

Experiments with Program Parallelization Using Archetypes and Stepwise Refinement *

Berna L. Massingill¹

University of Florida, P.O. Box 116120, Gainesville, FL 32611
(352) 392-1260 FAX (352) 392-1220
blm@cise.ufl.edu

Abstract. Parallel programming continues to be difficult and error-prone, whether starting from specifications or from an existing sequential program. This paper presents (1) a methodology for parallelizing sequential applications and (2) experiments in applying the methodology. The methodology is based on the use of stepwise refinement together with what we call *parallel programming archetypes* (briefly, abstractions that capture common features of classes of programs), in which most of the work of parallelization is done using familiar sequential tools and techniques, and those parts of the process that cannot be addressed with sequential tools and techniques are addressed with formally-justified transformations. The experiments consist of applying the methodology to sequential application programs, and they provide evidence that the methodology produces correct and reasonably efficient programs at reasonable human-effort cost. Of particular interest is the fact that the aspect of the methodology that is most completely formally justified is the aspect that in practice was the most trouble-free.

1 Introduction

Much work has been done to make parallel programming tractable, but the development of parallel applications continues to be difficult and error-prone, whether the starting point is a specification or an existing sequential program. In this paper, we describe a methodology for parallelizing sequential applications based on the use of stepwise refinement together with what we call *parallel programming archetypes* (briefly, such an archetype is an abstraction that captures the commonality of a class of programs with similar structure), in which most of the work of parallelization is done using familiar sequential tools and techniques, and those parts of the process that cannot be addressed with sequential tools and techniques are addressed with formally-justified transformations. We then present the results of applying this methodology to a moderate-length application program, providing evidence that the methodology produces correct and reasonably efficient programs at reasonable human-effort cost. We focus attention on transforming programs for execution on distributed-memory–message-passing architectures, but we note that such programs may also be executed on architectures that support a shared-memory model.

* This work was supported by the AFOSR via an Air Force Laboratory Graduate Fellowship.

2 The methodology

Our methodology is based on the idea of transforming a sequential program into a parallel program by applying a sequence of small semantics-preserving transformations, guided by the pattern provided by a parallel programming archetype.

2.1 Parallel programming archetypes

By *parallel programming archetype* we mean an abstraction, similar to a design pattern, that captures the commonality of a class of programs. (An example of a sequential programming archetype is the familiar divide-and-conquer paradigm.)

Methods of exploiting design patterns in program development begin by identifying classes of problems with similar computational structures and creating abstractions that capture the commonality. Combining a problem class's computational structure with a parallelization strategy gives rise to a dataflow pattern and hence a communication structure. It is this combination of computational structure, parallelization strategy, and the implied pattern of dataflow and communication that we capture as a *parallel programming archetype*.

The commonality captured by the archetype abstraction makes it possible to develop a small collection of semantics-preserving transformations that together allow parallelization of any program that fits the archetype. Also, the common dataflow pattern makes it possible to encapsulate those parts of the computation that involve inter-process communication and transform them only once, with the results of the transformation made available as a *communication operation* usable in any program that fits the archetype.

2.2 Stepwise refinement and the sequential simulated-parallel version

A key feature of our methodology for parallelizing programs via a sequence of transformations is that all but the last of the transformations are performed in the sequential domain. Ideally all of the transformations would be formally stated and proved, with all but the final transformation proved using the techniques of sequential stepwise refinement. In the experiments described in this paper, however, we chose to focus our proof efforts on the final transformation — the one that takes the program into the parallel domain — since the sequential-to-sequential transformations are more amenable to checking by testing and debugging than the sequential-to-parallel transformation, and hence the case for a formal justification is more compelling for the former than for the latter. Thus, in this paper we give a formal justification only for this last transformation, in the form of a theorem applicable to all programs meeting certain stated criteria. The key intermediate stage in this transformation process is the last stage before transformation into the parallel domain; we call it the *sequential simulated-parallel version* of the program (since it essentially simulates the operation of a program consisting of N processes executing on a distributed-memory–message-passing architecture) and define it as follows:

Definition: Sequential simulated-parallel program. A sequential simulated-parallel program is one with the following characteristics:

