Parallel Program Archetypes [*]

Berna L. Massingill[†] and K. Mani Chandy
California Institute of Technology, m/c 256-80
Pasadena, California 91125
{berna,mani}@cs.caltech.edu

Abstract

A parallel program archetype *is an abstraction that captures the common features of a class of problems with similar computational structure and combines them with a parallelization strategy to produce a pattern of dataflow and communication. Such abstractions are useful in application development, both as a conceptual framework and as a basis for tools and techniques. This paper describes an approach to parallel application development based on archetypes and presents two example archetypes with applications.*

1. Introduction

1.1. Overview

This paper proposes a specific method of exploiting computational and dataflow patterns to help in developing reliable parallel programs. A great deal of work has been done on methods of exploiting design patterns in program development. This paper restricts attention to one kind of pattern that is relevant in parallel programming: the pattern of the parallel computation and communication structure.

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*, or just an *archetype*.

In this paper we describe our overall strategy for archetype-based application development, and then present an example archetype and show how it can be used to develop applications. (Additional examples of archetypes and applications are presented in [15].) We observe that our work to date has concentrated on target architectures with distributed memory and message-passing, and this paper reflects this focus, but we believe that the work has applicability for shared-memory architectures as well.

1.2. Previous work

Much previous work also addresses the identification and exploitation of patterns: The idea of design patterns, especially for object-oriented design, has received a great deal of attention (e.g., [11, 18]). Libraries of program skeletons for functional and other programs have been developed [2, 4, 6]. Algorithm templates of the more common linear algebra programs have been developed and then used in designing programs for parallel machines [1]. Parallel structures have been investigated by many other researchers [3, 9]. Structuring parallel programs by means of examining dataflow patterns has also been investigated [8].

Our contribution is to show that combining consideration of broadly-defined computational patterns with dataflow considerations is useful in the systematic development of efficient parallel programs in a variety of widely-used languages such as Fortran and C.

1.3. Archetype-based help for developers

Although the dataflow pattern is the most significant aspect of an archetype in terms of its usefulness in easing the task of developing parallel programs, including computational structure as part of the archetype abstraction helps in identifying the dataflow pattern and also provides some of the other benefits associated with patterns. Just as a pattern does, an archetype abstracts common features of a class of programs; it can thus serve as a framework for thinking about programs in its class and can also serve as a basis for

[*]This work was supported in part by the AFOSR under grant number AFOSR-91-0070, and in part by the NSF under Cooperative Agreement No. CCR-9120008. The government has certain rights in this material.

[†]Present address: Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611-6120; blm@cise.ufl.edu.

a variety of class-specific guidelines and tools, such as the following.

Reusable code. Program skeletons and code libraries can be developed for each archetype, such that an application programmer creates an application by fleshing out the sequential aspects of the appropriate archetype's skeleton, making use of its library. The skeletons and libraries can be implemented and tuned for a variety of target architectures, thereby improving portability and performance for all applications based on them.

Parallelization help. Programmers often transform sequential programs to execute efficiently on parallel machines. The process of transformation can be laborious and error-prone, but it can be systematized for sequential programs that fit specific computational patterns, so that if a sequential program fits one of these patterns (archetypes), the transformation steps appropriate to that pattern can be used. Exploitation of the pattern can make the transformation more systematic, more mechanical, and better suited to automation. Also, in some cases, parallelizing compilers can generate programs that execute more efficiently on parallel machines if programmers provide information about their programs in addition to the program text itself; an archetype could provide guidelines for providing such information.

Other assistance. An archetype provides a framework for reasoning about application programs. It can also be a basis for class-specific guidelines for testing, as well as for class-specific performance modeling, as described in [17].

2. An archetype-based development strategy

Our general strategy for writing programs using archetypes has the goal of allowing application developers to do as much of their work as possible using familiar sequential tools and techniques. Application development can start with either a sequential algorithm or simply a problem description; it then proceeds as follows.

Archetype selection. The application developer identifies an appropriate archetype by comparing the problem with archetype documentation. (We envision a library of archetypes, each with a description and example applications that help developers decide whether it fits their problems.)

