AIIM

Association for Information and Image Management

1100 Wayne Avenue, Suite 1100
Silver Spring, Maryland 20910
301/587-8202

Centimeter

Inches

MANUFACTURED TO AIIM STANDARDS
BY APPLIED IMAGE, INC.

1 of 1

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.

An Informal Introduction to Program Transformation and Parallel Processors*

by

Kevin W. Hopkins
Southwest Baptist University
Bolivar, MO 65613

***Work performed at Argonne National Laboratory, a contract Laboratory of the United States Department of Energy, while a Faculty Research Participant.**
Program administered by the Argonne Division of Educational Programs with funding provided by the U. S. Department of Energy.

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

The submitted manuscript has been authored by a contractor of the U. S. Government under contract No. W-31-109-ENG-38. Accordingly, the U. S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U. S. Government purposes.

1. Introduction. In the summer of 1992, I had the opportunity to participate in a Faculty Research Program at Argonne National Laboratory. I worked under Dr. Jim Boyle on a project transforming code written in pure functional Lisp to Fortran code to run on distributed-memory parallel processors. To perform this project, I had to learn three things: the transformation system, the basics of distributed-memory parallel machines, and the Lisp programming language. Each of these topics in computer science was unfamiliar to me as a mathematician, but I found that they (especially parallel processing) are greatly impacting many fields of mathematics and science. Since most mathematicians have some exposure to computers, but certainly are not computer scientists, I felt it was appropriate to write a paper summarizing my introduction to these areas and how they can fit together. This paper is not meant to be a full explanation of the topics, but an informal introduction for the "mathematical layman." I place myself in that category as well as my previous use of computers was as a classroom demonstration tool.

2. The Transformation System. The idea of program transformation has existed for at least twenty years, and many different transformation systems have been developed [3, 4, 5, 7, 10, 11, 14, 25]. At Argonne National Laboratory, Dr. Jim Boyle has developed the TAMPR (Transformation-Assisted Multiple Program Realization) system and used it on a number of projects. He has also supervised a number of visitors, ranging from undergraduate researchers [17] to faculty researchers.

Program transformation is a process by which a computer takes an input program and, by applying a series of correctness-preserving transformations, produces a program which does the same thing as the original program, but has

been optimized. These transformations are performed in a number of small steps. Thus it is easier to prove each step is correct and to guide the process to the desired conclusion, a correct program. Proving the correctness of the result is associated with denotational semantics [23] and is a topic that is being studied by Victor Winter, a guest of Dr. Boyle for a week this summer. How many of us ever consider proving that a program we have written is correct if it seems to be doing what we want it to? This is in sharp contrast to our desire to prove mathematical results, and is an indication that the field of computer science is addressing issues that mathematicians have felt necessary for years.

There are a number of possible definitions of the word optimized as used in the preceding paragraph. It could be that certain information was known about the data coming to the program and the program could be made more efficient (in time, storage, or both; for example in a matrix computation if you knew the matrix was upper triangular or tri-diagonal) if this information was incorporated into the program. Optimization could mean the program is now set up to implement a new data type such as double-precision complex or quaternion [17] (the quaternion data type was needed to solve an eigensystem problem that originated from the study of the structure of a crystal) that is nonexistent on most machine architectures. Optimization could mean the program is now set up to run on a different architecture from the original program. It could also mean the program is now in a different language from the original program. Dr. Boyle (and various visitors) has examined all of these types of optimization over the years. Often it is found that one optimization builds directly on the results of a previous optimization by reusing or recycling previously written transformations.

The main benefit of transformation is a reduction of programming work and

an increase in correctness. Let us assume a programmer has written a complicated program or entire programming package and tested it to the point that he is confident of its correctness. Now he receives access to a much faster machine or has someone interested in using his package on a different machine architecture. Changing the program by hand to work on the new environment (I found it amazing the number of little subtleties that needed to be considered for each environment), and then debugging, may require almost the same amount of effort as the original writing.

