Towards Architecture-Adaptable Parallel Programming

SANTHOSH KUMARAN[1] AND MICHAEL J. QUINN[2]

[1]*IBM, 3200 Windy Hill Road, Atlanta, GA 80339; e-mail: kumaras@research.cs.orst.edu*
[2]*Department of Computer Science, Oregon State University, Corvallis, OR 97331*

ABSTRACT

Parallel processing is facing a software crisis. The primary reasons for this crisis are the short life span and small installation base of parallel architectures. In this article, we propose a solution to this problem in the form of an architecture-adaptable programming environment. Our method is different from high-level procedural programming languages in two ways: (1) our system automatically selects the appropriate parallel algorithm to solve the given problem efficiently on the specified architecture; (2) by using a divide-and-conquer template as the basic mechanism for achieving parallelism, we considerably simplify the implementation of the system on a new platform. There is a trade-off, however: the loss of generality. From a pragmatic point of view, this is not a major liability since our strategy will be useful in building domain-specific problem solving environments and application-oriented compilers, which can be easily and effectively ported to diverse architectures. We give preliminary results from a case study in which our method is used to adapt the parallel implementations of the conjugate gradient algorithm on a multiprocessor, a multicomputer, and a workstation network.

1 INTRODUCTION

The most efficient parallel algorithm for solving a problem often depends on the target architecture. Thus, unless a parallel programming system has the ability to adapt the algorithm to the architecture, it will not be truly machine independent.

In the traditional approaches to machine-independent parallel programming, the user encodes an algorithm as a parallel program using a high-level programming language. Using a combination of compilers and run-time systems, this program can be executed on a variety of platforms, but the algorithm embedded in the program may not execute efficiently on all the platforms. Hence, only limited machine independence is achieved.

In this article, we present a new scheme for machine-independent parallel programming. Our scheme is built on the following three key ideas: (1) the use of a database of parameterized algorithmic templates to represent *computable* functions; (2) frame-based representation of processing environments; and (3) the use of an analytical performance prediction tool for automatic algorithm design.

By automating the detailed design of an algorithm and the generation of a parallel program, our approach relieves the user from much of the burden of parallel programming. There is a trade-off, however: the set of problems that can be solved efficiently using our approach is limited by the contents of the template database. However, we believe our strategy will be useful in

Received September 1995
Revised March 1996

Scientific Programming, Vol. 6, pp. 163–186 (1997)
CCC 1058-9244/97/020163-24

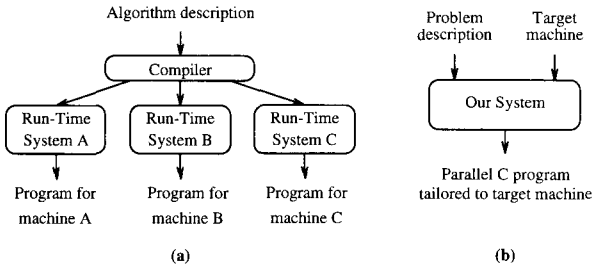

FIGURE 1 (a) Traditional algorithm-oriented approach to parallel processing. (b) Our problem-oriented approach to parallel processing.

building domain-specific *problem-solving environments* and *application-oriented compilers*, which can be easily and *effectively* ported to diverse architectures.

Figure 1 contrasts our approach with the traditional approach. In our problem-oriented approach, the user describes the problem to be solved, rather than an algorithm to solve the problem. The set of problems that can be solved using a system may be called the *scope* of the system. We restrict the scope of our system to provide a portable, easy to use, and high-performance processing environment. In contrast, the traditional approach maximizes the scope to include all *Turing computable* problems at the expense of restricting portability, programmability, and performance.

To see that limited scope is not a major liability, one only need to look at the recent history of the computer industry.

1. The massive surge in the popularity of personal computers is primarily due to the availability of domain-specific software packages with restricted scope, prime examples being word processors and spreadsheets.
2. In the realm of scientific computing, users are increasingly moving towards problem-solving environments such as MATLAB, abandoning the traditional programming languages, such as Fortran.
3. The biggest challenge to parallel computing comes from the "killer workstations," simply because the improvement in performance resulting from using the parallel computer is not enough to justify the additional cost and effort. Pragmatically, this implies that it is fruitless to parallelize *all* applications. Those that benefit from parallelization form a subset, and programming models with enough expressive power to cover a reasonable number of applications will have just as much practical use as a Turing equivalent model.

We use a computational model based on divide-and-conquer to design the algorithm templates. In the next few sections, we describe this model and the details of our scheme to automatically generate architecture-adaptable parallel programs. We have applied our scheme to develop efficient parallel programs for several scientific applications on diverse architectures. Included in this article is a case study describing the application of our strategy to parallelize the conjugate gradient (CG) method on a shared memory multiprocessor, a distributed memory multicomputer, and a network of workstations. We believe the diversity of the target platforms and the complexity of the application make this case study a good test of the validity of our approach.

2 COMPUTATIONAL MODEL

There is only one basic mechanism for parallelism in our model: a meta-function called *parallel divide-and-conquer* (PDC). Divide-and-conquer is a well-known problem-solving strategy in which a problem is solved by dividing it into a number of smaller subproblems and then solving the subproblems by the recursive application of the same procedure. Infinite recursion is prevented by using a *base predicate* which triggers a *base function*. The solutions to the subproblems form partial results, which are combined to form the final result. In PDC, the subproblems are solved in parallel, providing an easy opportunity for exploiting parallelism in architecture.

In our model, a program is represented as a sequence of divide-and-conquer. Figure 2a shows the graphical representation of such a program in the form of a DAG, comprising three divide-and-conquer operations. The shaded squares denote the base cases. Note that the number of subprograms generated and the depth of recursion change for each invocation of the operation. Essentially, each operation has a well-defined top-level structure, but the details can change for each invocation. We use the notion of a *parameterized template* to represent these operations; the template describes the top-level structure and the parameters are used to add the details. The lowest layer of our template database is made up of such templates. Meta-templates, consisting of cascaded divide-and-conquer operations such as the one shown in Figure 2a, are formed from these *base* templates.

Another important aspect of our computational model is the mapping of the subproblems to the processing nodes. We combine the divide-and-conquer paradigm with the single-program multiple-data (SPMD) style of programming to obtain an efficient

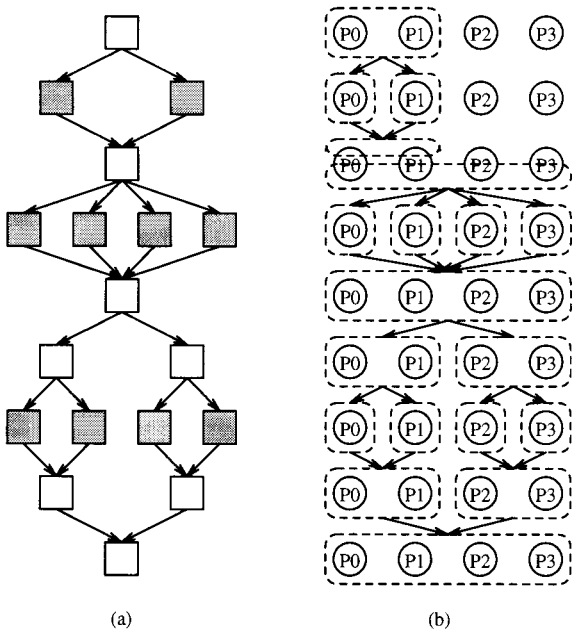

FIGURE 2 Mapping of the divide-and-conquer tree to the processing nodes. Task graph shown on the left is mapped as shown on the right. The shaded squares denote the base case. Note that the first divide-and-conquer operation is performed using only two processors, whereas the rest of the computation uses four processors.

implementation of the cascaded divide-and-conquer explained above. The subproblems at each level of the DAG are mapped to all or a subset of the processors. This is in contrast to the conventional approach to divide-and-conquer programming where each subproblem gets mapped to a single processor. Figure 2b shows a possible implementation of the program in Figure 2a using two processors for the first divide-and-conquer operation and four processors for the rest of the computation.

A meta-template is an abstract, high-level representation of a generic method to solve a problem. There may be a large number of plausible implementations for a meta-template. We generate an efficient program to solve a problem on a given architecture by choosing the implementation that performs best on that architecture. Thus, we see that there is a search space associated with each meta-template and the problem of generating an efficient program reduces to a search problem. The size of this search space is determined by the number of constituent base templates, the number of parameters in each one of them, and the number of permissible values for each of these parameters. Note that there could be several meta-templates to solve the same problem, adding one more dimension to the search space.

3 METHODOLOGY

We begin with a collection of meta-templates for the problem and an abstract description of the architecture. The templates represent *methods* for solving the problem. The number of templates in the collection is problem dependent—some problems will have only a single template, whereas others may have two or more. Our goal is to generate an efficient algorithm to solve the problem on the specified architecture.

To achieve this goal, we traverse the path from a generic method to an algorithm by adding the necessary details. This means customizing the template by determining the appropriate values for the parameters. If the search space is small, we can exhaustively search for the best set of values for the parameters, provided we have a good objective function. The role of the analytical performance prediction tool is to provide this objective function. Given a set of parameter values and the relevant specifications of a target platform, the tool predicts the performance of the implementation on the specific platform.

What kind of details do we need to add to the template to make it an efficient algorithm? Here is a partial list:

1. Structure of the divide-and-conquer tree: This will vary based on the processing environment for the sample template.
2. Mapping of the processing nodes to the leaves of the tree: The mapping that minimizes the communication overhead is desired.
3. Depth of the tree: This determines the granularity of the resulting parallel program.
4. Optimal subset mapping: Sometimes performance may be enhanced by using only a subset of the resources.
5. Machine-specific data decomposition: There are several ways grid data can be decomposed, and based on the problem instance and the architecture, a particular decomposition may be superior.
6. Machine-specific solution method: When there are several candidate templates, the one that maximizes the performance needs to be selected.

