Parallel Program Archetypes *

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

January 30, 1997

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

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*.

*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.

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., [24, 33]). Libraries of program skeletons for functional and other programs have been developed [7, 14, 18]. Algorithm templates of the more common linear algebra programs have been developed and then used in designing programs for parallel machines [4]. Parallel structures have been investigated by many other researchers [8, 22]. Structuring parallel programs by means of examining dataflow patterns has also been investigated [21]. 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, including Fortran and C.

1.1 Archetype-based assistance for application development

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. Such archetypes are useful in many ways:

- A program skeleton and code library can be created for each archetype, where the skeleton deals with process creation and interaction between processes, and the code library encapsulates details of the interprocess interaction. If a sequential program fits an archetype, then a parallel program can be developed by fleshing out the skeleton, making use of the code library. The fleshing-out steps deal with defining the *sequential* structure of the processes. Thus, programmers can focus their attention primarily on sequential programming issues.

- One way to achieve portability and performance is to implement common patterns of parallel structures — those for a particular archetype or archetypes — on different target architectures (e.g., multicomputers, symmetric multiprocessors, and non-uniform-memory-access multiprocessors), tuning the implementation to obtain good performance. The cost of this performance optimization effort is amortized over all programs that fit the pattern. Indeed, since an archetype captures computational structure as well as dataflow pattern, it is possible for an implementation of a dataflow pattern to support more than one archetype.

- Programmers often transform sequential programs to execute efficiently on parallel machines. The process of transformation can be laborious and error-prone. However, this transformation process can be systematized for sequential programs that fit specific computational patterns; then, 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.

- Just as the identification of computational patterns in object-oriented design is useful in teaching systematic sequential program design, identification of computational and dataflow patterns (archetypes) is helpful in teaching parallel programming.

- Similarly, just as the use of computational patterns can make reasoning about sequential programs easier by providing a framework for proofs of algorithmic correctness, archetypes can provide a framework for reasoning about the correctness of parallel programs. Archetypes can also provide frameworks for testing and documentation.

- 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. Although the focus of this paper is on active stepwise refinement by programmers and not on compilation tools, we postulate that the dataflow pattern is information that can be exploited by a compiler.

- Archetypes may also be helpful in developing performance models for classes of programs with common structure, as discussed in [32].

- Archetypes can be useful in structuring programs that combine task and data parallelism, as described in [12].

1.2 An archetype-based program development strategy

Our general strategy for writing programs using archetypes is as follows:

1. Start with a sequential algorithm (or possibly a problem description).

2. Identify an appropriate archetype.

3. Develop an initial archetype-based version of the algorithm. This initial version is structured according to the archetype's pattern and gives an indication of the concurrency to be exploited by the archetype. Essentially, this step consists of 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, it is guided by the archetype pattern.

An important feature of this initial archetype-based version of the algorithm is that it can be executed sequentially (by converting any exploitable-concurrency constructs to sequential equivalents, as described in the examples). For deterministic programs, this sequential execution gives the same results as parallel execution; this allows debugging in the sequential domain using familiar tools and techniques.

4. Transform 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.

5. Implement 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/or libraries that encapsulate some 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.

This paper presents two example archetypes and shows how they and this strategy can be used to develop applications. Our work to date has concentrated on target architectures with distributed memory and message-passing, and the discussion reflects this focus, but we believe that the work has applicability for shared-memory architectures as well.

2 Example: The one-deep divide-and-conquer archetype

2.1 Computational pattern

2.1.1 Traditional divide and conquer

Divide and conquer is a useful paradigm for solving a number of problems, including such diverse applications as sorting, searching, geometric algorithms, and matrix multiplication. Briefly, the divide-and-conquer approach works as follows: The original problem P is split into N subproblems $P_1, \ldots, P_N$. Each

subproblem P_i is solved (directly if it is sufficiently small, otherwise recursively) to produce subsolution S_i. The N subsolutions are combined to produce solution S to problem P.

In this paradigm, the subproblems into which the original problem is split can be solved independently, and a parallel program can be produced by exploiting this potential concurrency, as shown in figure 1: Every time the problem is split into concurrently-executable subproblems, a new process is created, until some threshold size is reached, whereupon the subproblem is solved sequentially, possibly using a sequential divide and conquer algorithm. Such programs can

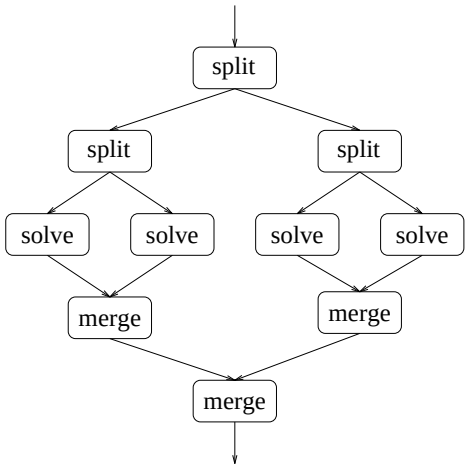

Figure 1: Parallelization of traditional divide and conquer.

be inefficient, however, for two reasons.

First, problem sizes for parallel programs are usually large, so data is often distributed among processes. Splitting the input data into small parts can require inspection of all the input data, which can be expensive in terms of data transfer and the amount of memory required for a single process. Similar considerations apply to the output data.

Second, as can be seen in figure 1, the amount of actual concurrency varies over the lifetime of the algorithm; only during the "solve" phase of the algorithm are the maximum number N of processes actually used, which can lead to inefficiency if N is large and the tree shown in figure 1 is deep.

2.1.2 "One-deep" divide and conquer

Both of these sources of inefficiency can be addressed by a modification of the original divide-and-conquer paradigm called "one-deep divide and conquer" [36].

First, we assume that the input data is originally distributed among processes, and that after execution the output data is to be distributed among processes as well.

Second, rather than splitting the problem into a small number of subproblems, which are then recursively split into smaller subproblems, and so forth, this version of divide and conquer performs only a single level of split/solve/merge (hence the name "one-deep"), splitting the original problem directly into N subproblems, solving them independently, and then merging the N subsolutions to obtain a solution to the original problem. That is, the algorithm is structured as follows:

1. Split problem P into N subproblems $P_1, \ldots, P_N$. Parameters for the split are computed using a small sample of the problem data; once these parameters are computed, collecting the data for the subproblems can be done independently (that is, data for P_i can be extracted from P independently of data for P_j).

2. Solve the subproblems (independently) using a sequential algorithm to produce subsolutions $S_1, \ldots, S_N$.

3. Merge the subsolutions into a solution S for P. This merge is accomplished by first repartitioning the subsolutions (based again on parameters computed using a small sample of data from all subsolutions) — i.e., rearranging subsolutions $S_1, \ldots, S_N$ into $S'_1, \ldots, S'_N$ — and then performing a local merge operation on each of the repartitioned subsolutions S'_i. Combining the results of these local merge operations (typically through concatenation) gives the total solution S.