1. The atomic data objects of the program are partitioned into N groups, one for each simulated process; the i -th group simulates the local data for the i -th process.
2. The computation consists of an alternating sequence of *local-computation blocks* and *data-exchange operations*, in which:
 - (a) Each *local-computation block* is a composition of N program blocks, in which the i -th block accesses only local data for the i -th simulated process. Such blocks correspond to sections of the parallel program in which processes execute independently and without interaction.
 - (b) Each *data-exchange operation* consists of a set of assignment statements that satisfy the following restrictions: (i) If an atomic data object is the target of an assignment, it is not referenced in any other assignment; (ii) no left-hand or right-hand side may reference atomic data objects belonging to more than one of the N simulated-local-data partitions, although the left-hand and right-hand sides of an assignment may reference data from different partitions; and (iii) for each simulated process i , at least one assignment statement must assign a value to a variable in i 's local data. Such blocks correspond to sections of the parallel program in which processes exchange messages: Each assignment statement can be implemented as a single point-to-point message-passing operation, and a group of message-passing operations with a common sender and a common receiver can be combined for efficiency.

Definition: Sequential simulated-parallel version. A sequential simulated-parallel version of program P is a sequential simulated-parallel program that refines P .

As we demonstrate in the next section, sequential execution of such a program truly simulates execution of the corresponding parallel program, in that all executions of the parallel program give the same results as an execution of the sequential program.

In general, producing such a simulated-parallel program could be tedious, time-consuming, and error-prone. However, in our methodology the transformation process is guided by an archetype, with the archetype providing, for programs in the class for which it is an abstraction, class-specific guidelines for program parallelization as well as class-specific code libraries encapsulating the communication and other operations required for parallelization. For a program that fits an archetype for which guidelines and a code library have been developed, the task of producing the simulated-parallel version is much more manageable, since the archetype guides the transformation process, and the program's data-exchange operations correspond to the archetype's communication operations, simplifying their transformation.

3 Supporting theory

In this section we present a general theorem allowing us to transform sequential simulated-parallel programs into equivalent parallel programs.

3.1 The parallel program and its simulated-parallel version

The target parallel program. The goal of the transformation process is a parallel program with the following characteristics:

1. The program is a collection of N sequential, deterministic processes.
2. Processes do not share variables; each has a distinct address space.
3. Processes interact only through sends and blocking receives on single-reader–single-writer channels with infinite slack (i.e., infinite capacity).
4. An execution is a fair interleaving of actions from processes.

The simulated-parallel version. We can simulate execution of such a parallel program as follows:

1. Simulate concurrent execution by interleaving actions from (simulated) processes.
2. Simulate separate address spaces by defining a set of distinct address-space data structures.
3. Simulate communication over channels by representing channels as queues, taking care that no attempt is made to read from a channel unless it is known not to be empty.

Figure 1 illustrates the relationship between the simulated-parallel and parallel versions of a program.

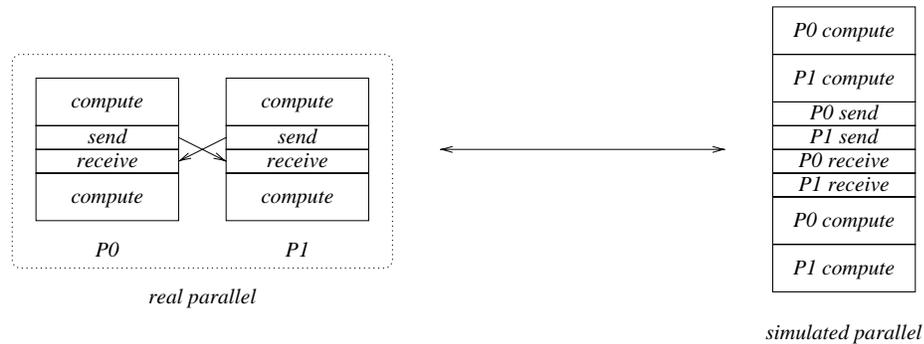


Fig. 1. Correspondence between parallel and simulated-parallel program versions.

3.2 The theorem

Theorem 1. Given deterministic processes P_0, \dots, P_{N-1} with no shared variables except single-reader–single-writer channels with infinite slack, if I and I' are two maximal interleavings of the actions of the P_j 's that begin in the same initial state, then I and I' both terminate, and in the same final state.

Proof. A complete proof of this theorem appears in [20]. The technique used is to show that given interleavings I and I' beginning in the same state, I' can be permuted to match I without changing its final state.