Initial development. The developer then creates an initial archetype-based version of the algorithm, structured according to the archetype's pattern and giving an indication of the concurrency to be exploited by the archetype, by structuring the original algorithm to fit the archetype pattern and "filling in the blanks" of the archetype with application-specific details. Transforming the original algorithm into this archetype-based equivalent can be done in one stage or via a sequence of smaller transformations, in either case guided by the archetype pattern. An important feature of this initial archetype-based version of the algorithm is that it can be readily mapped to an equivalent sequential program (as described in the examples); for deterministic programs, this sequential equivalent gives the same results as the parallel program-in-work, allowing debugging to be done in the sequential domain.

Transformation for efficiency. The developer next transforms the initial archetype-based version of the algorithm into an equivalent algorithm suitable for efficient execution on the target architecture. The archetype assists in this transformation, either via guidelines to be applied manually or via automated tools. Again, the transformation can optionally be broken down into a sequence of smaller stages, and in some cases intermediate stages can be executed (and debugged) sequentially. A key aspect of this transformation process is that the transformations defined by the archetype preserve semantics and hence correctness.

Implementation. Finally, the developer implements the efficient archetype-based version of the algorithm using a language or library suitable for the target architecture. Here again the archetype assists in this process, not only by providing suitable transformations (either manual or automatic), but also by providing program skeletons and libraries that encapsulate most of the details of the parallel code (process creation, message-passing, and so forth). A significant aspect of this step is that it is only here that the application developer must choose a particular language or library; the algorithm versions produced in the preceding steps can be expressed in any convenient notation, since the ideas are essentially language-independent.

3. Example archetype: Mesh-spectral

3.1. Computational pattern

A number of scientific computations can be expressed in terms of operations on N-dimensional grids. While it is possible to abstract from such computations patterns resembling higher-order functions (like that of divide and conquer, for example), our experience with real-world applications suggests that such patterns tend to be too restrictive and inflexible to address any but the simplest problems. Instead, the pattern captured by the mesh-spectral archetype[1] is one in which the overall computation is based on N-dimensional grids (N is usually 1, 2, or 3) and structured as a sequence of the following operations on those grids.

Grid operations, which apply the same operation to each point in the grid, using data for that point and possibly

[1]We call this archetype "mesh-spectral" because it combines and generalizes two earlier archetypes, a mesh archetype focusing on grid operations, and a spectral-methods archetype focusing on row and column operations.

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. Input variables may also include "global" variables (variables common to all points in the grid, e.g., constants).

Row (column) operations, which apply the same operation to each row (column) in the grid. (Analogous operations can be defined on subsets of grids with more than 2 dimensions.) The operation must be such that all rows (columns) are operated on independently; that is, the calculation for row i cannot depend on the results of the calculation for row j, where $i \neq j$.

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

File input/output operations, which 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).

3.2. Parallelization strategy and dataflow

Most of the operations that characterize this archetype have obvious exploitable concurrency, given the data-dependency restrictions described in §3.1 (e.g., for row operations, results for row i cannot depend on results for row j), and they lend themselves to parallelization based on the strategy of partitioning the data grid into regular contiguous subgrids (local sections) and distributing them among processes. As described in this section, some operations impose requirements on how the data is distributed, while others do not. All operations assume that they are preceded by the equivalent of barrier synchronization.

Grid operations. Provided that the restriction in §3.1 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. Grid operations impose no restrictions on data distribution, although the choice of data distribution may affect the resulting program's efficiency. (The relationship between data distribution and efficiency is important but orthogonal to the concerns of this paper.)

Row (column) operations. Provided that the restriction in §3.1 is met, rows can be operated on simultaneously or in any order. These operations impose restrictions on data distribution: Row operations require that data be distributed by rows, while column operations require that data be distributed by columns.