For many problems, either the split or the merge step is degenerate — for example, no split is needed if the data is initially distributed in an acceptable way among processes.

Figure 2 illustrates this pattern. The details of how the parameters for the split or merge are computed can vary among algorithms and implementations; for example, it can be done in a single process, with the results then made known to all processes, or it can be done via identical computations in all processes concurrently.

2.2 Parallelization strategy and dataflow

Parallelization of the one-deep divide-and-conquer archetype is a straightforward exploitation of the obvious concurrency: During the split phase, once the parameters are computed, the actual split can be accomplished by N concurrent processes, each collecting data for one subproblem P_i. During the solve phase, all subproblems can be solved concurrently. During the merge phase, again once the parameters are computed, the merge can be accomplished by N concurrent

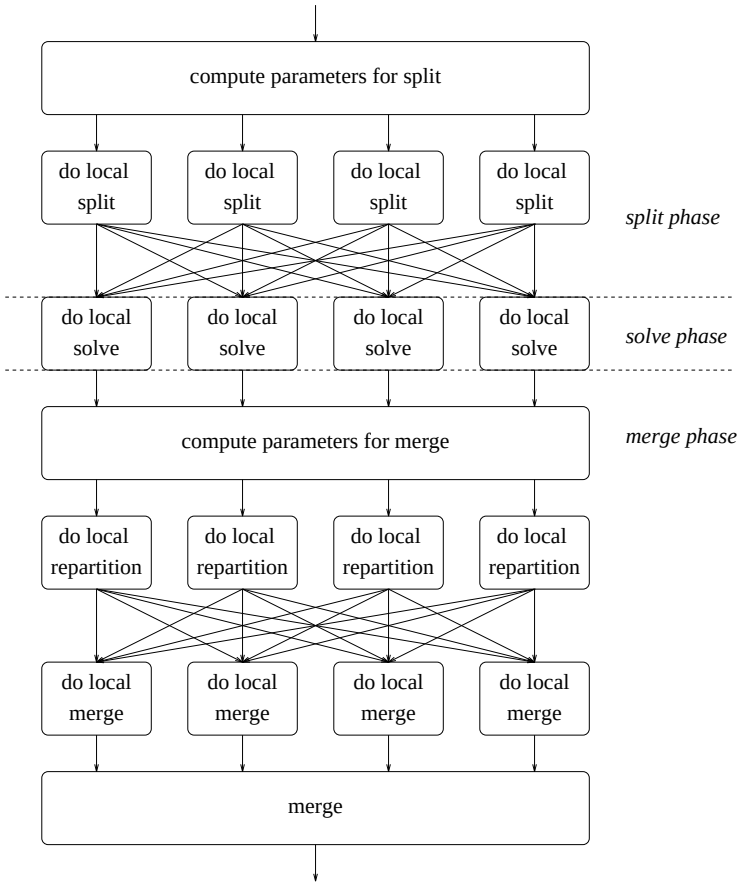

Figure 2: Parallelization of one-deep divide and conquer.

processes, each collecting and locally merging data for one subsolution S'_i. Depending on the details of the algorithm, the computation of parameters for the split and the merge can be either be done entirely sequentially (either by having one master process perform the computation and make its results available to the other processes, or by having all processes perform the same computation concurrently), or it can be done by first independently computing N sets of local parameters and then merging them sequentially.

The archetype's dataflow pattern arises from combining the above computational pattern with the desired distribution of problem data (input data at the start of the computation and output data at the end of the computation). Typically, the input and/or output data is structured as a one-dimensional array, in which the array elements can be scalars (as in mergesort) or a more complicated data structure (as in the skyline problem), with elements distributed among processes, with the details of the distribution depending on the problem. Figure 3 illustrates dataflow in the case in which the split phase is degenerate (i.e., the initial distribution of data among processes is used as the result of the split) and the merge phase consists of concatenation (i.e., the complete solution is the concatenation of the subsolutions). Analogous dataflow patterns arise for algorithms in which the split phase is nontrivial and the merge phase is degenerate.

2.3 Communication patterns

It is straightforward to infer the interprocess communication required for one-deep divide and conquer from dataflow patterns like the one of figure 3:

- All-to-all communication is needed to redistribute data during the split and merge phases, with every process p sending to every other process q a distinct portion of its data.
- Either (i) a combination of "gather" and broadcast or (ii) all-to-all communication is needed before the sequential part of the computation of split or merge parameters; each process that is to perform the computation must obtain data from every other process.
- Broadcast communication is needed after the computation of split or merge parameters if not all processes have performed the computation.

2.4 Applying the archetype

The one-deep divide-and-conquer archetype is most readily applied to algorithms that are already in the form of traditional recursive divide and conquer. The key to applying the archetype to such an algorithm lies in determining how to transform the recursive (often two-way) split and merge of the original algorithm into an N-way one-deep split and merge. It is difficult to automate

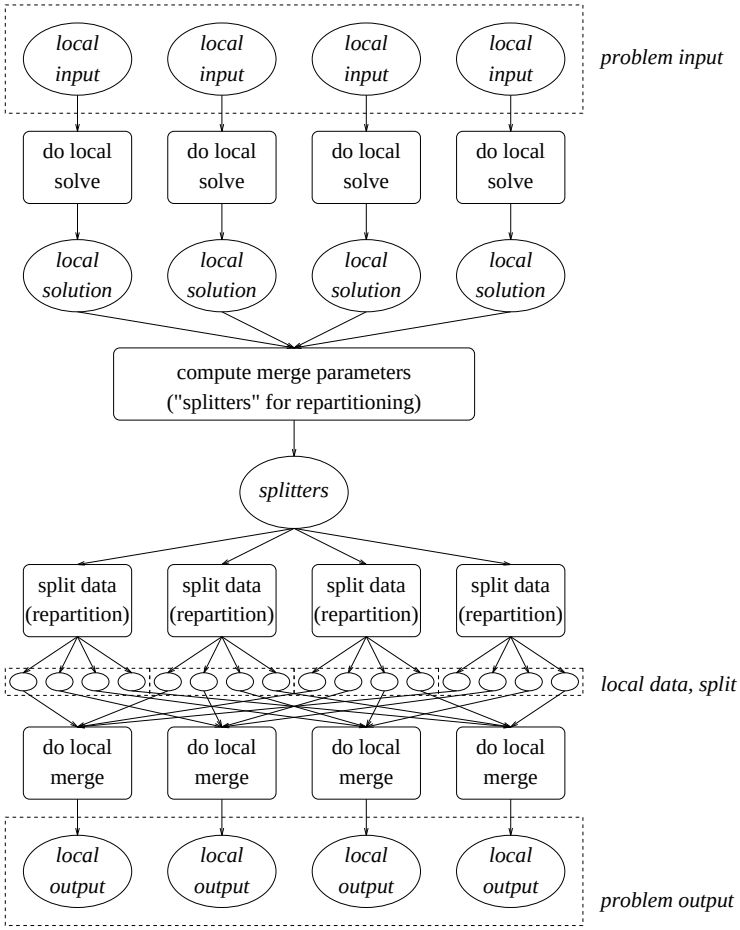

Figure 3: Dataflow in one-deep divide and conquer with degenerate split and concatenation merge.

or provide detailed guidelines for this step, but the transformation is aided to some extent by consideration of the overall archetype pattern, as is illustrated by the examples in the remainder of this section.

