support for adaptive scientific algorithms on distributed memory machines*; Concurrency: Practice and Experience, Vol. 3(3), pp 159-178, June 1991.
- [5] Chandy, K.M.; Kesselman, C.: CC++: A Declarative Concurrent Object-Oriented Programming Notation; California Institute of Technology, Report, Pasadena, 1992.
- [6] Forslund, D.; Wingate, C.; Ford, P.; Jenkins, S.; Jackson, J.; Pope, S.: *Experiences in writing a distributed Particle Simulation code in C++*; USENIX C++ Conference Proceedings, San Francisco, CA, 1990.
- [7] Fox, G.; Hiranandani, S.; Kennedy, K.; Koelbel, C.; Kremer, U.; Tseng, C.-W.; Wu, M.-Y.: *FORTRAN D Language Specification*; NPAC Technical Report, Syracuse University, also: Technical Report, Rice University, 1991.
- [8] Horwitz, S.; Prins, J.; and Reps, T.: *On the Adequacy of Program Dependence Graphs*; Proceedings of the Fifteenth ACM Symposium on the Principles of Programming Languages, 1988.
- [9] Kuck D. J.: *The Structure of Computers and Computation*; John Wiley and Sons, 1978.
- [10] Lemke, M.; Quinlan, D.: *Fast adaptive composite grid methods on distributed parallel architectures*; Proceedings of the Fifth Copper Mountain Conference on Multigrid Methods, Copper Mountain, USA-CO, April 1991. Also in Communications in Applied Numerical Methods, Wiley, 1992.
- [11] Lemke, M.; Quinlan, D.: *P++, a C++ Virtual Shared Grids Based Programming Environment for Architecture-Independent Development of Structured Grid Applications*; accepted for CONPAR/VAPP V, September 1992, Lyon, France; to be published in Lecture Notes in Computer Science, Springer Verlag, September 1992.
- [12] Lemke, M.; Quinlan, D.; Witsch, K.: *An Object Oriented Approach for Parallel Self Adaptive Mesh Refinement on Block Structured Grids*. Proceedings of the 9th GAMM-Seminar Kiel, Notes on Numerical Fluid Mechanics, Vieweg, Germany, 1993.
- [13] McCormick, S.; Quinlan, D.: *Asynchronous multilevel adaptive methods for solving partial differential equations on multiprocessors: Performance results*; Parallel Computing 12, 1989, pp 145-156.
- [14] Parsons Selke R.: *A Rewriting Semantics for Program Dependence Graphs*; Proceedings of the Sixteenth ACM Symposium on the Principles of Programming Languages, 1989.
- [15] Parsons, R.: *Semantic Program Dependence Graphs*; Ph.D. Thesis, Rice University, 1992.
- [16] Parsons R. and Quinlan D.: *Run-Time Recognition of Task Parallelism Using Deferred Evaluation*; Los Alamos National Laboratory Technical Report, in preparation.
- [17] Peery, J.; Budge, K.; Robinson, A.; Whitney, D.: *Using C++ as a scientific programming language*; Report, Sandia National Laboratories, Albuquerque, NM, 1991.
- [18] Quinlan, D.: *Parallel Adaptive Mesh Refinement*, Ph.D. Thesis, University of Colorado at Denver, 1993.

**DATE
FILMED**

1 / 6 / 94

END