Reduction operations. Provided that the operation used to perform the reduction is associative (e.g., maximum) or can be so treated (e.g., floating-point addition, if some degree of nondeterminism is acceptable), reductions can be computed concurrently by allowing each process to compute a local reduction result and then combining them, for example via recursive doubling. Reduction operations, like grid operations, may be performed on data distributed in any convenient fashion. After completion of a reduction operation all processes have access to its result; this is to be guaranteed by the implementation.

File input/output operations. Exploitable concurrency and appropriate data distribution depend on considerations of file structure and possibly system-dependent I/O considerations. One approach is to operate on all data sequentially in a single process, which implies a data "distribution" in which all data is collected in a single process. Another approach is to perform I/O "concurrently" in all processes (actual concurrency may be limited by system or file constraints), using any convenient data distribution.

Patterns of dataflow arise as a consequence of how the above operations are composed to form an individual algorithm; if two operations requiring different data distributions are composed in sequence, they must be separated by data redistribution (for distributed memory). Distributed memory introduces the additional 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. A key element of this archetype is support for ensuring that these requirements are met. This support can take the form of guidelines for manually transforming programs, as in our archetype-implementation user guides [7, 12], or it could be expressed in terms of more formal transformations with proofs of their correctness, as described in [14].

3.3. Communication patterns

The dataflow patterns just described give rise to the need for the following communication operations.

Grid redistribution. If different parts of the computation require different distributions (e.g., if a row operation is followed by a column operation), data must be redistributed among processes.

Exchange of boundary values. If a grid operation uses values from neighboring points, points on the boundary of each local section will require 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 (in which neighboring processes exchange boundary values) to refresh these shadow copies.

Broadcast of global data. When global data is computed or changed in one process only (e.g., if it is read from a file), a broadcast operation is required to re-establish copy consistency.

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

Support for file input/output operations. File input/output operations can be supported by several communication patterns, e.g., data redistribution (one-to-all or all-to-one).

All of the required operations can be supported by a communication library containing a boundary-exchange operation, a general data-redistribution operation, and a general reduction operation. It is straightforward to write down specifications of these operations in terms of pre- and post-conditions (which is helpful in determining where they should be used); these specifications can then be implemented in any desired language or library as part of an archetype implementation.

3.4. Applying the archetype

The key difficulties in applying the mesh-spectral archetype to an algorithm have to do with converting the initial archetype-based version to an architecture-specific version. These difficulties are most pronounced when the target architecture imposes requirements for distributed memory and message-passing, but similar transformations may produce more efficient programs for other architectures (e.g., non-uniform-memory-access multiprocessors) as well. However, because the specific transformations required by an application are instances of patterns captured by the archetype, this conversion process is easier to perform than a more general conversion from sequential to parallel or from shared-memory to distributed-memory. Further, the required communication operations can be encapsulated and implemented in a reusable form, thereby amortizing the implementation effort across multiple applications. The example in §3.5 illustrates how this archetype can be used to develop algorithms and transform them into versions suitable for execution on a distributed-memory message-passing architecture. An additional example is presented in [15].

3.5. Application example: Poisson solver

This example makes use of grid variables, a reduction operation, and the use of a global variable for control flow; it illustrates how the archetype guides the process of transforming a sequential algorithm into a program for a distributed-memory message-passing architecture.

Problem description. The problem (as described in [19]) is to find a numerical solution to the Poisson problem

$$-\frac{\partial^2 U}{\partial x^2} - \frac{\partial^2 U}{\partial y^2} = f(x, y)$$

with Dirichlet boundary condition

$$u(x, y) = g(x, y)$$

using discretization and Jacobi iteration; i.e., by discretizing the problem domain and applying the following operation to all interior points until convergence is reached:

$$\begin{aligned} 4u^{(k+1)}_{(i,j)} &= h^2 f_{i,j} + u^{(k)}_{(i-1,j)} + u^{(k)}_{(i+1,j)} \\ &\quad + u^{(k)}_{(i,j-1)} + u^{(k)}_{(i,j+1)} \end{aligned}$$

A sequential program for this computation is straightforward: It maintains two copies of variable u, one for the current iteration (`uk`) and one for the next iteration (`ukp1`). The initial values of `uk` are given by g for points on the boundary of the grid and by an "initial guess" for points in the interior. Array `f` is used to store the values of f at the grid points. During each iteration, the program computes new values for the values of `ukp1` at each interior point based on the values of `uk` and `f`. It then computes the maximum (`diffmax`) of $|u^{(k+1)}_{(i,j)} - u^{(k)}_{(i,j)}|$ to check for convergence and then copies `ukp1` back to `uk`. (Avoiding this copying is possible and would produce a more efficient program, but at a cost in program readability and simplicity.)