A combinatorial explosion of search space is conceivable for complex applications, making exhaustic search impractical. For these cases, there are two ways to prune the search tree:

1. We can make use of application-specific knowledge to limit the number of parameters and their permissible values. This kind of pruning is done

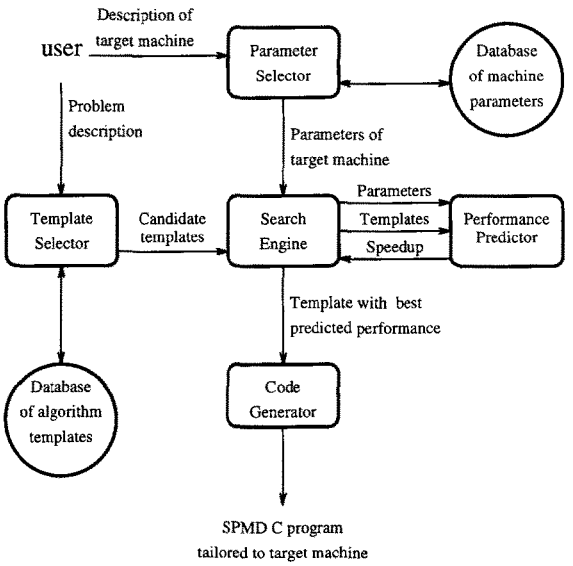

FIGURE 3 Schematic representation of our method for generating efficient parallel programs to solve a given problem on a specified architecture.

manually at the template design stage. We will show an example of this in the case study.

2. We can use branch-and-bound algorithms to eliminate the fruitless searching of unproductive branches. The system can automatically perform this pruning while searching for the best implementation.

If there is more than one template for a problem, then each one of them will be customized and the best one selected using the performance prediction tool. Converting the detailed template to a message-passing program or a shared memory program can be accomplished using current compiler technology [1]. Figure 3 shows the method schematically.

It is important to note that divide-and-conquer *is bing used in this system merely as a methodology for designing the templates. Users do not write divide-and-conquer functions—they call higher-level functions like matrix-vector multiply or dot product. Emitted code is not a divide-and-conquer program. It is an SPMD program in which every processor is active throughout the execution of the program and doing useful work:*

3.1 Divide-and-Conquer Template

Our template is based on the algebraic model of divide-and-conquer proposed by Mou and Hudak in [2]. The template encapsulates problem solving using divide-and-conquer in three phases: a divide phase, a conquer phase, and a combine phase. An overview of the template is given below.

- **Data distribution declarations:** This explains how data points are distributed among the processing units for distributed memory machines; for shared memory systems, this represents the logical division of data points among processing nodes. Using grid problems as an example, row decomposition, column decomposition, and block decomposition can all be captured using appropriate data distribution declarations. These declarations can also be thought of as *pre-conditions* and *post-conditions* on the template. The distribution of input data is a necessary pre-condition for the invocation of the template; the result of the invocation (post-condition) is the output data distributed in the specific layout.
- **Divide phase:** Actions in this phase can be expressed using two functions:
  1. Divide function: As shown in Figure 2, the SPMD implementation of the parallel divide-and-conquer requires a mapping from the subproblems to the processing nodes. The purpose of the divide function is to specify this mapping. For example, consider the first divide-and-conquer operation in Figure 2. The original problem is mapped to a pool of two processors. In the divide phase, two subproblems are generated. To map these subproblems to the processing nodes, we split the processing nodes into two partitions: a left partition and a right partition. The first subproblem is allocated to the left partition and the second problem to the right partition. This is an example of a frequently used simple divide function: binary, equal, one-dimensional, *left-right* division. Thus, the domain of the divide function is the set of processing nodes of the target environment.
  2. Pre-adjust function: Notice that the divide function implies the partitioning of the domain of the function computed by the template as well. For example, if the function is computing the product of a banded matrix and a vector, the left-right divide will cut the input vector and the matrix into two chunks as shown in Figure 4. In addition to partitioning data, the divide phase may need to modify the partitioned data sets. This is accomplished using a pre-adjust function which is applied to the partitions *before* the subproblems contained in these

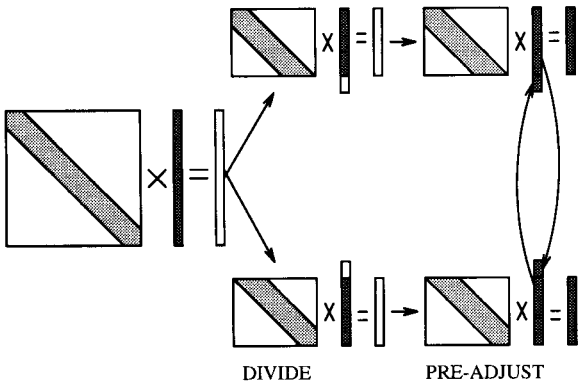

FIGURE 4 Divide phase of a template for banded-matrix-vector multiplication. Pre-adjust function is used to extend the top half of the vector towards the bottom and the bottom half towards the top. Actual multiplication occurs inside the base function.

partitions can be solved. The divide phase of the banded-matrix-vector multiplication template is illustrated schematically in Figure 4, where we show how the divide function splits the data sets and the pre-adjust function modifies them.

- **Conquer phase:** We need to specify only a sequential base function and a base predicate in this phase, because everything else reduces to recursive applications of the previously defined template. The base predicate is used to specify the terminating condition of the recursion. Because we use the divide function to partition the set of processing nodes, the recursion will have to terminate when there is only a single node in a partition. Thus, the default base predicate checks the number of nodes in the partition and returns true if there is only one. In this case, the base function is simply a sequential program.

  In some architectures, it might be advantageous to terminate the recursion early with a group of processing nodes in each partition, instead of single-node partitions. The base function will be a parallel program in such cases, but less complex than a program designed to run efficiently on the entire platform. An interesting special case is when the base functions are data-parallel programs, giving *nested data parallelism.*
- **Combine phase:** When the subproblems are solved, we will have partial results distributed among the processors. In the combine phase, we wish to combine them to produce the final answer. Similar to the divide phase, we use two functions to accomplish this:

  1. Post-adjust function: Subproblem solutions are modified using this function. We can use the power of recursion and invoke the template itself from within the post-adjust function, if necessary. The example template for matrix multiplication, given below, illustrates this. A simpler example will be the computation of the dot product of two vectors. In the combine phase, we use the post-adjust function to add the partial results to form the global sum.
  2. Combine function: The combine function is merely the inverse of the divide function. As an example, the *left-right combine* merges the left and right partitions into a single partition.

Notice that the divide and combine functions do not operate directly on the application data. These functions merely change the state of a set of registers — which we call system data — maintained by each processor. The system data collectively determine the *position* of a processor within the DAG representing the divide-and-conquer operation. For example, consider how the position of the processor **P1** changes during the last divide-and-conquer operation in Figure 2. The first application of the divide function changes the system data of **P1** to make it the second processor in the left partition. The second divide makes it the first processor in the right partition of the first left-right pair. During the combine phase, the first application of the combine function makes it once again the second processor in the left partition. The adjust functions, which operate directly on the application data, use the position information to determine the exact nature of communications and computations at each processor.

Figure 5a shows the execution of the generic PDC on n processors. Figure 5b shows the execution on a single processor, the base case.

Figure 6 shows an example template for matrix multiplication, $C = AB$. This is one of three templates we have for matrix multiplication in the system database.

Figure 7a shows the execution of the matrix multiplication template on n processors. The unrolled execution sequence for four processors is shown in Figure 7b. Figure 8 shows how the data structures at each processor change as execution proceeds.

Our current design of the system uses a higher-order function to implement the template with the arguments of this function representing the fields of the template. All communications appear exclusively in the adjust functions. Additionally, these templates

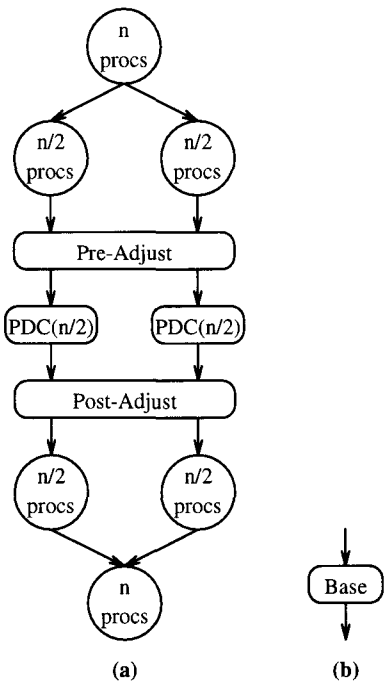

FIGURE 5 Schematic view of the execution of the generic PDC template. (a) PDC on n processors. (b) PDC on one processor: the base case.

have the benefit of having only regular and well-defined communication patterns.

3.2 Representation of the Processing Environment

The computing environment is described using a frame structure. The slots in the frame represent attributes, values of which may be represented by other frames. The collection of frames, thus formed, holds all the information we need to design a program that will execute efficiently on the represented environment. The information stored in the frame includes the number of processors, the processing power of the nodes, the interconnection network, and the memory hierarchy. Figure 9 shows the frame representation of a typical high-performance computing environment.

3.3 Performance Prediction

Performance prediction plays an important role in the development of a detailed algorithm from a generic template, as pointed out in Section 3. Our analytical performance prediction tool is built on the model developed by Clement and Quinn [3]. It exploits the algebraic structure of divide-and-conquer algorithms

Data distribution:

matrix A distributed row-wise among the processors
matrix B distributed column-wise among the processors
matrix C distributed row-wise among the processors

Divide function:

Divide the processor pool into two equal partitions, a LEFT partition and a RIGHT partition.
(In distributed-memory machines, this would automatically
imply the division of data structures as well.)

Pre-adjust function:

None.

Base function:

Sequential Matrix Multiplication.

Post-adjust function:

Swap columns of B between partitions.
Apply Matrix Multiplication Template to both partitions.

Combine function:

Combine the LEFT and RIGHT partitions.

FIGURE 6 An example template for matrix multiplication.