2.5 Application example: Mergesort

This section presents in some detail the development of a one-deep version of the mergesort algorithm based on the general one-deep divide-and-conquer archetype and its transformation into a program suitable for execution on a distributed-memory message-passing computer.

2.5.1 Problem description

The problem addressed by the mergesort algorithm is to sort an array into ascending order. In sequential mergesort, the split phase of the algorithm simply splits the array into left and right halves, the base-case solve trivially sorts an array of a single element, and the merge phase of the algorithm merges two sorted arrays.

2.5.2 Archetype-based algorithm, version 1

The one-deep version of mergesort proceeds as follows:

- The split phase is degenerate; the initial distribution of data among processes is taken to be the split.
- The local solve phase consists of sorting the data in each process locally, using any efficient sequential algorithm.
- The merge phase proceeds as follows:
  1. Using information about some (or all) local lists, compute $N-1$ splitters $s_1, s_2, \ldots, s_{N-1}$, where $s_i \leq s_{i+1}$. (These are the "parameters" of the merge phase. There are several approaches to computing these points; we do not give details.)
  2. Use these splitters to split the local sorted lists into N sorted sublists, such that elements with values at most s_i belong to the i-th list.
  3. Redistribute these sublists in such manner that sublists with elements with values at most s_i from *all* processes are located at process i.
  4. In each process, merge these sorted sublists.

After the algorithm terminates, process i has a sorted list whose elements are larger than the elements of process $i-1$'s list but smaller than the elements of process $i+1$'s list.

It is then straightforward to write down this algorithm; figure 4 shows C-like pseudocode, using the CC++ [11] **parfor** construct to express exploitable

concurrency. Observe that the iterations of each **parfor** loop are independent (this is part of the computational pattern captured by the archetype), so this algorithm can be executed (and debugged, if necessary) sequentially by replacing the **parfor** loops with **for** loops. Observe also that this algorithm could be executed without change and with the same results on an architecture that supports the **parfor** construct.

```
void
mergesort(int DataSize, int N,
        Distr_array Data[N][DataSize/N])
{
        Local_splitters LS[N] ;
        Global_splitters GS ;
        Distr_arrays SplitData[N][N][DataSize] ;

        /* ---- solve phase ---- */
        parfor (i=0 ; i<N ; i++)
                local_sort(Data[i]) ;
        /* ---- merge phase ---- */
        parfor (i=0 ; i<N ; i++)
                compute_local_splits(Data[i], LS[i]) ;
        compute_splits(LS, GS) ;
        parfor (i=0 ; i<N ; i++)
                local_repartition(GS, Data[i], SplitData[i]) ;
        parfor (i=0 ; i<N ; i++)
                local_merge(SplitData, Data[i]) ;
}
```

Figure 4: Mergesort, version 1.

2.5.3 Archetype-based algorithm, version 2

Guided by the archetype (i.e., by the dataflow pattern of figure 3), we can then rewrite the algorithm of figure 4 in a form more suitable for a distributed-memory message-passing architecture. For such architectures, the archetype can be expressed as an SPMD (single-process, multiple-data) computation with `N` processes, each corresponding to one of the elements of the **parfor** loops in the initial version. The archetype's dataflow pattern indicates how this conversion is to be done. The archetype supplies any code skeleton needed to create and connect the `N` processes; figure 5 shows pseudocode for one process. The transformation between the two algorithm versions, guided by the archetype, is straightforward and when performed according to archetype-based guidelines preserves program semantics.

```
void
mergesort_process(int DataSize, int N,
        Distr_array_section Data[DataSize])
{
        Local_splitters LS[N] ;
        Global_splitters GS ;
        Distr_array_sections SplitData[N][DataSize] ;

        /* ---- solve phase ---- */
        local_sort(Data) ;
        /* ---- merge phase ---- */
        compute_local_splits(Data, LS[i]) ;
        broadcast(LS[i]) ;
            /* gather the LS[j]s broadcast by other processes */
        gather(LS) ;
        compute_splits(LS, GS) ;
        local_repartition(GS, Data, SplitData) ;
            /* exchange SplitData[j] with process j */
        redistribute(SplitData) ;
        local_merge(SplitData, Data) ;
}
```

Figure 5: Mergesort, version 2.

2.5.4 Implementation

Transformation of the algorithm shown in figure 5 into code in a sequential language plus message-passing is straightforward based on the dataflow diagram of figure 3, with most of the details encapsulated in the broadcast, gather, and redistribution routines. This algorithm and a traditional parallel mergesort algorithm have been implemented in C with NX and executed on the Intel Delta. Figure 6 shows the speedups (compared to sequential mergesort) obtained using the two algorithms. In the figure "perfect" speedup represents the best speedup obtainable without superlinear effects and is simply the number of processors. As anticipated, the one-deep version performs significantly better. An analogous comparison for a shared-memory architecture can be found in [36].

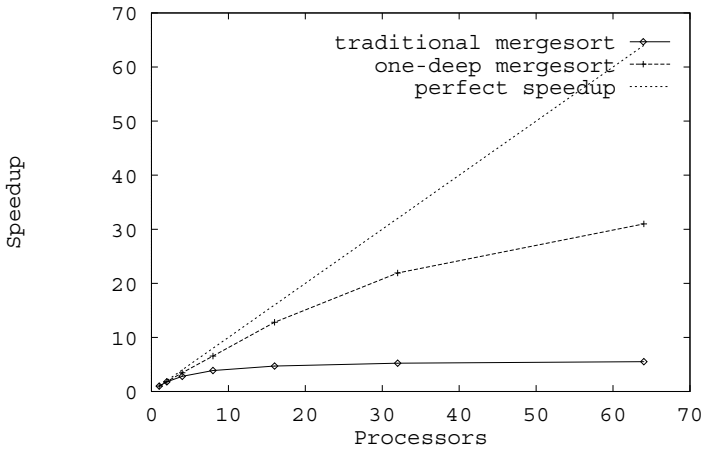

Figure 6: Speedups of traditional and one-deep mergesort compared to sequential mergesort, for 1,048,576 (2^{20}) integers, on the Intel Delta.