3.3 Implications and application of the theorem

Theorem 1 implies that if we can produce a sequential simulated-parallel program that meets the same specification as a sequential program, then we can mechanically convert it into a parallel program that meets that same specification, by transforming the simulated processes into real processes, the simulated multiple address spaces into real multiple address spaces, and the simulated communication actions into real communication actions (simulating channels using tagged point-to-point messages if necessary). In general producing such a simulated-parallel program could be quite difficult, but if we start with a program that fits an archetype, and produce a sequential simulated-parallel version of the form described in §2.2, where the data-exchange operations correspond to the communication operations of the archetype, then the task becomes manageable.

Each collection of assignments constituting a data-exchange operation can be replaced with a collection of sends and receives. Further, it is straightforward to choose an ordering for the simulated-parallel version that does not violate the restriction that we may not read from an empty channel, namely one in which all sends in a data-exchange operation are done before any receives. Finally, if the data-exchange operation corresponds to an archetype communication routine, it can be encapsulated and implemented as part of the archetype library of routines, which can be made available in both parallel and simulated-parallel versions. The application developer thus need not write out and transform the parts of the application that correspond to data-exchange operations.

4 Application experiments

The experiments described in this section consist of applying our methodology independently to two sequential implementations of an electromagnetics application. We describe the application, the archetype used to parallelize it, and the experiments.

4.1 The application

The application parallelized in this experiment is an electromagnetics code that uses the finite-difference time-domain (FDTD) technique to model transient electromagnetic scattering and interactions with objects of arbitrary shape and composition. With this technique, the object and surrounding space are represented by a 3-dimensional grid of computational cells. An initial excitation is specified, after which electric and magnetic fields are alternately updated throughout the grid. By applying a near-field to far-field transformation, these fields can also be used to derive far fields, e.g., for radar cross section computations. Thus, the application performs two kinds of calculations:

Near-field calculations. This part of the computation consists of a time-stepped simulation of the electric and magnetic fields over the 3-dimensional grid. At each time

step, we first calculate the electric field at each point based on the magnetic fields at the point and neighboring points, and then we similarly calculate the magnetic fields based on the electric fields.

Far-field calculations. This part of the computation uses the above-calculated electric and magnetic fields to compute radiation vector potentials at each time step by integrating over a closed surface near the boundary of the 3-dimensional grid. The electric and magnetic fields at a particular point on the integration surface at a particular time step affect the radiation vector potential at some future time step (depending on the point's position); thus, each calculated vector potential is a double sum, over time steps and over points on the integration surface.

Two versions of this code were available to us: Version A [17], which performs only the near-field calculations, and Version C [4], which performs both near-field and far-field calculations. The two versions were sufficiently different that we parallelized them separately, producing two parallelization experiments.

4.2 The archetype

In many respects this application is a very good fit for our *mesh archetype*, as described in this section.

Computational pattern. The pattern captured by the mesh archetype is one in which the overall computation is based on N -dimensional grids (where N is 1, 2, or 3) and structured as a sequence of the following operations on those grids:

Grid operations. Grid operations apply the same operation to each point in the grid, using data for that point and possibly neighboring points. If the operation uses data from neighboring points, the set of variables modified in the operation must be disjoint from the set of variables used as input.

Reduction operations. Reduction operations combine all values in a grid into a single value (e.g., finding the maximum element).

File input/output operations. File input/output operations read or write values for a grid.

Data may also include global variables common to all points in the grid (constants, for example, or the results of reduction operations), and the computation may include simple control structures based on these global variables (for example, looping based on a variable whose value is the result of a reduction).

Parallelization strategy and dataflow. Devising a parallelization strategy for a particular archetype begins by considering how its dataflow pattern can be used to determine how to distribute data among processes in such a way that communication requirements are minimized.

For the mesh archetype, the dataflow patterns of the archetype's characteristic operations lend themselves to a data-distribution scheme based on partitioning the data grid into regular contiguous subgrids (local sections) and distributing them among processes.

Grid operations. Provided that the previously-described restriction is met, points can be operated on in any order or simultaneously. Thus, each process can compute (sequentially) values for the points in its local section of the grid, and all processes can operate concurrently.