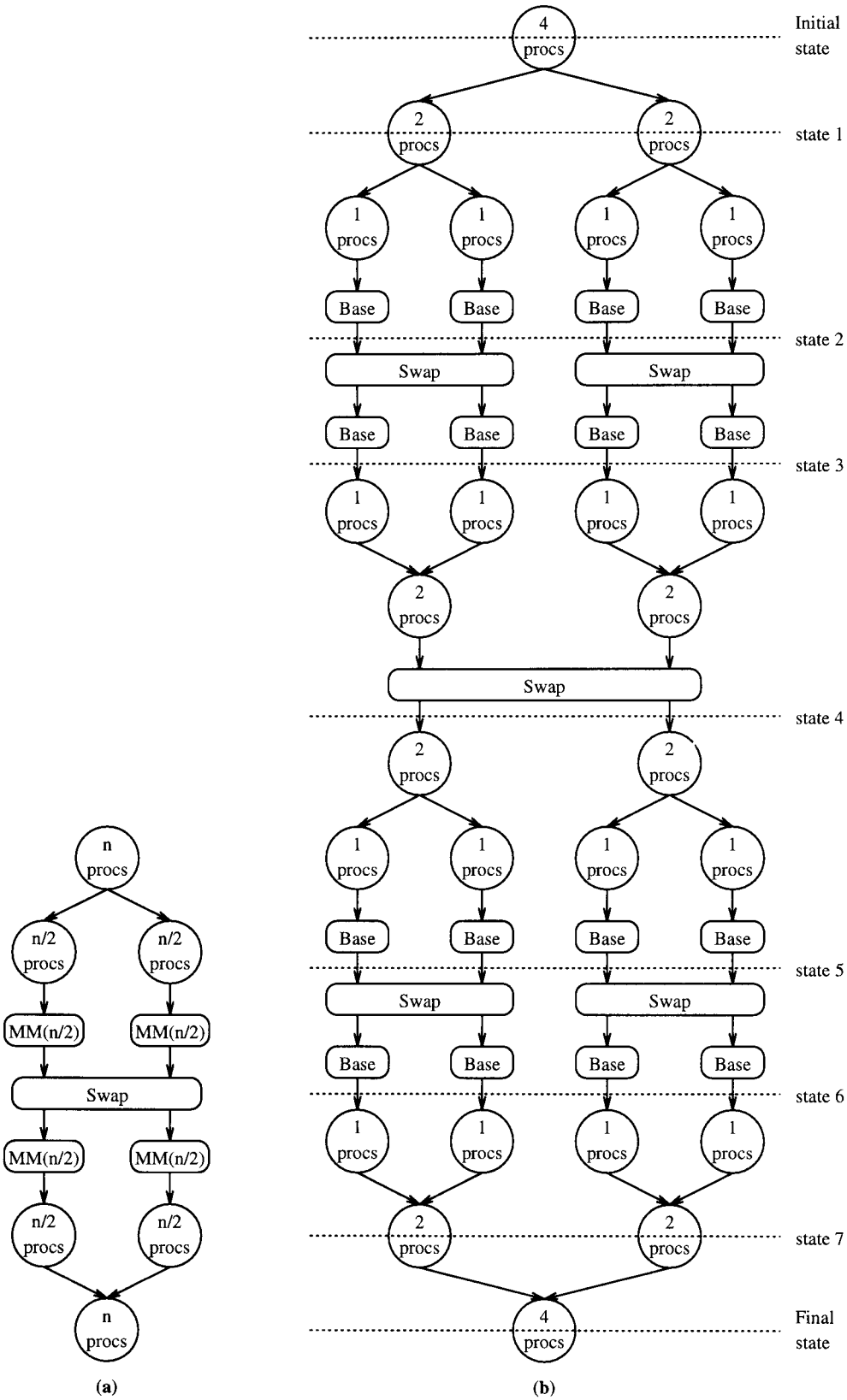

FIGURE 7 Schematic view of the execution of the matrix multiplication template. (a) Matrix multiplication on n processors. (b) Matrix multiplication on four processors after unrolling the recursion.

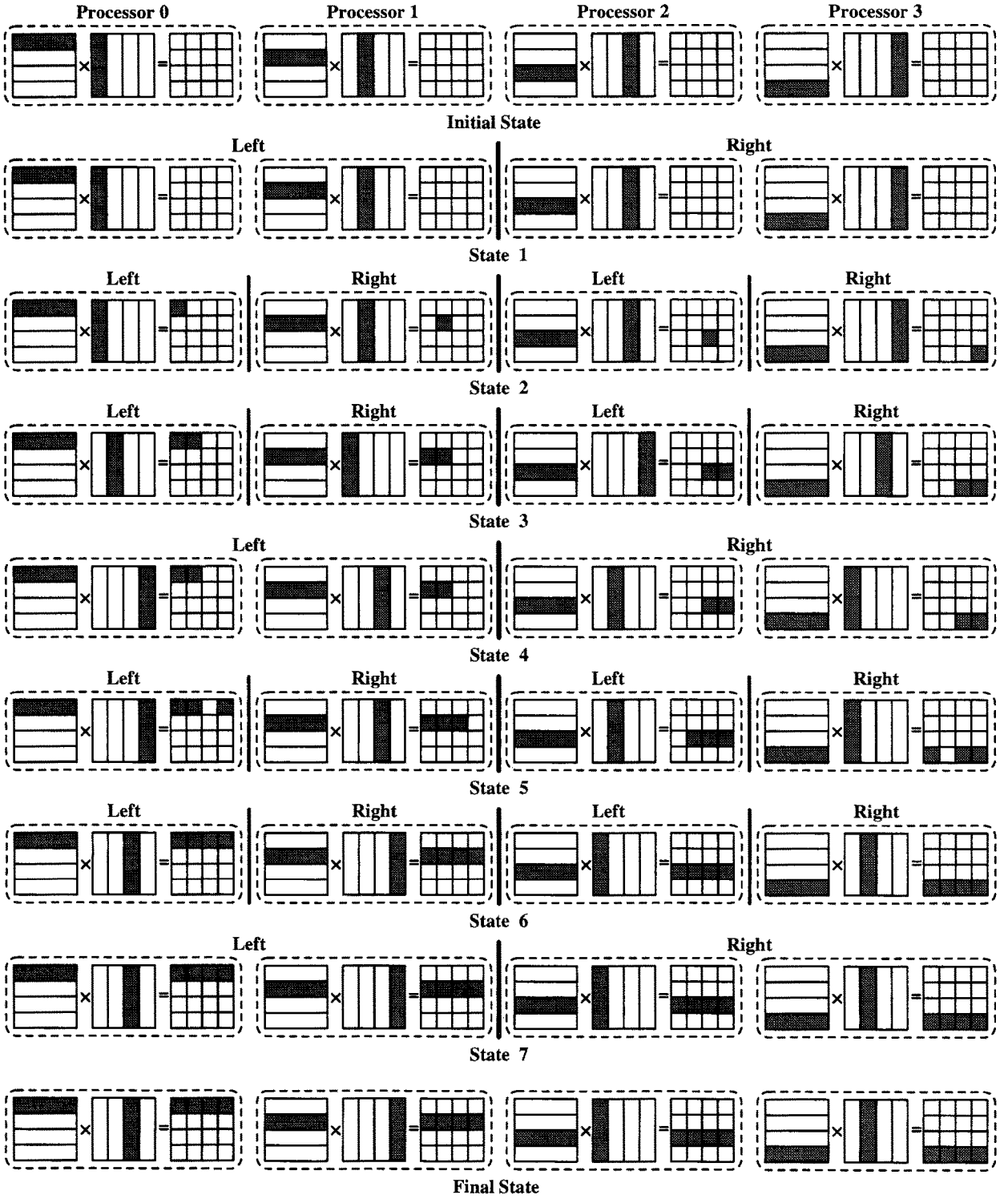

FIGURE 8 Snapshots of the data structures and processor partitions at the states labeled in Figure 7b. Shaded areas show the matrix blocks stored at a processor at the indicated state.

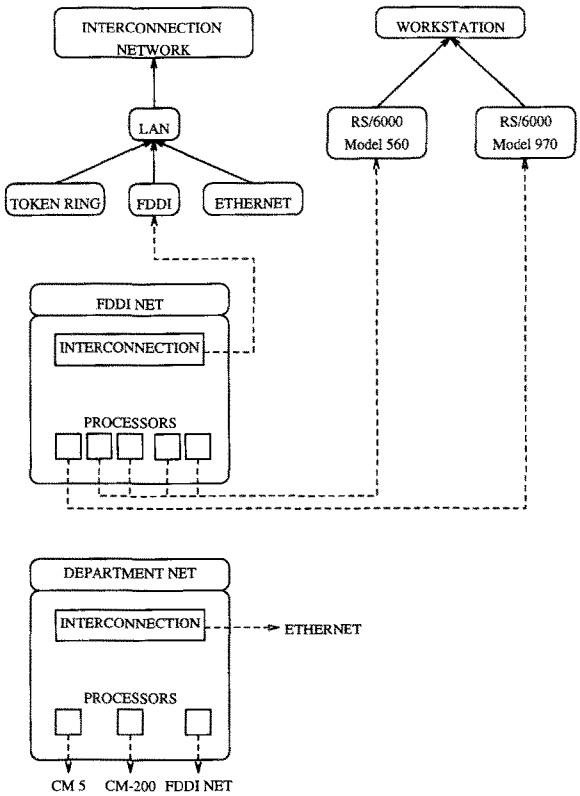

FIGURE 9 Frame representation of a typical processing environment.

to estimate their run-time on the specific processing environment.

We begin by introducing the terminology used for describing the model as shown in Table 1.

The input size is a vector, because some problems may have more than one input parameter. An example is the banded linear system solver, which has three input parameters: the number of unknowns, the bandwidth, and the tolerance. The input size vector of the subproblems can be formulated as a function of the original vector and the number of subproblems ($\vec{n}_s = g(\vec{n}, k)$). This function is very simple in most cases: The input size of the subproblems is obtained by simply dividing the input size of the original problem by k. For some applications, such as the banded-matrix-vector multiplication shown in Figure 4, a slightly more complex function is required.

Table 1. Terminology

$\vec{n}$	Input size vector
p	Number of processors
k	Number of subproblems generated by each divide operation
$\vec{n}_s$	Input size vector of the subproblems
$f()$	Sequential time
$S()$	Speedup
T_{par}	Parallel time
T_{comp}	Computation time
T_{comm}	Communication time
C_{comm}	Communication time for combine phase
D_{comm}	Communication time for divide phase
C_{comp}	Computation time for combine phase
D_{comp}	Computation time for divide phase
C_{sync}	Synchronization time for combine phase
D_{sync}	Synchronization time for divide phase

We make several simplifying assumptions in forming the performance prediction model:

- The divide tree is *complete* and *balanced.*
- The processing environment is homogeneous.
- The base functions are sequential.
- There is no overlapping between computation and communication.
- The effect of cache on the speedup is negligible.

The first three assumptions have no bearing on most processing environments, including the three used for the case study in the next section. Nevertheless, we plan to improve the model in the future to include arbitrary trees, heterogeneous processing environments, and nonsequential base functions.