2.6 Other application examples

The above discussion of mergesort illustrates that the key difficulty in applying the one-deep archetype to a divide-and-conquer problem is algorithmic — determining an effective N-way split and/or merge. This section therefore presents additional examples — the skyline problem and quicksort — focusing on this algorithmic aspect. Other problems amenable to one-deep solutions include the convex hull problem and the problem of finding the two nearest neighbors in a set of points in a plane.

2.6.1 The skyline problem

The skyline problem, as described in [30], consists of merging a collection of rectangularly-shaped buildings into a single skyline. In the sequential divide-and-conquer algorithm, the base-case solution takes a single building and returns it as a skyline, and the merge operation merges two skylines into one by considering their overlap. A one-deep version of this algorithm is similar to the one-deep mergesort algorithm in §2.5; it proceeds as follows:

- The split phase is degenerate; the initial distribution of data among N processes (each process has part of the collection of buildings) is taken to be the split.
- The local solve phase consists of finding the skyline for each local collection, using the sequential algorithm.
- The merge phase proceeds as follows:
    1. Sample the data locally to find the distribution of points within the local skylines (in particular, find the left-most and the right-most points of each local skyline).
    2. Using these sample points, compute "splitters", which are the locations of vertical lines that cut all local skylines into N regions (which possibly have approximately equal number of points).
    3. Use these splitters to split each skyline into N adjacent buildings, each located between two splitters.
    4. Redistribute these buildings between the processes, so that each process receives all buildings within some region between two splitters.
    5. In each process, combine the buildings using the merge algorithm from the sequential algorithm.

The concatenation of the local skylines is then the final skyline.

2.6.2 Quicksort

The problem addressed by the quicksort algorithm is the same as that addressed by mergesort, namely to sort an array into ascending order. In sequential quicksort, the split phase of the algorithm splits the array into two halves based on the value of a pivot element, with smaller elements in one half and larger elements in the other. The base-case solve trivially sorts an array of a single element, and the merge phase simply concatenates the two subsolutions with the pivot element between them. The one-deep version of quicksort, unlike the one-deep versions of mergesort and the skyline algorithm, has a nontrivial split phase and a degenerate merge phase. It proceeds as follows:

- The split phase selects N-1 pivot elements (where N is the number of processors) $p_1, \ldots, p_{N-1}$ and partitions data into segments $P_1, \ldots, P_N$ such that data in segment P_i is between p_i and p_{i+1}.
- The local solve phase consists of sorting the data in each process locally, using any efficient sequential algorithm.
- The merge phase is degenerate.

After the algorithm terminates, process i has a sorted list whose elements are larger than the elements of process $i-1$'s list but smaller than the elements of process $i+1$'s list, so the final sorted list is the concatenation of the local lists.

3 Example: The mesh-spectral archetype

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 traditional 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 (where 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 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).

[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.

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.[2]

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.

[2] This paper addresses only the question of which data distributions are compatible with the problem's computational structure. Within these constraints, programmers may choose any data distribution; choosing the data distribution that gives the best performance is important but orthogonal to the concerns of this paper. However, an archetype-based performance model, such as that described in [32], may help with this choice.

Observe that after completion of a reduction operation all processes have access to its result; this must be guaranteed by the implementation.

File input/output operations. Exploitable concurrency and appropriate data distribution depend on considerations of file structure and (perhaps) system-dependent I/O considerations. One possibility 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 possibility 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 [19, 28], or it could be expressed in terms of more formal transformations with proofs of their correctness (currently in work).

3.3 Communication patterns

The above dataflow patterns give rise to the need for the following communication operations:

Grid redistribution. If different parts of the computation require different distributions — for example, if a row operation is followed by a column operation — data must be redistributed among processes, as in figure 7.

Exchange of boundary values. If a grid operation uses value 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, as shown in figure 8.

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

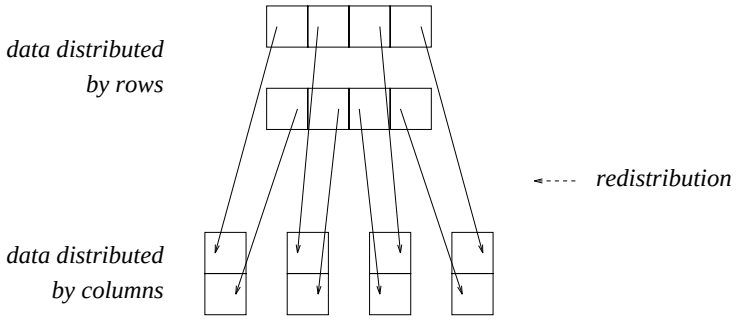

Figure 7: Redistribution: rows to columns.

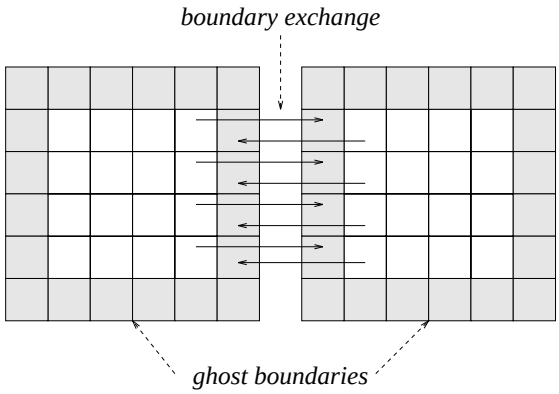

Figure 8: Boundary exchange.

Support for reduction operations. Reduction operations can be supported by several communication patterns depending on their implementation, e.g., all-to-one/one-to-all or recursive doubling. Figure 9 shows recursive doubling used to compute the sum of the elements of an array.

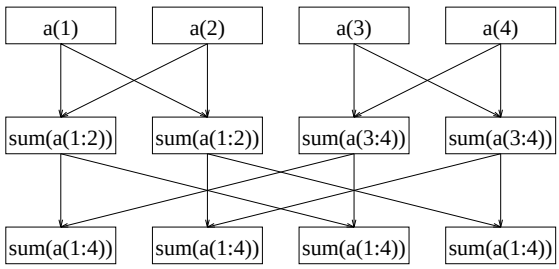

Figure 9: Recursive doubling to compute a reduction (sum).

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 postconditions (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

In contrast to the one-deep divide-and-conquer algorithms examined in §2, where the key difficulties were algorithmic, 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 examples in §3.5 and §3.6 illustrate how this archetype can be used to develop algorithms and transform them into versions suitable for execution on a distributed-memory message-passing architecture. In addition, §3.7 briefly describes real-world applications based on this archetype.

3.5 Application example: Two-dimensional FFT

We first present a simple example making use of row and column operations and data redistribution. This example illustrates how the archetype guides the process of transforming a sequential algorithm into a program for a distributed-memory message-passing architecture.