Archetype-based algorithm, version 1. It is not difficult to see that the sequential algorithm described fits the pattern of the mesh-spectral archetype: The data consists of several grids (`uk`, `ukp1`, and `f`) and a global variable `diffmax` that is computed as the result of a reduction operation and used in the program's control flow. Thus, it is straightforward to write down an archetype-based version of the algorithm, as shown in Figure 1 using a grid with dimensions `NX` by `NY`. Observe that since the iterations of each **forall** are independent, this algorithm can be executed (and debugged, if necessary) sequentially by replacing each **forall** with nested **do** loops. Observe also that this algorithm could be executed without change and with the same results on an architecture that supports the **forall** construct, since the iterations of the **forall** are independent and the reduction operation (a global maximum) is based on an associative operation.

Archetype-based algorithm, version 2. We next consider how to transform the initial version of the algorithm into a version suitable for execution on a distributed-memory message-passing architecture. The overall computation is to be expressed as an SPMD computation, with

```
    subroutine poisson(NX, NY)
    integer, intent(in) :: NX, NY
    real, dimension(NX, NY) :: uk, ukp1, f
    real :: diffmax

!---initialize boundary of u to g(x,y),
!   interior to initial guess
    call initialize(uk, f)
!---compute until convergence
    diffmax = TOLERANCE + 1.0
    do while (diffmax > TOLERANCE)
    !---compute new values
        !HPF$ INDEPENDENT
        forall (i = 2:NX-1, j = 2:NY-1)
            ukp1(i,j) = 0.25*(H*H*f(i,j)    &
                + uk(i,j-1) + uk(i,j+1)     &
                + uk(i-1,j) + uk(i+1,j))
        end forall
    !---check for convergence:
    !   compute max(abs(ukp1(i,j) - uk(i,j)))
        diffmax = maxabsdiff(               &
                ukp1(2:NX-1,2:NY-1),        &
                uk(2:NX-1,2:NY-1))
    !---copy new values to old values
        uk(2:NX-1, 2:NY-1) = ukp1(2:NX-1, 2:NY-1)
    end do ! while
    call print(uk)
    end subroutine poisson
```

Figure 1. Poisson solver, version 1.