The SPMD programs generated by our system currently do not support the overlapping of communication and computation. In our computational model, the opportunity for such overlap is limited, because it an be done only in the adjust functions. We do not consider this to be a liability as there is empirical and analytical evidence to suggest that communication-computation overlap has limited benefits [4, 5].

Can we ignore the impact of cache on the performance? Because we are using performance prediction to compare implementations, we are interested in relative performance rather than absolute performance. As long as cache effects do not change the relative performance of the implementations we are comparing, it is safe to ignore them. Our experience with the system, including the case study presented in this paper, validates this assumption. However, cache effects could be significant for certain applications and architectures. Refining the performance prediction model to include memory effects is one of our future goals, especially since our system has access to the required information—such as the input size, the memory access pattern of the application, and the memory hierarchy of the architecture.

Predictor functions Attached to each template is a function to compute its predicted performance. We will call this the *predictor* function. Parameters of the template are arguments of this function. Additionally, the predictor function has an extra argument denoting the type of the architecture. There are only three permissible values for this argument. These values and the associated architecture type are listed below:

1. SM: Shared memory machines.
2. DX: Distributed memory plateforms with an eXclusive access communication medium.
3. DN: Distributed memory plateforms with a nonexclusive access communication medium.

When the *Performance Predictor* receives the instantiated template from the *Search Engine* (see Figure 3), it invokes the predictor function with the appropriate values for the arguments. The predictor function returns an expression, which encapsulates the predicted performance in a format independent of the specific details of the architecture. The Performance Predictor uses the machine parameters it received from the Search Engine to reduce this expression to a number representing the predicted *speedup*.

The predictor function essentially evaluates a small set of expressions, some of them defined recursively. At the top level, this function is the same for all templates and is defined by the following set of equations with the first entry in each equation showing the set of architecture classes to which it is applicable:

The equations above merely reflect the structure of the divide-and-conquer template and hence remain the same for all templates. The predictor function of each template will have additional expressions to compute the application-specific details. Next we discuss these details and the methods used for computing them.

1. $f(\vec{n})$. We need the number of scalar floating-point operations, the number of vectorizable loops, and the number of vector operations in each such loop of the sequential base function to compute $f(\vec{n})$. The predictor function of each template computes these numbers using the input size vector $\vec{n}$.

   We assume that the vectorizable loops in the base function can be identified without prior knowledge of the target platform. In reality, a loop that vectorizes on one vector machine may not vectorize on another machine, primarily due to variations in the compiler technology. Because the templates would be developed by experts rather than novice users, we assume the base functions are coded in such a way that most compilers can vectorize them.
2. $D_{comp}(\vec{n}, p)$ and $C_{comp}(\vec{n}, p)$. Just as in the previous case, the predictor function computes the scalar and vector operation counts using the input size vector and the number of processors. These numbers correspond to computations in the adjust functions of the template.
3. $D_{comm}(\vec{n}, p)$ and $C_{comm}(\vec{n}, p)$. We make use of the structure of divide-and-conquer to formulate

---

$$\{\text{SM, DX, DN}\} \quad S(\vec{n}, p) = f(\vec{n})/T_{par}(\vec{n}, p)$$

$$\{\text{DX, DN}\} \quad T_{par}(\vec{n}, p) = T_{comp}(\vec{n}, p) + T_{comm}(\vec{n}, p)$$

$$\{\text{DN}\} \quad T_{comm}(\vec{n}, p) = \begin{cases} 0 & (p = 1) \\ T_{comm}(\vec{n}_s, p/k) + D_{comm}(\vec{n}, p) + C_{comm}(\vec{n}, p) & (p > 1) \end{cases}$$

$$\{\text{DX}\} \quad T_{comm}(\vec{n}, p) = \begin{cases} 0 & (p = 1) \\ k \times T_{comm}(\vec{n}_s, p/k) + D_{comm}(\vec{n}, p) + C_{comm}(\vec{n}, p) & (p > 1) \end{cases}$$

$$\{\text{SM}\} \quad T_{par}(\vec{n}, p) = T_{comp}(\vec{n}, p) + T_{sync}(\vec{n}, p)$$

$$\{\text{SM}\} \quad T_{sync}(\vec{n}, p) = \begin{cases} 0 & (p = 1) \\ k \times T_{sync}(\vec{n}_s, p/k) + D_{sync}(\vec{n}, p) + C_{sync}(\vec{n}, p) & (p > 1) \end{cases}$$

$$\{\text{SM, DX, DN}\} \quad T_{comp}(\vec{n}, p) = \begin{cases} f(\vec{n}) & (p = 1) \\ T_{comp}(\vec{n}_s, p/k) + D_{comp}(\vec{n}, p) + C_{comp}(\vec{n}, p) & (p > 1) \end{cases}$$

ID: CORR (Correspondence communication)
Parameters:
DIRECTION: LR (Left sends to the Right)
DATA SIZE: 40 Bytes
PROCESSORS: 8 (Four on the Left and four on the Right)

FIGURE 10 An example entry in the list of communication operations.

these expressions in an architecture-independent manner. The predictor function simply returns a list of communication operations in the adjust functions of the template. These operations are well-defined system primitives. Examples include *correspondence communication* and *mirror image communication*. In addition to the identifier of an operation, the list entries will also include the values of the parameters of this operation. An example is shown in Figure 10.

4. $D_{sync}(\vec{n}, p)$ and $C_{sync}(\vec{n}, p)$. The predictor function simply returns the number of synchronizations required in the adjust functions in a shared memory environment as the values of these expressions.

Notice that the predictor function is not computing the execution times directly. This is accomplished by the Performance Predictor using machine-specific details. We will show how this is done for the communication time component. Because communication primitives are only few in number, the cost function for each such primitive is stored explicitly in the machine database. Because the Performance Predictor knows the specific target machine of the template, it invokes the appropriate cost function of this machine for each entry in the list it receives from the predictor function. Computation and synchronization times are computed similarly, by combining the expressions returned by the predictor function with machine parameters.

The two-step computation of predicted performance described above has the advantage of decoupling the templates from the architectural details, while maintaining great flexibility in analytical performance prediction.

4 CASE STUDY: CONJUGATE GRADIENT

We present an example in which the scheme described in the previous sections is used to develop efficient parallel implementations of the CG method for three diverse architectures.

4.1 Mathematical Description of the CG Method

The CG method is an iterative scheme for solving linear systems of equations. Given a symmetric, positive definite, coefficient matrix A, and a vector b, it computes the solution vector of the linear system $Ax = b$ using the algorithm shown in Figure 11 [6].

4.2 An Algorithmic Template for CG

Each CG iteration involves a matrix-vector multiplication and a few dot products and SAXPYs. Each one of these operations can be expressed using a divide-and-conquer template. Thus, the CG method is represented in our scheme as a meta-template with several constituent divide-and-conquer templates.

The meta-template is parameterized using the following three parameters:

1. Processors used for matrix-vector multiplication (P1). The matrix-vector multiplication is the most compute-intense task in the CG iteration. Hence, it will be beneficial to use all the available processors for this operation. Thus, the size of the target platform essentially determines this parameter.
2. Processors used for the rest of the operations (P2). The poor granularity of dot product can affect the overall performance of the CG implementation. This parameter would let us improve the granularity by computing the dot product on a subset of the available processors. The system uses performance prediction to decide the optimum granularity depending on the machine characteristics and problem size.

The computational complexity of the CG iteration is concentrated in the matrix-vector multiplication. This is an $O(n^2)$ operation, whereas the rest of the computation has only linear time complexity. By tying together the granularities of all linear-time operations, we reduce the num-

$i = 0; g_i = h_i = b - Ax_i;$
while (not converged) do:
$\lambda_i = {g_i}^T h_i / ({h_i}^T A h_i)$
$x_{i+1} = x_i + \lambda_i h_i$
$g_{i+1} = b - Ax_{i+1}$
$\gamma_i = (g_{i+1} - g_i)^T g_{i+1} / ({g_i}^T g_i)$
$h_{i+1} = g_{i+1} + \gamma_i h_i$

FIGURE 11 CG algorithm.

```
Pre-conditions:
    Matrix A distributed as specified by the parameter MAT
    Vector h distributed on P1 nodes
    Vector g distributed on P2 nodes
    Vector x distributed on P2 nodes
    Vector b distributed on P2 nodes

CG-Template(P1,P2,MAT)
    switch (MAT)
        case Row-contiguous:
            Invoke DC-Template for row-oriented matrix vector multiplication
                with P1 as the number of processors to use.
            Invoke DC-Template to distribute the product vector from P1 nodes to P2 nodes.
        case Column-contiguous:
            Invoke DC-Template for column-oriented matrix vector multiplication
                with P1 as the number of processors to use.
            Invoke DC-Template to distribute the product vector from one node to P2 nodes.
        case Block-contiguous:
            Invoke DC-Template for block-oriented matrix vector multiplication
                with P1 as the number of processors to use.
            Invoke DC-Template to distribute the product vector from one node to P2 nodes.
    End switch (MAT).

    Invoke DC-Template to distribute the vector h from P1 nodes to P2 nodes.

    Invoke a series of DC-Templates for dot product and SAXPY
        with P2 as the number of processors to use.

    Invoke DC-Template to redistribute the vector h from P2 nodes to P1 nodes.
End CG-Template.
```

FIGURE 12 A parameterized meta-template for a single iteration of CG.

ber of parameters and thereby limit the size of the search space. In contrast, if we allow the granularity of each individual operation to change independently, the search space will have a combinatorial explosion. But this will not necessarily lead to a better solution, because the cost of redistributing the data (in the distributed memory machines) or synchronizations (in the shared memory machines) will eventually force the search engine to choose an implementation with the same granularity for these operations.

This is an example of the pruning of the search space using application-specific knowledge. As mentioned in Section 3, an alternate method is to start with a full set of parameters and then use branch-and-bound algorithms to eliminate unproductive branches of the search tree.

3. Decomposition of the coefficient matrix (MAT). The distribution of the coefficient matrix among the processors is an important parameter, because the adjust functions, the divide function, and the combine function will be determined by this distribution. We consider three different distributions: row-contiguous, column-contiguous, and block-contiguous.

In Figure 12, we present a parameterized meta-template for a single iteration of CG using pseudocode.

The CG-template in Figure 12 is called a meta-template because it is built by the composition of a number of *simple* divide-and-conquer templates. This example illustrates our layered approach to building adaptable parallel programs using divide-and-conquer.