3.5.1 Problem description

The problem is to perform a two-dimensional discrete Fourier transform (using the FFT algorithm) in place. This can be done (as described in [31]) by performing a one-dimensional FFT on each row of the two-dimensional array and then performing a one-dimensional FFT on each column of the resulting two-dimensional array.

3.5.2 Archetype-based algorithm, version 1

It is clear that the sequential algorithm described fits the pattern of the mesh-spectral archetype: The data (the two-dimensional array) is a grid, and the computation consists of a row operation followed by a column operation. Thus, it is easy to write down an archetype-based version of the algorithm. Figure 10 shows HPF[25]-like pseudocode for this version. 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 a **do** loop. Observe also that this algorithm could be executed without change and with the same results on an architecture that supports the **forall** construct.

3.5.3 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. For such an architecture, the archetype can be expressed as an SPMD computation with `P` processes, with the archetype supplying any code skeleton needed to create and connect the `P` processes. Guided by the archetype (i.e., by the discussion of dataflow and communication patterns above), we can transform the algorithm of figure 10 into an SPMD computation in which each process executes the pseudocode shown in figure 11: Since the precondition of the row operation is that the data be distributed by rows, and the precondition

```
        subroutine twoDfft(N, M, Data)
        integer, intent(in) :: N, M
        complex :: Data(N, M)
!-------do row FFTs
        !HPF$ INDEPENDENT
        forall (i = 1:N)
                call rowfft(Data(i,:))
        end forall
!-------do column FFTs
        !HPF$ INDEPENDENT
        forall (j = 1:M)
                call colfft(Data(:,j))
        end forall
        end subroutine twoDfft
```

Figure 10: Two-dimensional FFT, version 1.

of the column operation is that the data be distributed by columns, we must insert between these two operations a data redistribution. For the sake of tidiness, we add an additional data redistribution after the column operation to restore the initial data distribution. Observe that most of the details of interprocess communication are encapsulated in the redistribution operation, 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.

3.5.4 Implementation

Transformation of the algorithm shown in figure 11 into code in a sequential language plus message-passing is straightforward, with most of the details encapsulated in the redistribution routine. This algorithm has been implemented on top of a general mesh-spectral archetype implementation (consisting of a code skeleton and an archetype-specific library of communication routines). The archetype in turn has been implemented in both Fortran M [23] and Fortran with MPI [29]. 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 12 shows speedups of the MPI version of the parallel code compared to the equivalent sequential code (produced by executing version 1 of the algorithm sequentially), executed on the IBM SP.

3.6 Application example: Poisson solver

We next present a less simple example making use of grid operations, a reduction operation, and the use of a global variable for control flow. This example

```
        subroutine twoDfft_process(N, M, P, Data_rows)
        integer, intent(in) :: N, M, P
        complex :: Data_rows(N/P, M)
        complex :: Data_cols(N, M/P)
!-------do row FFTs
        do i = 1, N/P
                call rowfft(Data_rows(i,:))
        end do
!-------redistribute
        call redistribute(Data_rows, Data_cols)
!-------do column FFTs
        do j = 1, M/P
                call colfft(Data_cols(:,j))
        end do
!-------redistribute to restore original distribution
        call redistribute(Data_cols, Data_rows)
        end subroutine twoDfft_process
```

Figure 11: Two-dimensional FFT, version 2.

again illustrates how the archetype guides the process of transforming a sequential algorithm into a program for a distributed-memory, message-passing architecture.

3.6.1 Problem description

The problem (as described in [37]) 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:

$$4u_{(i,j)}^{(k+1)} = h^2 f_{i,j} + u_{(i-1,j)}^{(k)} + u_{(i+1,j)}^{(k)} + u_{(i,j-1)}^{(k)} + u_{(i,j+1)}^{(k)}$$

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

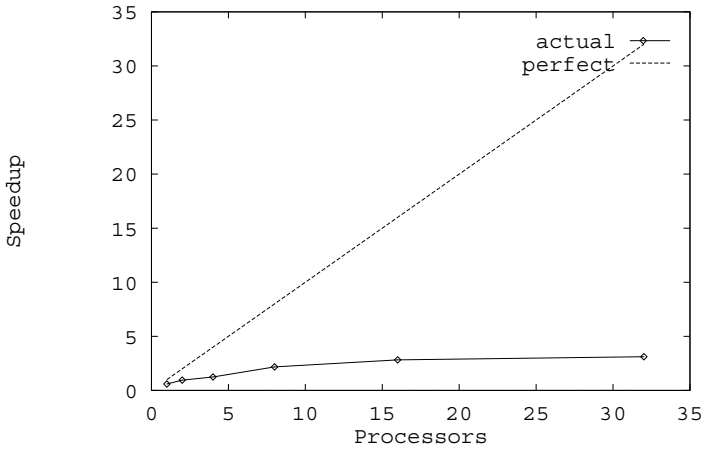

Figure 12: Speedup of parallel 2D FFT compared to sequential 2D FFT for 800 by 800 grid, FFT repeated 10 times, on the IBM SP. Disappointing performance is a result of too small a ratio of computation to communication. This parallelization of 2D FFT might nevertheless be sensible as part of a larger computation or for problems exceeding the memory requirements of a single processor.

`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 $|u_{(i,j)}^{(k+1)} - u_{(i,j)}^{(k)}|$ to check for convergence.

3.6.2 Archetype-based algorithm, version 1

It is fairly clear again 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. Figure 13 shows HPF-like pseudocode for this version, 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.

3.6.3 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. As with the two-dimensional FFT program, the overall computation is to be expressed as an SPMD computation, with 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 13 into an SPMD computation in which each process executes the pseudocode shown in figure 14: 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 com-

```
        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 13: Poisson solver, version 1.

putation of `ukp1` is preceded by a boundary exchange operation. The reduction operation is also transformed in the manner described previously; since a post-condition 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. As with the previous example, 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. Also as with the previous example, observe that most of the details of interprocess communication are encapsulated in the redistribution operation, 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.

3.6.4 Implementation

As in the previous example, transformation of the algorithm shown in figure 14 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 on top of the same general mesh-spectral archetype implementation mentioned in §3.5.4. Figure 15 shows speedups of the MPI version of the parallel code compared to the equivalent sequential code, executed on the IBM SP.

3.7 Other application implementations

As the preceding examples illustrate, 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 experiences as sketched above and in the following section suggest that even fairly naive implementations of the communication library can give acceptable performance. Performance can then be improved by tuning the library routines, with potential benefit for other archetype-based applications.

In this section we describe additional real-world applications we have developed based on the mesh-spectral archetype. For historical reasons, to date they have been implemented based on special-case versions of the archetype (a spectral archetype focusing on row and column operations, and a mesh archetype focusing on grid operations), but they could equally well be implemented using the general archetype.