the archetype supplying any code skeleton needed to create and connect the processes. Since the operations that make up the computation have no data-distribution requirements, it is sensible to write the program using a generic block distribution (distributing data in contiguous blocks among NPX*NPY processes conceptually arranged as an NPX by NPY grid); we can later adjust the dimensions of this process grid to optimize performance. Guided by the archetype (i.e., by the discussion of dataflow and communication patterns above), we can transform the algorithm of Figure 1 into an SPMD computation in which each process executes the pseudocode shown in Figure 2: The program's grids are distributed among processes, with each local section surrounded by a ghost boundary to contain the data required by the grid operation that computes ukp1. The global variable diffmax is duplicated in each process; copy consistency is maintained because each copy's value is changed only by operations that establish the same value in all processes (initialization and reduction). Each grid operation is distributed among processes, with each process computing new values for the points in its local section. (Observe that new values are computed only for points in the intersection of the local section and the whole grid's interior.) To satisfy the precondition of a grid operation using data from neighboring points, the computation of ukp1 is preceded by a boundary exchange operation. The reduction operation is transformed as described in §3.2; since a postcondition of this operation is that all processes have access to the result of the reduction, copy consistency is re-established for loop control variable diffmax before it is used. All of these transformations can be assisted by the archetype, via any combination of guidelines, formally-verified transformations, or automated tools that archetype developers choose to create. Observe also that most of the details of interprocess communication are encapsulated in the boundary-exchange and reduction operations, which can be provided by an archetype-specific library of communication routines, freeing the application developer to focus on application-specific aspects of the program.

```
    subroutine poisson_process(NX, NY, NPX, NPY)
    integer, intent(in) :: NX, NY, NPX, NPY
    real, dimension(0:(NX/NPX)+1, 0:(NY/NPY)+1) &
        :: uk, ukp1, f
    real :: diffmax, local_diffmax
    integer :: ilo, ihi, jlo, jhi

!---initialize boundary of u to g(x,y),
!   interior to initial guess
    call initialize_section(uk, f)
!---compute intersection of "interior" with
!   local section
    call xintersect(2,NX-1,ilo,ihi)
    call yintersect(2,NY-1,jlo,jhi)
!---compute until convergence
    diffmax = TOLERANCE + 1.0
    do while (diffmax > TOLERANCE)
    !---compute new values
        call boundary_exchange(uk)
        do j = jlo, jhi
        do i = ilo, ihi
            ukp1(i,j) = 0.25*(H*H*f(i,j)    &
                + uk(i,j-1) + uk(i,j+1)     &
                + uk(i-1,j) + uk(i+1,j))
        end do
        end do
    !---check for convergence:
    !   compute max(abs(ukp1(i,j) - uk(i,j)))
        local_diffmax = maxabsdiff(         &
                ukp1(ilo:ihi,jlo:jhi),      &
                uk(ilo:ihi,jlo:jhi))
        diffmax = reduce_max(local_diffmax)
    !---copy new values to old values
        uk(ilo:ihi,jlo:jhi) =               &
                ukp1(ilo:ihi,jlo:jhi)
    end do ! while
    call print_section(uk)
    end subroutine poisson_process
```

Figure 2. Poisson solver, version 2.

Implementation. Transformation of the algorithm shown in Figure 2 into code in a sequential language plus message-passing is straightforward, with most of the details encapsulated in the boundary-exchange and reduction routines. This algorithm has been implemented using a general mesh-spectral archetype implementation consisting of a code skeleton and an archetype-specific library of communication routines. This skeleton and library have in turn been implemented in both Fortran M [10] and Fortran with MPI [16]. The Fortran M version has been used to run applications on the IBM SP and on networks of Sun workstations; the MPI version has been used to run applications on the IBM SP and on networks of Sun and Pentium-based workstations. Figure 3 shows speedups of the MPI version of the parallel code compared to the equivalent sequential code, executed on the IBM SP. (In the figure, "perfect" speedup represents the best speedup obtainable without superlinear effects and is simply the number of processors.)

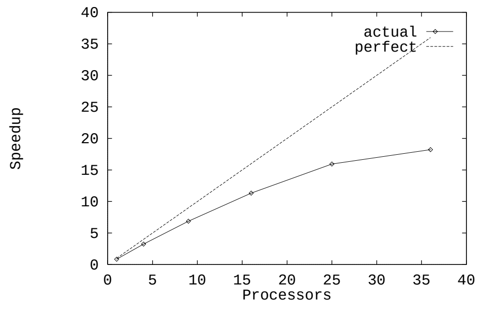

Figure 3. Speedup of parallel Poisson solver compared to sequential Poisson solver for 800 by 800 grid, 1000 steps, on the IBM SP.

3.6. Other application examples

In addition to the example applications in the preceding sections, we have developed a number of real-world applications based on the mesh-spectral archetype and its subsets the mesh and spectral archetypes.

Computation fluid dynamics. We have developed several CFD applications treating compressible and incompressible flows. These applications were run on a variety of platforms, including the Intel Paragon, the IBM SP, and networks of workstations; they are described in more detail in [15].

Electromagnetics. We parallelized two versions of an electromagnetic scattering simulation program, as described in [13]. The resulting programs were run on the IBM SP and networks of workstations.

Air-quality modeling. We developed several parallel versions of a program that models smog in the Los Angeles basin, conceptually based on the mesh-spectral archetype although not using its implementation. This application has been run on a number of platforms, including the Intel Delta, the Intel Paragon, the Cray T3D, and the IBM SP2; it is described in [5].