The base templates used for building meta-templates fall into two categories on the basis of their functionality:

1. Basic linear algebra templates. Examples include matrix multiplication, dot product, and matrix-vector multiplication.
2. Data redistribution templates. Each linear algebra template has a set of pre-conditions and post-conditions, which are specified in terms of the distribution of the input and output data structures, respectively. When two templates are concatenated to form a meta-template, it might be necessary to redistribute the output data from the first to meet the pre-conditions on the second. Data redistribution templates are used for this purpose. Notice that these will be required only for distributed memory machines.

All constituent base templates of the CG template utilize either the Left-Right divide (LR) or the LEFT-RIGHT-TOP-BOTTOM divide (LRTB), the two standard divide functions. The LR divide splits the processor pool into two equal partitions, a LEFT partition and a RIGHT partition. It imposes a linear ordering on the processors. Figure 13 shows the LR divide of eight processors. The LRTB divide splits the processors into four partitions, LEFT-TOP, LEFT-BOTTOM, RIGHT-TOP, and RIGHT-BOTTOM. The processors are arranged logically as a square two-dimensional (2D) mesh as shown in Figure 14, where the LRTB divide is applied on a pool of 16 processors.

Linear Algebra Templates

Below we describe the linear algebra templates used for building the CG template:

1. Row-oriented matrix-vector multiplication: $Ab = c$. All data structures are distributed among the processors with row-contiguous distribution used for the matrix.

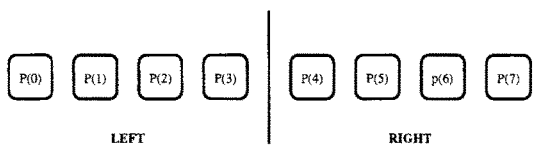

FIGURE 13 Left-Right division of eight processors.

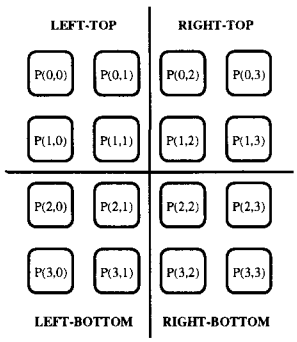

FIGURE 14 Left-right-top-bottom division of 16 processors.

Divide function: LR.
Pre-adjust function: None.
Base function: Sequential matrix-vector multiplication.
Post-adjust function: (1) Swap values of b between partitions. (2) Recursively apply template to both partitions.

2. Column-oriented matrix-vector multiplication: $Ab = c$. The input data structures are distributed among the processors with column-contiguous distribution used for the matrix. On completion, vector c will be accumulated on a single processor.
   Divide function: LR.
   Pre-adjust function: None.
   Base function: Sequential matrix-vector multiplication.
   Post-adjust function: The first processor on the LEFT gets the vector c from its counterpart on the RIGHT and adds it to its own c.
3. Block-oriented matrix-vector multiplication: $Ab = c$. The matrix is distributed block-wise among the processors arranged in a 2D mesh. The input vector b is distributed among the processors in column-major order. The output vector c is accumulated at a single processor. In the pre-adjust phase, subvectors are assembled at each node. In the post-adjust phase, the partial results are combined and accumulated.
   Divide function: LRTB.
   Pre-adjust function: Get the chunk of b from the *vertical* counterpart and store it at the appropriate location within the subvector being assembled.
   Base function: Sequential matrix-vector multiplication.
   Post-adjust function: (1) The FIRST processor in the LEFT-TOP and the FIRST processor in the LEFT-BOTTOM get c from their counterparts on the RIGHT and add it to their own c. (2) The FIRST processor in the LEFT-TOP gets the c from its counterpart on the BOTTOM and concatenates it to its own c.
4. Dot product: $d = ab^T$. The input vectors are distributed among the processors and the output is replicated at each processor.
   Divide function: LR.
   Pre-adjust function: None.
   Base function: Sequential dot product.
   Post-adjust function: Each processor adds its counterpart's partial result to its own.
5. SAXPY: $b = ax + y$.
   Divide function: LR.
   Pre-adjust function: None.
   Base function: Sequential saxpy.
   Post-adjust function: None.

Data Redistribution Templates

We have used two data redistribution templates to form the CG template.

1. Vector distribution. A vector stored at a single processor is distributed among all processors using this template.
   Divide function: LR.
   Pre-adjust function: The first processor on the LEFT sends the latter half of its vector to the first processor on the RIGHT.
   Base function: None.
   Post-adjust function: None.
2. Vector concatenation. A vector distributed among all processors is concatenated and stored at a single processor using this template.
   Divide function: LR.
   Pre-adjust function: None.
   Base function: None.
   Post-adjust function: The first processor on the LEFT gets the subvector from its counterpart on the RIGHT and concatenates this subvector to its own.

4.3 Generation of Efficient Programs on Diverse Platforms

The CG template is adapted to a specified target platform by tuning the values of the parameters described earlier. The Search Engine module of the system will invoke the Performance Predictor several times to determine the set of parameters that maximizes the speedup. For a machine with p processors, there are only $3(\log p + 1)$ leaves in the search tree, making exhaustive search possible.

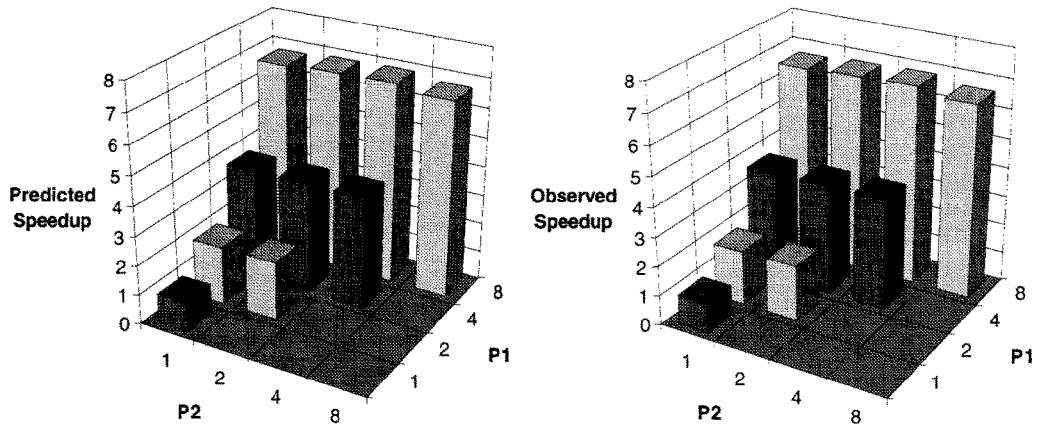

FIGURE 15 Predicted and observed performance of CG on SGI using ROW decomposition of the matrix. P1 is the number of processors used for matrix-vector multiplication and P2 is the number of processors used for the rest of the computation.

In this case study, we have considered three vastly different parallel processing envrionments as target platforms: an eight-processor Silicon Graphics Power Challenge shared memory machine, a 32-processor CM-5, and an FDDI network of four IBM RS/6000 model 560 workstations.

Adapting the Template to a Shared Memory Machine

Figures 15, 16, and 17 show the predicted and observed speedups for an input size of 1024 for row, column, and block distribution of the matrix, respectively. The predictions tend to be more optimistic than the actual performance, but in terms of the relative performance, the predicted values match with observations.

We will show how the Search Engine can make the correct decision using the performance prediction by closely examining the data for the four-processor machine. Figure 18 shows the predicted and observed performance for the nine possible combinations of the parameter values. These combinations are formed by the cartesian product of the sets {1, 2, 4} and {ROW, COLUMN, and BLOCK}. The first set denotes the permisible values of the parameter $P2$, the number of processors to be used for the dot product computations. The elements in the second set represent the three matrix decomposition options.

The best performance is predicted when $P2 = 4$ with row decomposition of the coefficient matrix. The actual implementations showed maximum speedup for the same values of these parameters.

On an eight-processor machine, predictions showed

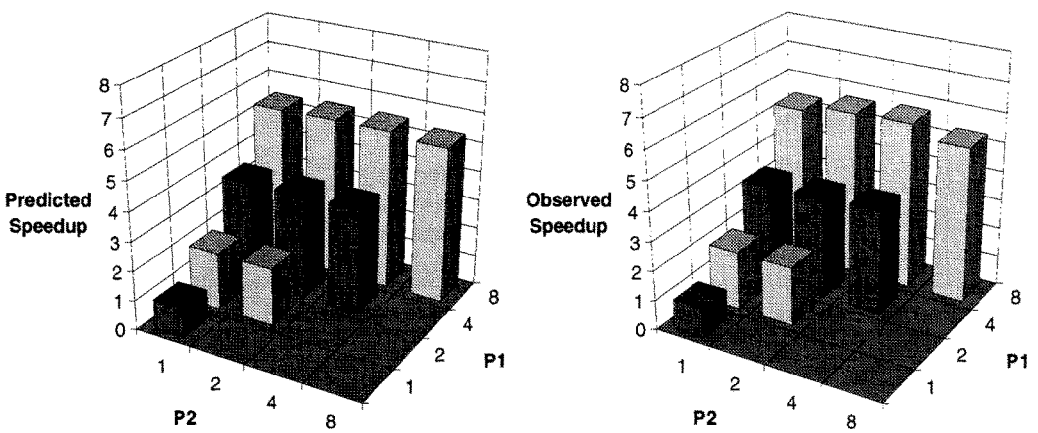

FIGURE 16 Predicted and observed performance of CG on SGI using COLUMN decomposition of the matrix.