Transformations provide an alternative. A set of correctness-preserving transformations could be written to perform the conversion automatically. Depending on the nature of the package, the work involved in producing the correctness-preserving transformations may be less than the work of changing the package by hand. In any case, the transformations will be reusable in any similar conversions in the future (and this was the goal of my work). This is in contrast to the approach of repeating all the work by hand every time a program is to be altered. A case where transformations have been used in this manner is the LINPACK (a package of linear algebra routines) software package, where fully three-quarters of the programs were computed from others by transformations [9].

Another advantage of transformations is that they allow for the possibility to experiment with many different forms of the same program to see which is optimal [6]. Such experimentation is practically impossible when all the different versions must be produced by hand.

TAMPR (see [8, 17] for a more extensive description of the TAMPR system) mainly involves a pattern replacement process. The writer of the transformation specifies which pattern needs to be matched and what will replace that matched

pattern. One transformation might search for multiple occurrences of $a+b$ in the code (with neither a nor b changed in between) and on finding such occurrences assign an intermediate variable the value $a+b$ with that variable then replacing occurrences of $a+b$, thus reducing the number of addition operations and the number of storage retrievals. However, all replacements must be grammatically equivalent to the original pattern (thus a knowledge of computer language grammars is essential for writing these transformations) which usually leaves the transformation more complex than might first be anticipated.

3. Parallel Processors. Parallel processing is a rather new and promising field of research in computer science. (As an indication of how current the field of parallel processing is, I note a search of the Argonne Library's database revealed 488 titles matching the keyword "parallel"; most of those were produced since 1985.) Rick Stevens, director of Argonne's High Performance Computing and Communications Program, stated the following about parallel processing:

"The past decade has seen the emergence of a new way of doing science and engineering. The new mode of 'computational science' is poised to join theory and experiment as a third approach to solving scientific and engineering problems. Its widespread use will fundamentally alter the way modern researchers attack problems." [20]

I learned about parallel processors through a series of discussions with Dr. Boyle and attendance at a few seminars. A parallel processor is a set of independent processors interconnected in such a way that communication can take place between the processors as each performs its separate tasks. A basic parallel processor that most are familiar with is a network of workstations or personal computers, with each processor doing the task assigned by its own user. More

typical of a parallel machine is one that allows for breaking up one task and having each processor performing a portion of that task. Thus the job is finished more quickly.

There are many reasons an increase in speed may be important or even necessary to attempt a problem. Many numerical techniques trade off accuracy or significantly limit the model to achieve reasonable execution time (recall many of the approximation techniques in Calculus or Differential Equations: such as Simpson's Rule, Bisection Method, or Euler's Method; where the smaller the step, the better the result, but the longer it will take). For many simulations, a "real-time" speed is required (see [22, p. 51] for a discussion of a real time simulation). These applications have the need for faster execution than "ordinary" computers provide. A whole series of Grand Challenge problems have been proposed that demand the use of parallel computers.

"The national mandate to resolve fundamental problems in science and engineering requires the assimilation of computational science and high-performance computer technology ... Improved massively parallel systems must perform faster and more accurate arithmetic calculations." [1, p. 1]

The cost for increasing the speed of one machine is currently more than the cost of using several "low-cost" processors in parallel to achieve the same or better increase in speed. Some examples of the uses of parallel processing at Argonne are climate modeling (the goal is to produce in ten years climate models that are 10,000 times or more faster than the current model) and modeling type-II superconductors (for more examples of the use of parallel processing, see [12, 19, 21]).

Ideally, if one doubles the number of processors one halves the time of the task. In practice, such speedup is not achieved, especially as the number of

processors increases [15]. There are two main reasons for this. One reason is that, in any task, there are subtasks that need to be performed sequentially (first one and then the second) rather than in parallel (both at the same time). As an example of this consider a man painting his house which normally takes him four days. If he has a helper, we would naturally assume it would take them two days working together. But if they only have one brush (thus allowing only one of them to work at a time) the job will still take four days (assuming rest time is negligible). The above example may have sounded like a trick question, but it is exactly those types of issues that must be considered.