```
        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 14: Poisson solver, version 2.

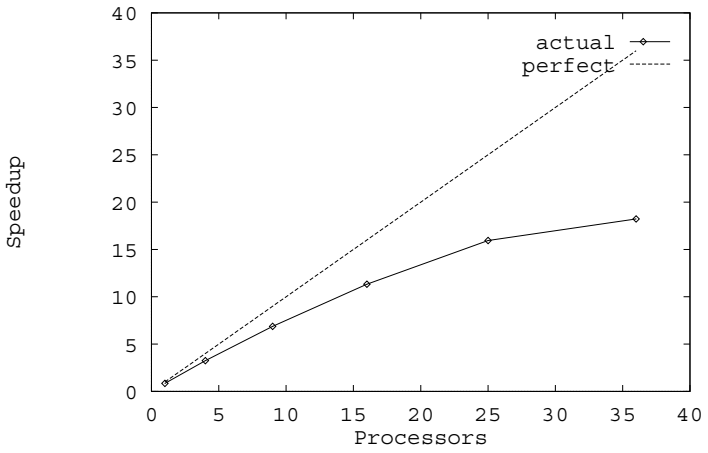

Figure 15: Speedup of parallel Poisson solver compared to sequential Poisson solver for 800 by 800 grid, 1000 steps, on the IBM SP.

3.7.1 Compressible flow

Two similar computational fluid dynamics codes have been developed using archetypes. These two codes simulate high Mach number compressible flow; both are based on the two-dimensional mesh archetype and have been implemented in Fortran with NX for the Intel Delta and the Intel Paragon. Figure 16 shows speedups for the first code; the speedups are for the Intel Delta and are relative to single-processor execution of the parallel code. Figures 19 and 20 show sample output for the two codes. The second version of the code is notable for the fact that it was developed by an "end user" (applied mathematician) using the mesh archetype implementation and documentation, with minimal assistance from the archetype developers.

3.7.2 Electromagnetic scattering

This code performs numerical simulation of electromagnetic scattering, radiation, and coupling problems using a finite difference time domain technique. It is based on the three-dimensional mesh archetype and has been implemented in Fortran M for networks of Sun workstations and the IBM SP. Figure 17 shows speedups of the parallel code compared to the equivalent sequential code, both executed on the IBM SP. This application is notable in that the parallel version of the code was developed from an existing sequential program by applying

a sequence of archetype-guided transformations to produce first a sequential simulation of the parallel code and then the actual parallel code. During development of this application, the transformations used to convert the original sequential program into the sequential simulated-parallel version had not been formally stated and proved, so the correctness of these steps was established by testing and debugging — which was done in the sequential domain with familiar tools and techniques. For the final transformation (from sequential simulated-parallel code to actual parallel code for programs based on this archetype), a formal proof of correctness was developed. Thus, the final parallel version needed no debugging; it ran correctly on the first execution.

3.7.3 Incompressible flow

This spectral code provides a numerical solution of the three-dimensional Euler equations for incompressible flow with axisymmetry. Periodicity is assumed in the axial direction; the numerical scheme uses a Fourier spectral method in the periodic direction and a fourth-order finite difference method in the radial direction. It is based on the two-dimensional spectral archetype and has been implemented in Fortran M for networks of Sun workstations and the IBM SP. Figure 18 shows speedups for the parallel code relative to single-processor execution on the IBM SP. Figure 21 shows sample output.

3.7.4 Smog model

This code, known as the CIT airshed model [15, 16, 17] models smog in the Los Angeles basin. It is conceptually based on the mesh-spectral archetype, although it does not use the mesh-spectral implementation, and has been implemented on a number of platforms, including the Intel Delta, the Intel Paragon, the Cray T3D, and the IBM SP2, as described in [15].

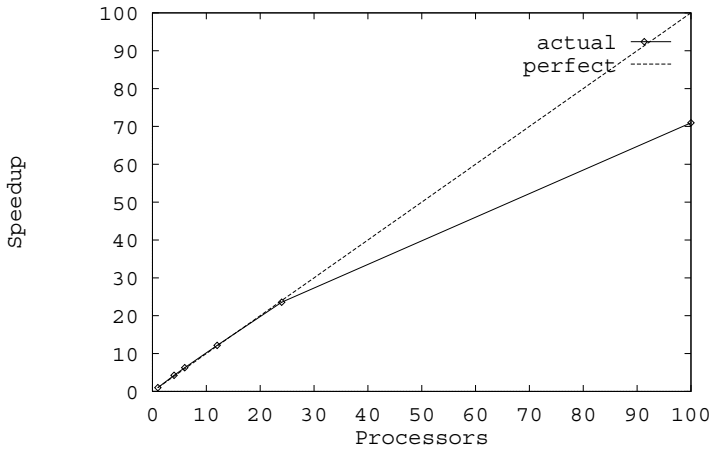

Figure 16: Speedup of 2D CFD code compared to single-processor execution for 150 by 100 grid, 600 steps, on the Intel Delta.

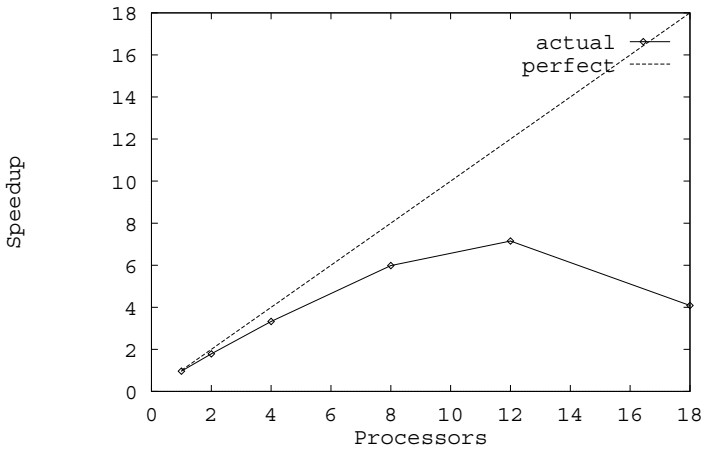

Figure 17: Speedup of parallel electromagnetics code compared to sequential code for 66 by 66 by 66 grid, 512 steps, on the IBM SP. The decrease in performance for more than 12 processors results from the ratio of computation to communication dropping too low for efficiency.