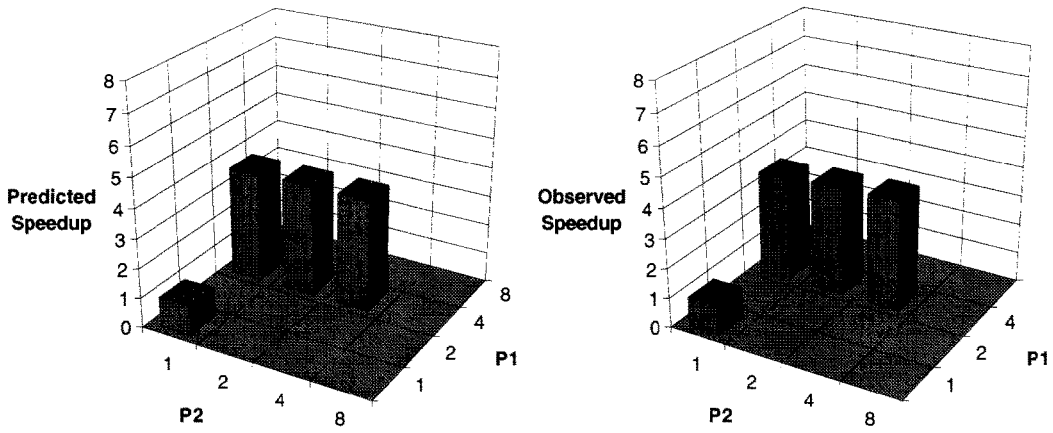

FIGURE 17 Predicted and observed performance of CG on SGI using BLOCK decomposition of the matrix.

row decomposition of the matrix along with $P2 = 4$ as the best choice of parameter values. These values were also validated by the observations.

In general, the Search Engine can make the following decisions based on the performance prediction:

1. The best decomposition scheme for the coefficient matrix is row-contiguous.
2. It is advantageous to use all processors for matrix-vector multiplication.
3. The rest of the computation is at best performed using four or fewer processors.

An explanation of these decisions is straightforward. The implementation that minimizes the synchronization points performs best on a shared memory machine. Because row-wise matrix-vector multiplication requires no synchronizations at all, this scheme beats the other options easily. It is advantageous to utilize all the available processors for matrix-vector multiplication, because this is a rather compute-intensive task. The granularity of the dot product computation that maximizes the performance is a function of the vector size as well as the machine parameters. For a vector size of 1024, four processors minimize the execution time.

Adapting the Template to a Workstation Network

Figure 19 shows the predicted and observed performance using row decomposition of the matrix for vary-

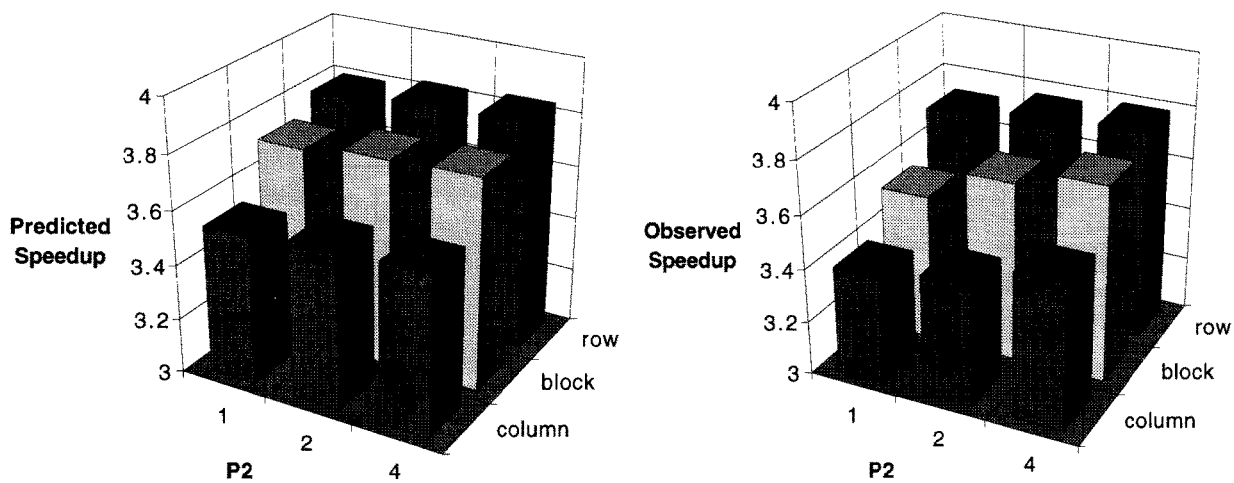

FIGURE 18 Predicted and observed performance of CG on a four-processor SGI for row, column, and block decomposition of the matrix.

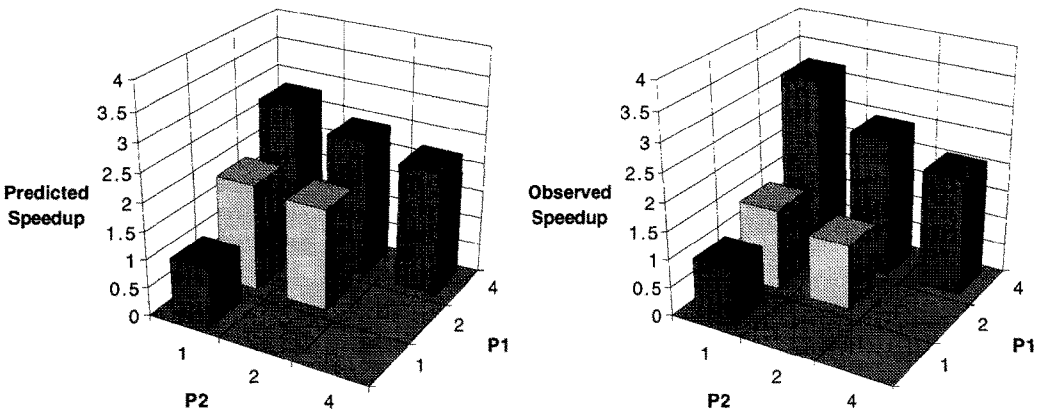

FIGURE 19 Predicted and observed performance of CG on workstation network using ROW decomposition of the matrix. P1 is the number of processors used for matrix vector multiplication and P2 is the number of processors used for the rest of the computation.

ing cluster sizes and dot product granularities. The problem size is kept the same as in the shared memory example — 1024 unknowns. Figure 20 shows the same information when the matrix is decomposed column-wise. Performance of the CG scheme using block distribution of the coefficient matrix is shown in Figure 21. The predictions enable the system to arrive at the best implementation on the workstation network: column decomposition of the coefficient matrix along with single processor execution of the dot product.

In the workstation environment, the execution time is minimized by the algorithm that minimizes the total number of messages generated, because the shared nature of the communication medium forces all messages to be serialized. This explains the selection of column decomposition for the workstation network. The extremely coarse nature of the network discourages parallel execution of the dot product computation.

Adapting the Template to a Multicomputer

Figures 22, 23, and 24 show the predicted and observed speedups on a CM-5 using row, column, and block decomposition, respectively. To see how predictions help in selecting the best values of the parameters, we will look closely at the data for a 16-processor machine and a 32-processor machine.

Figure 25 shows the predicted and observed perfor-

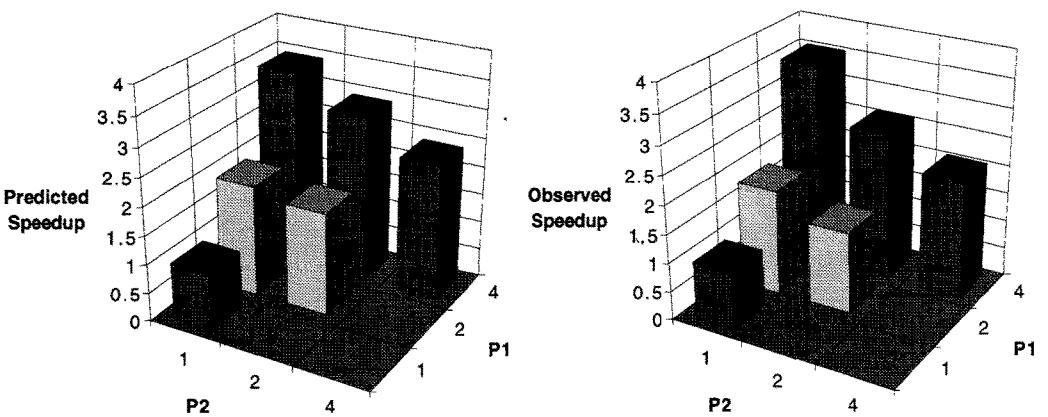

FIGURE 20 Predicted and observed performance of CG on workstation network using COLUMN decomposition of the matrix.

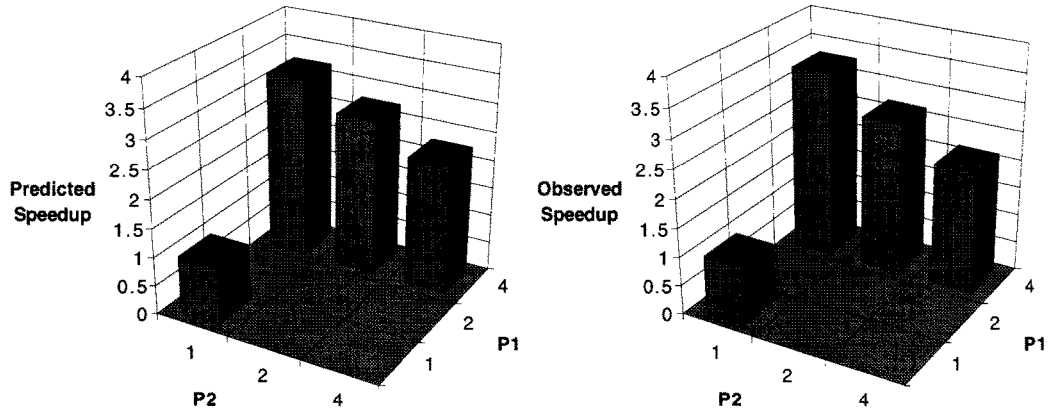

FIGURE 21 Predicted and observed performance of CG on workstation network using BLOCK decomposition of the matrix.

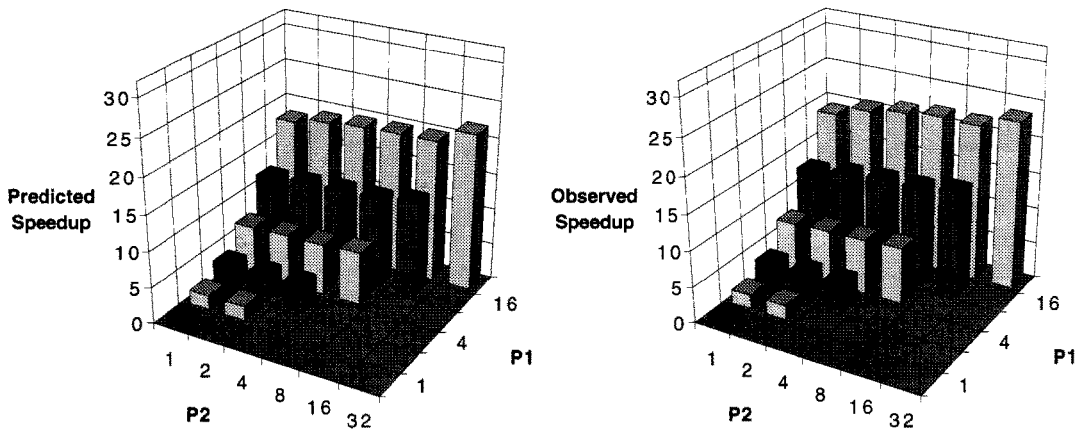

FIGURE 22 Predicted and observed performance of CG on CM-5 using ROW decomposition of the matrix. P1 is the number of processors used for matrix-vector multiplication and P2 is the number of processors used for the rest of the computation.