Reduction operations. Provided that the operation used to perform the reduction is associative or can be so treated, reductions can be computed concurrently by allowing each process to compute a local reduction result and then combining them, for example via recursive doubling (as described in, e.g., [22]).

File input/output operations. Exploitable concurrency depends on considerations of file structure and (perhaps) system-dependent I/O considerations. One possibility is to define a separate *host* process responsible for file I/O. A read operation then requires that the host process read the data from the file and then redistribute it to the other (*grid*) processes, while a write operation requires that the data first be redistributed from the grid processes to the host process and then written to the file. Another possibility is to perform I/O concurrently in all processes.

Each of these operations may incorporate or necessitate communication operations, as discussed below. Further, distributed memory introduces the requirement that each process have a duplicate copy of any global variables, with their values kept synchronized — that is, any change to such a variable must be duplicated in each process before the value of the variable is used again. The guidelines and transformations provided by the archetype ensure that these requirements are met.

Communication patterns. This combination of data-distribution scheme and computational operations gives rise to the need for a small set of communication operations:

Exchange of boundary values. If a grid operation uses values from neighboring points, computing new values for points on the boundary of each local section requires data from neighboring processes' local sections. This dataflow requirement can be met by surrounding each local section with a *ghost boundary* containing shadow copies of boundary values from neighboring processes, and using a boundary-exchange operation to refresh these shadow copies.

Broadcast of global data. When global data is computed or changed in one process only, a broadcast operation is required to re-establish copy consistency.

Support for reduction operations. Reduction operations can be supported by several communication patterns depending on their implementation — for example, all-to-one/one-to-all or recursive doubling.

Support for file input/output operations. Support for file input/output operations consists of redistribution operations that copy data from a single host process to the remaining (*grid*) processes and vice versa.

Implementation. We have developed for this archetype an implementation consisting of program-transformation guidelines, together with a code skeleton and an archetype-specific library of communication routines. The code skeleton and library are Fortran-based, with versions in Fortran M [10], Fortran with p4, and Fortran with NX. The implementation is described in detail in [19].

4.3 Parallelization strategy

As noted, in most respects our target application fits the pattern of the mesh archetype. The near-field calculations are a perfect example of this archetype and thus can be readily parallelized; all that is required is to partition the data and the computation and insert calls to boundary-exchange communication routines.

The far-field calculations fit the archetype less well and are thus more difficult to parallelize. The simplest approach to parallelization involves reordering the sums being computed: Each process computes local double sums (over all time steps and over points in its subgrid); at the end of the computation, these local sums are combined. The effect is to re-order, but not otherwise change, the summation. This method has the advantages of being simple and readily implemented using the mesh archetype (mostly local computations, with one reduction operation at the end). It has the disadvantage of not being guaranteed to produce the same result as the original program, since floating-point arithmetic is not truly associative. Nonetheless, because of its simplicity, we chose this method for an initial parallelization.

4.4 Applying our methodology

Determining how to apply the strategy. First, we determined how to apply the parallelization strategy, guided by documentation [19] for the mesh archetype, as follows:

1. Identify which variables should be distributed (among grid processes) and which duplicated (across all processes). For those variables that are to be distributed, determine which ones should be surrounded by a ghost boundary. Conceptually partition the data to be distributed into local sections, one for each grid process.
2. Identify which parts of the computation should be performed in the host process and which in the grid processes, and also which parts of the grid-process computation should be distributed and which duplicated. Also identify any parts of the computation that should be performed differently in the individual grid processes (e.g., calculations performed on the boundaries of the grid).

Generating the simulated sequential-parallel version. We then applied the following transformations to the original sequential code to obtain a simulated sequential-parallel version, operating separately on the two versions of the application described in §4.1:

1. In effect partition the data into distinct address spaces by adding an index to each variable. The value of this index constitutes a simulated process ID. At this point all data (even variables that are eventually to be distributed) is duplicated across all processes.
2. Adjust the program to fit the archetype pattern of blocks of local computation alternating with data-exchange operations.
3. Separate each local-computation block into a simulated-host-process block and a simulated-grid-process block.
4. Separate each simulated-grid-process block into the desired N simulated-grid-process blocks. This implies the following changes:

- (a) Modify loop bounds so that each simulated grid process modifies only data corresponding to its local section. This step was complicated by the fact that loop counters in the original code were used both to index arrays that were to be distributed and to indicate global position, and only the first of these uses must be changed in this step.
- (b) If there are calculations that must be done differently in different grid processes (e.g., boundary calculations), ensure that each process performs the appropriate calculations.
- (c) Insert calls to archetype library routines for data-exchange operations.