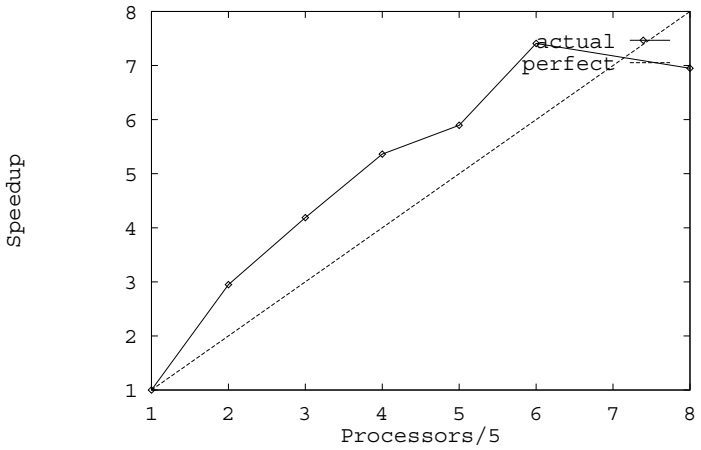

Figure 18: Speedup of spectral code compared to 5-processor execution for 1536 by 1024 grid, 20 steps, on the IBM SP. Because single-processor execution was not feasible due to memory requirements, a minimum of 5 processors was used, and so speedups are calculated relative to a base of execution on 5 processors. Inefficiencies in executing the code on the base number of processors (e.g., paging) probably explain the better-than-ideal speedup for small numbers of processors.

Figure 19: Output of CFD code 1: Density as a shock interacts with a sinusoidal density gradient.

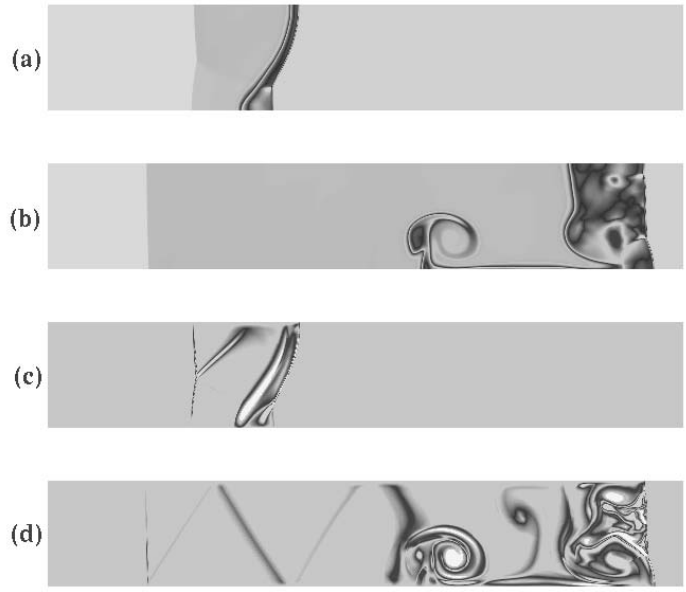

Figure 20: Output of CFD code 2: Density (a,b) and vorticity (c,d) images for a Mach 10 shock interaction with a sinusoidal N2-H2 interface, with IDG (ideal dissociating gas) chemistry at late and early times.

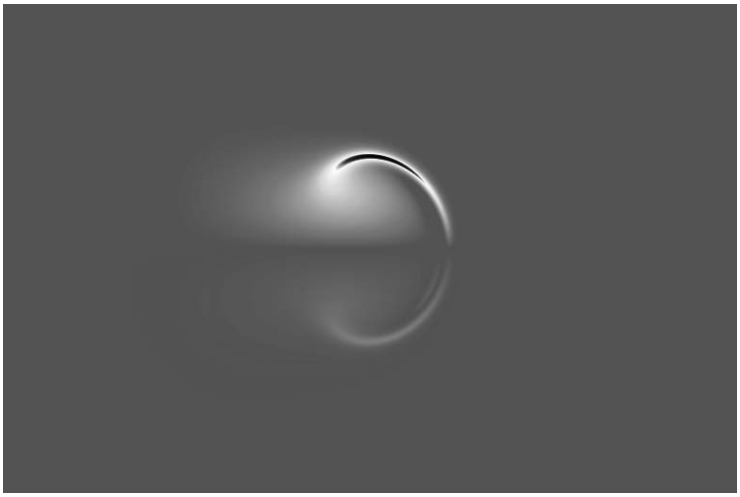

Figure 21: Output of spectral code: Azimuthal velocity in a swirling flow.

4 Related work

Design patterns. Many researchers have investigated the use of patterns in developing algorithms and applications. Previous work by the authors and others [10, 12] explores a more general notion of archetypes and their role in developing both sequential and parallel programs.

Gamma et al. [24] address primarily the issue of patterns of computation, in the context of object-oriented design. Our notion of a parallel program archetype, in contrast, includes patterns of dataflow and communication.

Schmidt [33] focuses more on parallel structure, but in a different context from our work and with less emphasis on code reuse.

Shaw [34] examines higher-level patterns in the context of software architectures.

Brinch Hansen's work on parallel structures [8] is similar in motivation to our work, but his model programs are generally more narrowly defined than our archetypes.

Other work addresses lower-level patterns, as for example the use of templates to develop algorithms for linear algebra in [4].

Program skeletons. Much work has also been done on structuring programs by means of *program skeletons*, including that of Cole [14], Botorog and Kuchen [6, 7], and Darlington et al. [18]. This work is more oriented toward functional programming than ours, although [14] mentions the possibility of expressing the idea of program skeletons in imperative languages, and [7] combines functional skeletons with sequential imperative code.

This work, like that of Brinch Hansen, describes a program development strategy that consists of filling in the "blanks" of a parallel structure with sequential code. Our approach is similar, but we allow the sequential code to reference the containing parallel structure, as in the mesh-spectral archetype examples.

Program development strategies. Fang [22] describes a programming strategy similar to ours, but with less focus on the identification and exploitation of patterns.

The Basel approach [9] is more concerned with developing and exploiting a general approach for classifying and dealing with parallel programs.

Ballance et al. [3] are more explicitly concerned with the development of tools for application support; while our work can be exploited to create such tools, it is not our primary focus.

Kumaran and Quinn [27] focus more on automated conversion of template-based applications into efficient programs for different architectures.

Software reuse. Previous work on software reuse, e.g., Krueger [26] and Volpano and Kieburtz [38], tends to focus on code reuse, while our approach includes reuse of designs as well as code.

Dataflow patterns. Other work, e.g., Dinucci and Babb [21], has addressed the question of structuring parallel programs in terms of dataflow; our work differs in that it addresses patterns of both dataflow and computation.

One-deep divide and conquer algorithms. Several researchers have developed algorithms that fit the one-deep divide-and-conquer pattern. A treatment of the overall pattern, focusing on shared-memory architectures, appears in [36]. Examples of specific algorithms include one-deep mergesort [20] and one-deep quicksort [13, 35].

Distributed objects. The mesh-spectral archetype is based to some extent on the idea of distributed objects, as discussed for example in work on pC++ [5] and POOMA [1]. We differ from this work in that we focus more on the pattern of computation and on identifying and exploiting patterns of computation and communication.