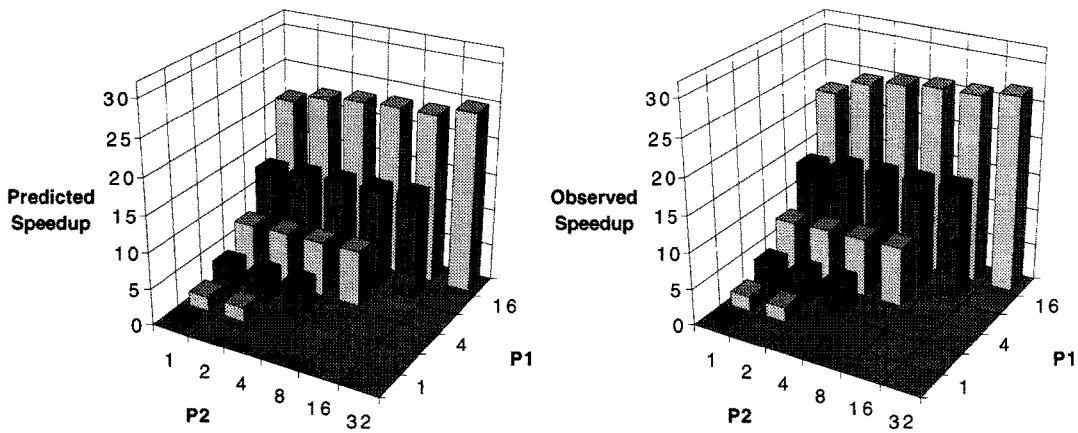

FIGURE 23 Predicted and observed performance of CG on CM-5 using COLUMN decomposition of the matrix.

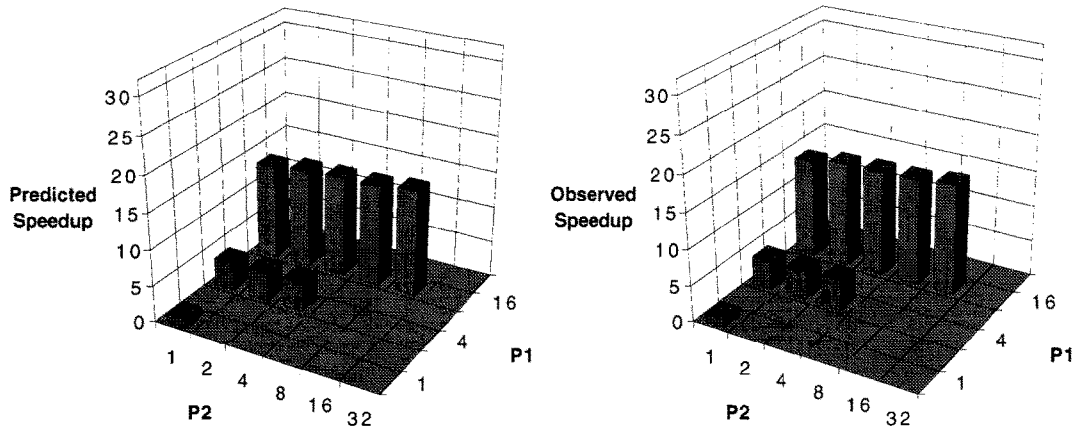

FIGURE 24 Predicted and observed performance of CG on CM-5 using BLOCK decomposition of the matrix.

mance for five different values of P2 and three different matrix decomposition options. Both predictions and observations agree on 16 as the best value of P2 and *block-contiguous* as the best matrix decomposition option.

The block decomposition of the matrix leads to the minimum number of communication steps in the algorithm and consequently to the best performance. The fine granularity of the machine justifies the spreading of the entire computation among all the available processors.

On a 32-processor machine, both predictions and observations show column decomposition outperforming row decomposition, with 32 as the optimal value of P2. In general, we see that the analytical performance prediction helps the Search Engine in choosing the appropriate values for the parameters, leading to an implementation that maximizes performance.

4.4 Comparison with Other Parallel Programming Systems

In the previous section, we gave performance figures for a parallel CG solver on diverse architectures programmed based on our approach. How does our system compare with other commonly used parallel programming tools in terms of performance? To answer this question, we implemented the CG method using CM Fortran on a CM-5 and Power Fortran on an SGI. Our best implementation on the 32-node CM-5 outperformed the CM Fortran program by a factor of

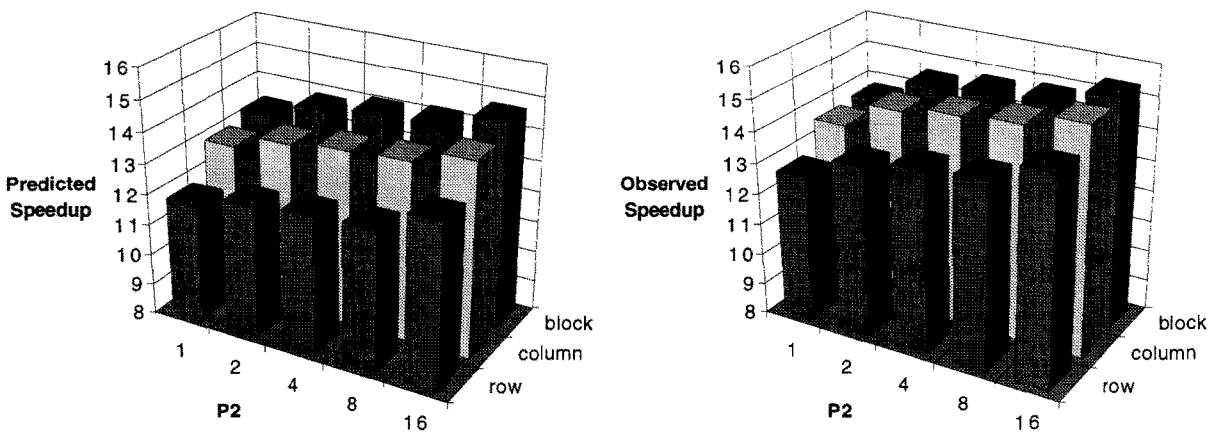

FIGURE 25 Predicted and observed performance of CG on a 16-processor CM-5 for row, column, and block decomposition of the matrix.

1.5. On the SGI multiprocessor, our performance was 1.7 times better than a Fortran program optimized using power Fortran directives.

4.5 Conclusions from the Case Study

Three key issues in parallel processing are performance, portability, and programmability. We believe that our method addresses all three of them, at the expense of generality. The loss in generality is not a serious drawback if the expressibility of the system is sufficient enough to cover most problems of practical interest. By using the CG method as an example, we have shown that an algorithm of significant practical interest can be represented using our system.

As detailed in the previous section, the case study corroborates our thesis that performance prediction along with the divide-and-conquer paradigm can provide architecture adaptability. The case study shows that the algorithms generated by the system can have efficient implementations on diverse processing environments. There are two reasons for this. First, the system is able to search the parameter space exhaustively, because this space is relatively small. The other reason for good efficiency is the low overhead of our implementation of the divide-and-conquer templates [7]. We have seen similar results from another case study involving a finite element ocean circulation model [8].

5 CODE GENERATION

Hatcher and Quinn describe a compiler for a data-parallel language targeted towards multiple-instruction multiple-data (MIMD) computers in [1]. Their compiler generates SPMD programs on distributed memory and shared-memory machines from data-parallel specifications. The task of our code generator is similar, albeit much simpler. In this section, we outline the implementation of the templates, and the mechanisms for handling communication calls and data distribution primitives.

5.1 Template Implementation

We use a set of higher-order functions to implement the templates. The base templates fall into four categories based on the presence or absence of the adjust functions. Templates in each category are implemented using a function associated with that category. Below we describe these categories and the associated functions:

```
void  PDC( basefun, dfun, cfun, prafun, posfun, p, dim, Tptr, DCptr )
void (*basefun) (); /* base function */
void (*dfun)    (); /* divide function */
void (*cfun)    (); /* combine function */
void (*prafun)  (); /* pre-adjust function */
void (*posfun)  (); /* post-adjust function */
int  p;             /* number of processors */
int  dim;           /* dimension of the processor grid */
struct TEMPLATE_DATA * Tptr;  /* pointer to application data */
struct DC_STATE * DCptr;      /* pointer to system data */
{
 int i, depth,tmp;
 tmp = 1;
 depth = 0;
 /* compute the depth of the DC-tree */
 while (tmp < p) {
  depth++;
  tmp <<= dim;
 }
 /* divide them with pre-adjusting */
 for (i=0; i<depth; i++) {
    (*dfun)(DCptr);
    (*prafun)(DCptr,Tptr);
 }
 /* activate the base function now  */
 (*basefun)(DCptr,Tptr);
 /* combine them  (with post-adjusting)  */
 for (i=0; i<depth; i++) {
    (*posfun)(DCptr,Tptr);
    (*cfun)(DCptr);
 }
 return ;
}
```

FIGURE 26 C function PDC.

1. Templates with pre-adjust and post-adjust functions. The associated function is called **PDC**. An example of this type of template is the block-oriented matrix multiplication. The code in Figure 26 shows the implementation of the PDC function in C.

   The base function, the divide-and-combine functions, and the adjust functions are passed as arguments to the higher-order function **PDC**. Additionally, the number of processors and the dimension of the processor grid, along with a pointer to the application data and a pointer to the system data, are also passed as arguments. The dimension of the processor grid refers to the logical 1D or 2D mesh embedded in the topology of the machine. Divide-and-combine functions assume the existence of such an embedding. The system data structures remain the same for all templates. They simulate the traversal up and down the divide-and-conquer tree. The application data will change for each template, because they are specific to the problem being solved.