The result of these transformations was a sequential simulated-parallel version of the original program.

Generating the parallel program. Finally, we transformed this sequential simulated-parallel version into a program for message-passing architectures, as described in §3.3.

4.5 Results

Correctness. For those parts of the computation that fit the mesh archetype — the near-field calculations — the sequential simulated-parallel version produced results identical to those of the original sequential code. For those parts of the computation that did not fit well — the far-field calculations — the sequential simulated-parallel version produced results markedly different from those of the original sequential code. Our original assumption that we could regard floating-point addition as associative and thus reorder the required summations without markedly changing their results proved to be incorrect². Correct parallelization of these calculations would thus require a more sophisticated strategy than that suggested by the mesh archetype, which we did not pursue due to time constraints. While disappointing, this result does not invalidate our methodology; what it invalidates is our convenient but incorrect assumption about the associativity of floating-point addition for the data involved.

For all parts of the computation, however, the message-passing programs produced results identical to those of the corresponding sequential simulated-parallel versions, on the first and every execution.

Performance. Both versions of the application were parallelized using our Fortran M implementation of the mesh archetype. Table 1 shows execution times and speedups³ for Version C, executing on a network of Sun workstations. Figure 2 shows execution times and speedups for Version A, executing on an IBM SP. Comparable performance figures for other problem sizes are given in [20].

² Analysis of the values involved showed that they ranged over many orders of magnitude, so it is not surprising that the result of the summation was markedly affected by the order of summation.

³ We define speedup as execution time for the original sequential code divided by execution time for the parallel code.

	Execution time (seconds)	Speedup
Sequential	78.6	1.00
Parallel, $P=1$	189.0	0.41
Parallel, $P=2$	51.4	1.52
Parallel, $P=4$	25.3	3.10

Table 1. Execution times and speedups for electromagnetics code (version C), for 33 by 33 by 33 grid, 128 steps, using Fortran M on a network of Suns.

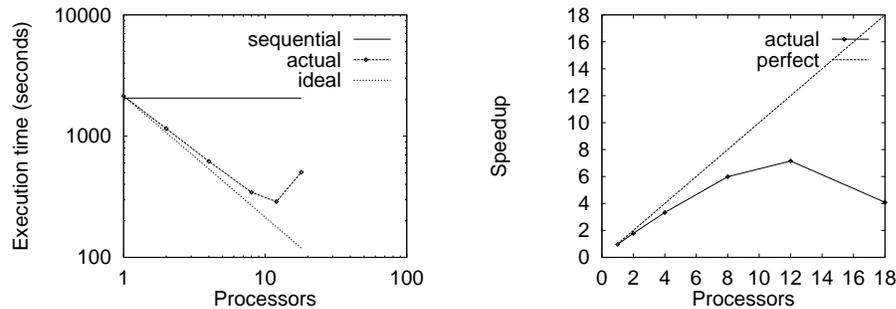


Fig. 2. Execution times and speedups for electromagnetics code (version A) for 66 by 66 by 66 grid, 512 steps, using Fortran M on the IBM SP.

Ease of use. It is difficult to define objective measures of ease of use, but our experiences in the experiments described in this chapter suggest that the parallelization methodology described herein produces results at a reasonable human-effort cost: Starting in both cases with unfamiliar code (about 2400 lines for Version C and 1400 lines for Version A, including comments and whitespace), we were able to perform the transformations described in §4.4 relatively quickly: For version C of the code, one person spent 2 days determining how to apply the mesh-archetype parallelization strategy, 8 days converting the sequential code into the sequential simulated-parallel version, and less than a day converting the sequential simulated-parallel version into a message-passing version. For version A of the code, one person spent less than a day determining how to apply the parallelization strategy, 5 days converting the sequential code into the sequential simulated-parallel version, and less than a day converting the sequential simulated-parallel version into a message-passing version.

5 Related work

Stepwise refinement. Program development via stepwise refinement is described by many researchers, for example Back [2], Gries [12], and Hoare [15] for sequential programs, and Back [1], Martin [18], and Van de Velde [22] for parallel programs.