Communication libraries. Many researchers have investigated and developed reusable general libraries of communication routines; MPI [29] is a notable example. Others have developed more specialized libraries, for example MPI-RGL [2] for regular grids. We differ from this work, again, in that our focus is on identifying and exploiting patterns.

Automatic parallelizing compilers. Much effort has gone into development of compilers that automatically recognize potential concurrency and emit parallel code; HPF [25] is a notable example. This work is orthogonal and, we hope, complementary to ours; we focus on giving application developers tools and techniques to manage explicit parallelism, for situations in which explicit parallelism allows for improved efficiency or more predictable performance.

5 Conclusions and future work

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 and/or proved-correct transformations for converting essentially sequential code into code for realistic parallel architectures, including message-passing architectures.

- An archetype implementation (in the form of a code skeleton and/or 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.

The primary weaknesses of our approach are these:

- Application developers must still identify which archetype to use for a particular problem, which may not be straightforward.
- Application developers must in some cases apply a certain amount of ingenuity to make the application (algorithm) and the archetype fit.

These weaknesses could to some extent be alleviated by the existence of a varied library of archetypes, each with many examples illustrating the range of its use.

Many directions for future work suggest themselves:

- As suggested above, developing an extensive library of archetypes would broaden the range of applications that can be developed using our strategy; collecting a range of examples for each archetype would assist developers in choosing and applying an appropriate archetype.
- Developing a theory and strategy for archetype composition would add the benefits of modular design to our approach and allow us to more easily address applications with compositional structures, for example task-parallel compositions of data-parallel computations.
- Our work to date has been targeted primarily toward developing programs for distributed-memory message-passing architectures. We believe that the archetypes approach is also applicable to shared-memory architectures, both symmetric and nonsymmetric; experiment is needed to determine whether this is the case.
- Our work to date has dealt mainly with deterministic archetypes, which are particularly useful in that they allow the initial stages of algorithm development to be completely tested and debugged sequentially. However, some problems are better suited to nondeterministic archetypes — for example, branch and bound — so our library of archetypes should include such archetypes as well.

- As can be inferred from the two examples presented in this paper, archetypes lend themselves to implementation in an object-oriented framework; this could be the basis for more structured versions of our archetype implementations (code skeletons and libraries).
- Finally, since they focus attention on restricted classes of problems rather than requiring full generality, archetypes provide a basis for class-specific tools for transforming programs, from formally-justified techniques to automated support. Developing such tools (which we believe to be easier than developing fully general tools) would further demonstrate the practical value of our approach.

Acknowledgments

The authors wish to thank:

- Svetlana Kryukova, Adam Rifkin, Paul Sivilotti, John Thornley, and other members of the Compositional Systems research group at Caltech for their work on the one-deep divide-and-conquer archetype (particularly Svetlana's work on mergesort, which forms the basis for §2.5, including the implementation), and for their careful critique of this paper.
- Dan Meiron for his help in defining the mesh-spectral archetype and in supervising applications.
- John Beggs, Donald Dabdub, Greg Davis, Rajit Manohar, and Ravi Samtaney, for their help in developing the real-world applications described in §3.7.

References

[1] S. Atlas, S. Banerjee, J. C. Cummings, P. J. Hinker, M. Srikant, J. V. W. Reynders, and M. Tholburn. POOMA: A high performance distributed simulation environment for scientific applications. ⟨ `http://www.acl.lanl.gov/PoomaFramework/papers/SCPaper-95.html` ⟩, 1995.

[2] C. F. Baillie, O. Bröker, O. A. McBryan, and J. P. Wilson. MPI-RGL: a regular grid library for MPI. ⟨ `http://www.cs.colorado.edu/ broker/mpi_rgl/mpi_rgl.ps` ⟩, 1995.

[3] R. A. Ballance, A. J. Giancola, G. F. Luger, and T. J. Ross. A framework-based environment for object-oriented scientific codes. *Scientific Programming*, 2(4):111–121, 1993.

[4] 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.

[5] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. X. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3):7–22, 1993.

[6] G. H. Botorog and H. Kuchen. Efficient parallel programming with algorithmic skeletons. In L. Bouge, editor, *Proceedings of EuroPar '96*, volume 1123–1124 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[7] 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.

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

[9] H. Burkhart, R. Frank, and G. Hächler. Structured parallel programming: How informatics can help overcome the software dilemma. *Scientific Programming*, 5(1):33–45, 1996.

[10] K. M. Chandy. Concurrent program archetypes. In *Proceedings of the Scalable Parallel Library Conference*, 1994.

[11] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object-oriented programming notation. In *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[12] 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.

[13] M. J. Clement and M. J. Quinn. Overlapping computations, communications and I/O in parallel sorting. *Journal of Parallel and Distributed Computing*, 28(2):162–172, August 1995.

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

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

[16] D. Dabdub and J. H. Seinfeld. Air quality modeling on massively parallel computers. *Atmospheric Environment*, 28(9):1679–1687, 1994.

[17] D. Dabdub and J. H. Seinfeld. Parallel computation in atmospheric chemical modeling. *Parallel Computing*, 22:111–130, 1996.

[18] 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.

[19] G. Davis and B. Massingill. The mesh-spectral archetype. Technical Report CS-TR-96-26, California Institute of Technology, 1996. Also available via ⟨ `http://www.etext.caltech.edu/Implementations/` ⟩.

[20] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *International Conference of Parallel and Distributed Information Systems*, pages 280–291, December 1991.

[21] 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.

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

[23] 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.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[25] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. *Scientific Programming*, 2(1–2):1–170, 1993.

[26] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–273, 1992.

[27] S. Kumaran and M. J. Quinn. An architecture-adaptable problem solving environment for scientific computing, 1996. Submitted to *Journal of Parallel and Distributed Computing*.

[28] B. Massingill. The mesh archetype. Technical Report CS-TR-96-25, California Institute of Technology, 1996. Also available via ⟨ `http://www.etext.caltech.edu/Implementations/` ⟩.

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

[30] M. E. Moret and H. D. Shapiro. *Algorithms from P to NP — Volume I: Design and Efficiency.* The Benjamin/Cummings Publishing Company, 1991.

[31] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes.* Cambridge University Press, 1986.

[32] A. Rifkin and B. L. Massingill. Performance analysis for mesh and mesh-spectral archetype applications. Technical Report CS-TR-96-27, California Institute of Technology, 1996.

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

[34] M. Shaw. Patterns for software architectures. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*, pages 453–462. Addison-Wesley, 1995.

[35] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, April 1992.

[36] J. Thornley. Performance of a class of highly-parallel divide-and-conquer algorithms. Technical Report CS-TR-95-10, California Institute of Technology, 1995.

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

[38] D. M. Volpano and R. B. Kieburtz. The templates approach to software reuse. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, chapter 9, pages 247–255. ACM Press, Addison Wesley, 1989.