The second reason that the theoretical speedup may not be achieved is that for parallel programs there is extra program overhead that must be included to guarantee that two processors do not attempt to do the same work or alter the same piece of data at the same time. If we return to the painting example this is analogous to having someone to allocate the tasks in such a manner that there is as little contention for shared resources (let's say, the brush, roller, and paint) as possible and to prevent both painters from trying to paint the same wall at the same time. In parallel processing this control agent is called a lock (see [18] for a discussion of locking).

In the first case, the lack of parallel work means that some -- and, as the number of processors increases, most -- of the processors must sit idle until there is work to do. The loss of speedup is usually a consequence of the algorithm being programmed and can be lessened only by changing the algorithm. In the second case, the contention to get tasks and data grows as the number of processors grows. This situation can be improved by certain programming choices and hardware considerations, but still imposes an upper bound on the amount of speedup that is

possible.

It is the programming choices of which algorithm to use and of how one divides the job that make programming in parallel far more challenging than programming a sequential machine. Dividing the job is called problem decomposition. Two methods are generally applied: domain decomposition, where the data is divided among the processors, and control decomposition, where the tasks are divided among the processors. In the Lisp-to-Fortran conversion all the parallelism is in control decomposition (for more information on parallel programming, see [24]).

I gained familiarity with two main types of parallel machines: global memory and distributed memory. The reason I needed to be familiar with both types of machines is that the distributed-memory version of the transformations was developed by starting with the global-memory version (see [7] for a discussion of the results of a program that was transformed to a global-memory version). Global-memory machines have one memory that any of the processors can access at any time, within the constraints of locking as mentioned in the preceding paragraph. A diagram of this setup is in Figure 1. The distributed-memory machines are such that each machine has its own local memory. This cuts down on much of the data locking overhead, but increases message-passing overhead, as now one processor may need the data that resides on another processor and so must ask for it by sending a message to that processor and waiting for a response. A diagram of this setup is in Figure 2.

4. Lisp. *The Little Lisper* [16] proved to be an excellent reference for me as I learned Lisp. (When Lisp is used in the paper, pure applicative Lisp is meant.

Other versions of Lisp, such as Common Lisp, have additional properties not used in pure applicative Lisp). Lisp is a language based on Church's lambda calculus [13] and is very mathematical in its structure. However, it is much different from the programming languages with which I was familiar, such as Fortran, BASIC, COBOL, or even a programmable calculator. Perhaps the best way to show the difference is to find something fairly simple, like the nth Fibonacci number, in Fortran and Lisp. In Fortran, this can be done using a do loop, with the result being in IFIB (and IFIBM1) at the end of the loop.

```

      IFIBM1 = 1
      IFIBM2 = 1
      DO 10 I = 2, N
          IFIB = IFIBM1 + IFIBM2
          IFIBM2 = IFIBM1
          IFIBM1 = IFIB
      10 CONTINUE

```

This has a very definite assignment and a very definite looping structure. Pure functional Lisp, on the other hand, has no assignment, and the looping is done by recursion. In Lisp, the code for finding the nth Fibonacci number is

```

(defun fib (lambda (n)
  (cond
    ((eq n 0) 1)
    ((eq n 1) 1)
    (t (plus (fib (difference n 1)) (fib (difference n 2))))
  )
)).

```

The main evaluation structure in Lisp is the definition of a function, which we have done here. One specifies the arguments (here n is the only one) and then gives the definition. If n is 0 or 1, then the value 1 is returned. Otherwise, what is returned is the sum of fib of $n-1$ and fib of $n-2$, which will require more steps into the recursion to find.

Lisp is very similar to the functional programming capability of Mathematica and I enjoyed comparing programs in the two languages. I also enjoyed writing Lisp programs as they forced me to think more mathematically. In running some existing Lisp programs, I was introduced to the n -queens problem (is there a way to place n queens on an n by n board so that all queens are safe from capture?). Seeing the 92 solutions the program found for the 8 queens problem made me curious how the solutions were related to each other under rotations and flips (an interesting twist on the standard rigid motions of a square problem which I hope to be able to examine in the future).