Based on our experiences with these applications, the key benefits of developing an algorithm using the mesh-spectral archetype are (i) the guidelines or transformations for converting the algorithm to a form suitable for the target architecture, and (ii) the encapsulated and reusable library of communication operations. The performance of the resulting programs is to a large extent dependent on the performance of this communication library, but our experience suggests that even fairly naive implementations of the communication library often give acceptable performance. Performance can then be improved by tuning the library routines, with potential benefit for other archetype-based applications.

4. Conclusions

Based on our experiences in developing archetypes, implementations, and applications, we believe that our proposed strategy for application development using archetypes is successful in many respects: An archetype eases the task of algorithm development by providing a conceptual framework for thinking about problems in the class it represents. It also eases the task of writing reliable parallel programs by providing guidelines or formally justified transformations for converting essentially sequential code into code for realistic parallel architectures. An archetype implementation (in the form of a code skeleton and a library of communication or other routines) eases the task of program development by encapsulating the explicitly parallel aspects of the program, allowing the programmer to focus on writing the (sequential) parts of the program that are specific to the application. Such an implementation also helps in writing efficient parallel programs by allowing the cost of optimizing communication operations to be amortized over several applications.

Although results so far are encouraging, much more could be done; directions for future work include the following: (i) developing a more comprehensive library of archetypes, with several examples for each archetype; (ii) developing a theory and strategy for archetype composition, to add the benefits of modular design to our approach; (iii) applying our work to shared-memory architectures as well as those with distributed memory and message-passing; and (iv) developing automated tools for applying archetype-guided program transformations.

Acknowledgments

Many people contributed to the work described in this paper. Thanks are due to Dan Meiron and Greg Davis for their help in defining and implementing the mesh-spectral archetype; and John Beggs, Donald Dabdub, Greg Davis, Rajit Manohar, Dan Meiron, and Ravi Samtaney, for their help in developing the applications described in §3.6.

References

[1] 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.

[2] G. H. Botorog and H. Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, 1996.

[3] P. Brinch Hansen. Model programs for computational science: A programming methodology for multicomputers. *Concurrency: Practice and Experience*, 5(5):407–423, 1993.

[4] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[5] D. Dabdub and R. Manohar. Parallel computation in atmospheric chemical modeling. *Parallel Computing*, 1997. To appear in special issue on regional weather models.

[6] J. Darlington, A. J. Field, P. O. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. White. Parallel programming using skeleton functions. In A. Bode, editor, *Proceedings of PARLE 1993*, volume 694 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[7] G. Davis and B. L. Massingill. The mesh-spectral archetype. Technical Report CS-TR-96-26, California Institute of Technology, 1996.

[8] D. C. Dinucci and R. G. Babb II. Development of portable parallel programs with Large-Grain Data Flow 2. In G. Goos and J. Hartmanis, editors, *CONPAR 90 — VAPP IV*, volume 457 of *Lecture Notes in Computer Science*, pages 253–264. Springer-Verlag, 1990.

[9] N. Fang. Engineering parallel algorithms. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, 1996.

[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] B. L. Massingill. The mesh archetype. Technical Report CS-TR-96-25, California Institute of Technology, 1996.

[13] B. L. Massingill. Experiments with program parallelization using archetypes and stepwise refinement. In *Proceedings of the Third International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'98)*, 1998.

[14] B. L. Massingill. A structured approach to parallel programming. Technical Report CS-TR-98-04, California Institute of Technology, 1998. PhD thesis.

[15] B. L. Massingill and K. M. Chandy. Parallel program archetypes. Technical Report CS-TR-96-28, California Institute of Technology, 1996.

[16] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3–4), 1994.

[17] A. Rifkin and B. L. Massingill. Performance analysis for archetypes. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, 1998.

[18] D. C. Schmidt. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, 38(10):65–74, 1995.

[19] E. F. Van de Velde. *Concurrent Scientific Computing*. Springer-Verlag, 1994.