The transformation of assignment statements into message-passing operations central to Theorem 1 is treated by Hoare [16] and Martin [18]. We present a more complete model of parallel programming that supports this and other experimental work in [20].

Automatic parallelizing compilers. Much effort has gone into development of compilers that automatically recognize potential concurrency and emit parallel code, for example Fortran D [9] and HPF [14]. We regard this work as complementary to our methodology and postulate that such compilers could automate some parts of our transformation process.

Design patterns. Many researchers have investigated the use of patterns in developing algorithms and applications. Our previous work [6, 7, 8] explores a more general notion of archetypes and their role in developing both sequential and parallel programs. Gamma et al. [11] address primarily the issue of patterns of computation, in the context of object-oriented design. Schmidt [21] focuses more on parallel structure, but with less emphasis on code reuse. Brinch Hansen’s work [5] is similar in motivation to our work, but his model programs are typically more narrowly defined. Other work addresses lower-level patterns, as for example the use of templates to develop algorithms for linear algebra in [3] and the use of templates in developing formally-verified software in [13].

6 Conclusions

This paper describes experiments with a methodology for parallelizing sequential programs based on applying the techniques of stepwise refinement under the guidance of a parallel programming archetype. The experiments demonstrate that the methodology (when applied in the proper context) produces programs that are correct and reasonably efficient at what seems to be reasonable human-effort cost. It is particularly heartening to note that the transformation for which we provide formal support — the transformation from sequential simulated-parallel to parallel — produced correct results in practice as well as in theory, with the resulting parallel programs producing results identical to those of their simulated-parallel predecessors on the first, and all subsequent, executions.

Much work remains to be done — identifying and developing additional archetypes, formally stating and proving additional transformations ([20] presents additional examples but is not exhaustive), and providing automatic support for transformations where feasible — but we are encouraged by our results so far.

Acknowledgments

Special thanks go to Eric Van de Velde, whose book [22] inspired this work, and to John Beggs, who provided and explained the electromagnetics application that was the subject of our experiments.

References

1. R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer-Verlag, 1990.
2. R. J. R. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic programs. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer-Verlag, 1990.
3. R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1993.
4. J. H. Beggs, R. J. Luebbers, D. Steich, H. S. Langdon, and K. S. Kunz. User’s manual for three-dimensional FDTD version C code for scattering from frequency-independent dielectric and magnetic materials. Technical report, The Pennsylvania State University, July 1992.
5. P. Brinch Hansen. Model programs for computational science: A programming methodology for multicomputers. *Concurrency: Practice and Experience*, 5(5):407–423, 1993.
6. K. M. Chandy. Concurrent program archetypes. In *Proceedings of the Scalable Parallel Library Conference*, 1994.
7. K. M. Chandy, R. Manohar, B. L. Massingill, and D. I. Meiron. Integrating task and data parallelism with the group communication archetype. In *Proceedings of the 9th International Parallel Processing Symposium*, 1995.
8. K. M. Chandy and B. L. Massingill. Parallel program archetypes. Technical Report CS-TR-96-28, California Institute of Technology, 1997.
9. K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The Parascope parallel programming environment. *Proceedings of the IEEE*, 82(2):244–263, 1993.
10. I. T. Foster and K. M. Chandy. FORTRAN M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26(1):24–35, 1995.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
12. D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
13. D. Hemer and P. Lindsay. Reuse of verified design templates through extended pattern matching. Technical Report 97-03, Software Verification Research Centre, School of Information Technology, The University of Queensland, 1997. To appear in *Proceedings of Formal Methods Europe (FME '97)*.
14. High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. *Scientific Programming*, 2(1–2):1–170, 1993.
15. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.
16. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
17. K. S. Kunz and R. J. Luebbers. *The Finite Difference Time Domain Method for Electromagnetics*. CRC Press, 1993.
18. A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.
19. B. Massingill. The mesh archetype. Technical Report CS-TR-96-25, California Institute of Technology, 1997. Also available via <http://www.etext.caltech.edu/Implementations/>.
20. B. Massingill. A structured approach to parallel programming (Ph.D. thesis). Technical Report CS-TR-98-04, California Institute of Technology, 1998.

21. D. C. Schmidt. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, 38(10):65–74, 1995.
22. E. F. Van de Velde. *Concurrent Scientific Computing*. Springer-Verlag, 1994.