There are many advantages of the using pure functional Lisp instead of a language such as Fortran. Because Lisp has very few language constructs, the transformational procedure is greatly facilitated. According to Boyle, "Writing a pure Lisp specification ... is certainly easier, more natural, and more amenable to verification than is writing the Fortran version directly" [7, p. 407]. The functional capability of Lisp ensures that each call to the same function with the same arguments will produce the same result (as a mathematical function should). Thus the result of one function evaluation will affect another function evaluation only if the first function result is an argument of the second function. It is not unusual for a Lisp function to have multiple Lisp functions as its arguments (as in the Fibonacci example). Therein lies the parallelism that is exploited in the conversion to

Fortran. Each argument that is a function evaluation can be evaluated independently of the others. This is what parallel evaluation means: two or more tasks can be done independently of each other and hence done in parallel. This parallelism is obvious in a Lisp program, but is often very difficult to find in a standard Fortran program (see [7] for the resulting parallel Fortran code as compared to the Lisp code for the Fibonacci example above).

The main data structures in Lisp are atoms and lists (the actual data structure is not needed in writing the specification, but is needed when considering storage requirements and how the program is transformed to Fortran). Lists are much like a set and have unspecified length (as opposed to Fortran, where the length of an array must be prespecified). This fact makes the storage representation in Lisp much different from that in Fortran. The memory of the computer is segmented into regions, each region holding certain types of data. See Figure 3 for a diagram of this memory for the case of a single processor.

Each list is stored by storing the addresses or pointers of the two parts: the car, or first part of the list and the cdr, or rest of the list. For many operations all that is needed is information about the car or cdr (such as, is it an atom, a list, or nil?). This can be determined from the examining only the address due to the regions of the memory. Each atom is stored in only one spot, even if it appears in multiple lists. Arithmetic operations with integers have to take account for the fact that zero is stored at the address numadd, so one cannot just add the addresses of two integers to find their sum.

It is the storage of lists and atoms that makes the distributed-memory parallel version more difficult than the global-memory parallel version (where the storage of lists and atoms is just like the single processor version). To access data, one

must know not only the address, but the processor, or node, that has the data. Thus the data must be split into two pieces, one piece indicating the node and the other the address on that node.

If one processor needs data that another one has, a message must be sent to request that data, and the first processor must wait for the return of the data (or suspend that operation and work on another task until the data is available). However, to ensure that the same atom is only stored once, all atoms must be stored on one processor. The data representation that was finally agreed upon is given in Figure 4. Integers (and real numbers when implemented) will have an address of themselves and will be stored with a nodeid of intrnode (or realnode) to distinguish them from lists with the same address. With this representation, the three basic questions (do we have an atom, list, or nil?) can still be determined by examining the address and/or the nodeid.

5. Conclusions. Transformation systems are beginning to pay dividends after years of development. With new computer technology developing almost more rapidly than it can be used, maintaining a library of programs that is both accurate and efficient for the new technology is a formidable challenge. Program transformation techniques can prove very useful to this end.

The field of parallel computing is still open for many research topics, in terms of both software and hardware development. I encountered parallel machines with eight and 512 processors at Argonne. Machines with 1000 or more processors, the so-called massively parallel machines, are being studied separately from the "smaller" parallel machines (in October 1992 the Fourth Symposium on the Frontiers of Massively Parallel Computation was held in McLean, Virginia). There is a feeling among some that the real future of parallel computing lies in these

massively parallel machines:

"A machine that is massively parallel - that is, one that can handle very large numbers of operations concurrently - is essential for creating the advanced climate model that scientists hope to complete by the end of the decade." [2, p. 46]

Argonne has initiated a High Performance Computing and Communications Program that involves collaboration with IBM to produce a massively parallel machine capable of 1 teraflop (one trillion floating point operations per second) by 1995, about 1000 times faster than the iPSC (the eight processor machine at Argonne). Other DOE laboratories have or are proposing other High Performance Computing and Communications Programs. The dividends in parallel are just beginning to come in. The future of these areas is bright. I think we will all be amazed at the advances that will come about in the near future.

Figure 1
Global-Memory Parallel Processor

Figure 2
Distributed-Memory Parallel Processor

Figure 3

Single Memory Lisp Storage

Figure 4

Multi-Processor Memory Lisp Storage

References

1. Argonne National Laboratory, *Computational Science and Engineering at Argonne National Laboratory*.
2. Argonne National Laboratory, *Research Highlights 1991-92*.
3. J. J. Arsac, "Syntactic source to source transformation and program manipulation," *Communications of the ACM* 22(1) pp. 43-54 (Jan. 1979).
4. R. Balzer, N. Golman, and D. Wile, "On the transformational implementation approach to programming," *Proceedings 2nd International Conference on Software Engineering*, pp. 337-344 (1976).
5. F. L. Bauer, "Programming as an evolutionary process," *Proceedings 2nd International Conference on Software Engineering*, pp. 223-234 (1976).
6. J. M. Boyle and T. J. Harmer, "A practical functional program for the CRAY X-MP," ANL Preprint MCS-P159-0690, Argonne National Laboratory, Argonne, Illinois (July 1990).
7. J. M. Boyle, "Abstract programming and program transformation - an approach to reusing programs," *Software Reusability*, Vol. I, ed. Alan J. Perlis, ACM Press, Addison-Wesley Publishing Company, Reading, Massachusetts pp. 361-413 (1989).
8. J. M. Boyle, "Concrete syntax and semantics of TAMPR transformations," Preprint, Argonne National Laboratory, Argonne, Illinois (Sept. 1981).
9. J. M. Boyle, "Towards automated synthesis of linear algebra programs," *Production and Assessment of Numerical Software*, ed. M. A. Hennell and L. M. Delves, Academic Press, New York pp. 223-245 (1980).
10. R. M. Burstall and J. A. Darlington, "A transformation system for developing

- recursive programs," *Journal of the ACM* **24**(1), pp. 44-67 (Jan. 1977).
11. T. E. Cheatham and B. Wegbreit, "A laboratory for the study of automatic programming," *AFIPS Conference Proceedings*, SJCC **40**, pp. 11-21 (1972).
 12. E. Chiricozzi and A. D'Amico, eds., *Parallel Processing and Applications: Proceedings of the International Conference on Parallel Processing and Applications*, North Holland, New York (1987).
 13. A. Church, *The Calculi of Lambda Conversion*, Princeton University Press, Princeton, New Jersey (1941).
 14. V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J. J. Levy, "A structure-oriented program editor: A first step towards computer assisted programming," *International Computing Symposium*, pp. 113-120 (1975).
 15. K. W. Dritz and J. M. Boyle, "Beyond "speedup": Performance analysis of parallel programs," ANL-87-7, Argonne National Laboratory, Argonne Illinois (Jan. 1989).
 16. D. P. Friedman and M. Felleisen, *The Little Lisper*, 2nd ed., Science Research Associates, Inc., Chicago (1986).
 17. K. W. Hopkins, *Computer Program Transformation*, preprint Greenville College Library, Greenville, Illinois (1983).
 18. E. Lusk, R. Overbeek, J. Boyle, R. Butler, T. Disz, B. Glickfel, J. Patterson, and R. Stevens, *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, Inc., New York (1987).
 19. R. W. Numrich, *Supercomputer Applications*, Plenum Press, New York (1985).
 20. R. L. Stevens, "Argonne's high performance computing program," Seminar

- held July 16, 1992 at Argonne National Laboratory, Argonne, Illinois (1992).
21. J. L. Sanz, *Opportunities and Constraints of Parallel Computing*, Springer-Verlag, New York (1989).
 22. L. Smarr and C. E. Catlett, "Metacomputing," *Communications of the ACM*, **35**(6) pp. 45-52 (June 1992).
 23. J. Stoy, *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Massachusetts (1977).
 24. *Using the Intel iPSC/860*, Mathematics and Computer Science Division Technical Memorandum ANL/MCS-TM-143, Argonne National Laboratory, Argonne, Illinois (1990).
 25. R. C. Waters, "The programmer's apprentice: Knowledge based program editing," *IEEE Transactions on Software Engineering* **8**(1), pp. 1-12 (Jan. 1982).

DATE

FILMED

9 / 7 / 94

END