   The routine first computes the depth of the divide-and-conquer tree. Here we assume the default base predicate; the recursion terminates when there is only a single processor in each

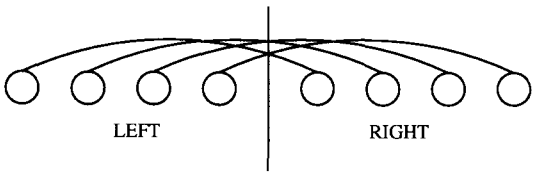

FIGURE 27 Correspondence communication with LEFT-RIGHT divison.

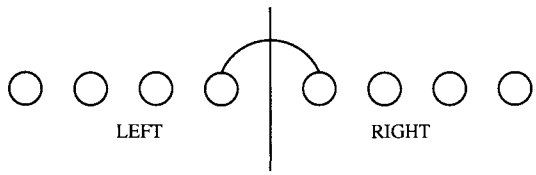

FIGURE 29 Last to First communication with LEFT-RIGHT division.

partition. Because the depth of the divide tree is known a priori, we replace the recursion with two iterative loops with the call to the base function placed between them. The first loop simulates going down the divide tree from the root to the leaves. At the leaves, we invoke the base function. The second loop simulates going up the tree from the leaves to the root.

2. Templates with only pre-adjust functions. We call the associated function **prePDC**. The vector distribution template presented earlier is an example of this category. This function is derived from **PDC** by replacing the second iteration by a single function call, which has the effect of transforming the state from the leaves to the root in a single operation.
3. Templates with only post-adjust functions. Similar to the previous case, we replace the first iteration by a function call to derive **postPDC** from **PDC** to represent this category. Most templates presented earlier in this article are examples of this type.
4. Templates with no adjust functions. We call the associated function **purePDC**. The SAXPY operation used earlier in the CG template is an example of this category.

5.2 Communication

The interprocessor communication is limited to the adjust functions. These communications appear as calls to a handler function in the pre-adjust and post-adjust functions. The handler functions are responsible for generating machine-specific communication calls. The divide-and-conquer paradigm has the advantage of requiring regular communication patterns. The handler functions exploit this regularity to generate machine-specific, efficient implementations.

Each divide function has an associated set of patterns. We present below some of the most frequently used communication patterns for the 1D left-right divide operation:

1. Correspondence communication. This pattern is shown in Figure 27 below. Each processor communicates with the corresponding processor on the other partition.
2. Mirror image communication. As shown in Figure 28, each communication link is symmetrical with respect to the centerline.
3. Between the Last processor on the LEFT partition and the First processor on the RIGHT partition. This pattern results in neighbor communication, as shown in Figure 29.
4. Between the First processor on the LEFT partition and the First processor on the RIGHT partition. This pattern is shown in Figure 30.

Each one of the above patterns has three different variations based on the message direction: *Left to Right*, *Right to Left*, or *Duplex*. The information passed to the handler function includes a pattern identifier, message direction, address at the source node, address at the target node, and the size of data being transferred.

On shared memory machines, the shared memory can be used for the information exchange. The source node writes to the appropriate locations in the shared

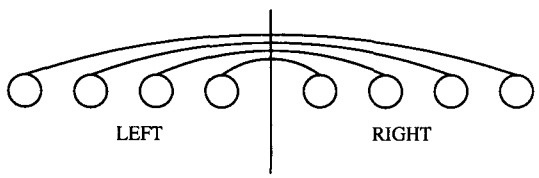

FIGURE 28 Mirror image communication with LEFT-RIGHT division.

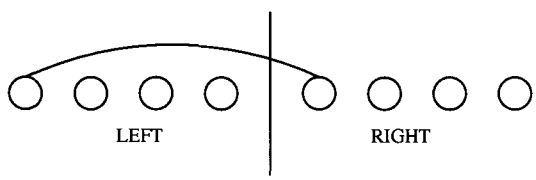

FIGURE 30 First to First communication with LEFT-RIGHT division.

memory, and the target node reads them. But unlike in message-passing architectures, processors should synchronize to ensure that a *read* does not happen before a *write*. It is the responsibility of the handler function to insert the necessary synchronization calls.

5.3 Data Distribution Primitives

A meta-template may include several data distribution primitives to ensure compatibility between the constituent base templates. We hae used two such primitives in the CG template, vector concatenation and vector distribution. These primitives themselves are implemented as divide-and-conquer templates. But unlike other base templates, these are required only on distributed memory machines. On platforms with shared memory, there is clearly no need for explicit data redistribution and the Code Generator suppresses these calls.

6 USER INTERFACE

Our goal is to isolate the computational scientist from the details of parallel hardware and algorithms. This is achieved by having a very clear delineation of roles for the participants. We briefly describe these roles below:

1. Enviroment builder: The role of the environment builder is to develop the enabling technology. This includes the user interface, search engine, and the performance prediction tools.
2. Environment maintainer: The primary role of the environment maintainer is to extend the problem-solving capabilities of the system by generating appropriate algorithmic templates and by providing the necessary parameters to drive the performance predictor.
3. Computational scientist: At the highest level, the users interact with the system by specifying the problem to be solved using a high-level notation. Thus, the computational scientist is able to concentrate on the science, without worrying about the details of the underlying computing environment.

The high-level notation mentioned above serves as the user interface. There are three important criteria for the selection of this notation:

1. The computational scientist should be familiar and comfortable with the notation.
2. It should be possible to represent computational science problems easily using the notation.
3. It should be possible to extract templates easily from "programs" written in the notation.

A good candidate for the user interface is the notation used by MATLAB, a well-known software package for numerical computation, data analysis, and graphics [9]. Complex numerical applications can be coded using the MATLAB notation. A front end will scan and parse the MATLAB programs, as well as generate an intermediate representation consisting of calls to high-level functions. For each function call, the *template selector* shown in Figure 3 will then search the database for matching templates.

7 RELATED WORK

Our performance prediction model is based on the work done by Clement and Quinn in predicting the performance of scalable data-parallel programs on multicomputers [3]. Parasher et al. [10] have proposed the use of performance prediction to improve the performance of parallel programs. As part of a high-performance Fortran (HPF)/Fortran 90D application development environment, their performance prediction framework helps users in selecting appropriate compiler directives.

The algebraic model of divide-and-conquer, introduced by Mou and Hudak in [2], has influenced the design of our templates. Chandy's group at Caltech [11] and Dongarra's group at Tennessee [12] are also investigating the use of "templates" for high-performance scientific computing. Our method differs from these and other work on templates by introducing a novel approach to architecture adaptability, combining parameterized templates with analytical performance prediction.

A vast amount of research in parallel programming has been motivated by the desire to make parallel programming easier and portable. Here we attempt to categorize these efforts and comment on their impact in the real world.

7.1 Architecture-Independent Programming Languages and Systems

A high-level parallel programming language can provide limited architecture independence and programmability if efficient compilers are available on several machines. Data-parallel C [1] and Fortran 90 [13] are two examples. Chameleon [14] is a shared memory library designed for architecture independence. Sev-

eral message-passing libraries—most notably PVM [15], MPI [16], and p4 [17]—are in existence to facilitate portable parallel programming on distributed memory machines. Crowl's Matroshka system presents the framework of a parallel programming language which has the ability to adapt to different architectures [18]. A Matroshka-based program exposes all the available parallelism in an application. Using annotations, a subset of this parallelism is selected for execution on a specific machine.

Selection of the appropriate algorithm is still up to the user when a general-purpose, procedural language is used for problem solving in parallel and distributed environments. This implies that effective portability is not achieved. For sequential machines, because there is only one machine model, this is not a problem. For distributed computing, this difficulty impedes the development of easily portable problem-solving environments. The message-passing libraries (such as PVM, MPI, p4, and Illinois Fast Messages [19]) merely become accessories to our system because our goal is the automation of algorithm design and implementation using the available programming tools.

Another approach to designing machine-independent parallel programming relies on implicit parallelism. The parallel programming languages based on the functional paradigm falls into this category. A number of such languages have been proposed, including EPL [20], Crystal [21], ParAlfl [22], SISAL [23], Id [24], and a parallel dialect of Haskell [25]. A functional language can provide a higher level of abstraction to the programmer, compared to explicitly parallel procedural languages. But the objective of our work is to provide to users a much higher level of abstraction. We are striving for a parallel-processing environment where the users are able to concentrate on the problems being solved, rather than the selection of the algorithms and their implementation in some *programming* paradigm. The choice of the programming paradigm, whether it is functional or procedural, is of little help to the computational scientist in designing the algorithm.

The object-oriented programming paradigm has also made an impact on machine-independent parallel programming. Several object-oriented parallel programming languages, systems, and models are in existence, including Mentat [26], pC++ [27], Concurrent Smalltalk [28], Illinois Concert System [29], Charm++ [30], Compositional C++ [31], and Actors [32]. Object technology makes it easy to separate the interface from the implementation. By hiding the architecture-specific parts in the implementation, an object-oriented parallel program will give the users a machine-independent interface. Several parallel class libraries (e.g., [33], [34], and [35]) and object-oriented frameworks (e.g., [36], [37], and [38]) have been built to exploit the software-engineering advantages of object technology. A key difference between our methodology and the object-oriented approaches mentioned above is that we are directly attacking the difficult problem of generating optimized implementations of object-oriented programs.

7.2 Algorithm Architecture Mapping

Representing the algorithms and architectures as task graphs, graph embedding can be used to generate parallel programs that adapt to machine topologies. The Oregami project [39] is an example of tool development using this approach.

In practice, graphical representations of parallel programs are complex and may contain several communication patterns intermingled. Further, with the advance of wormhole routing, communication overheads are dominated by message startup times rather than the distance between the communicating processors or link congestion. In short, complex task graphs make this approach impractical, and the change in technology renders it irrelevant.

7.3 Multiprocessor Scheduling

Another attempt at solving the problem resulted from looking at parallel processing as precedence-constrained multiprocessor scheduling with interprocessor communication delays. The tools developed by El-Rewini and Lewis [40] for "scheduling parallel program tasks onto arbitrary target machines" are examples of this approach.

As in the mapping problem, parallel algorithms are represented as task graphs. The size and complexity of the task graphs of real-life applications make this scheme impractical.

8 FUTURE WORK

We plan to use this method to develop domain-specific problem-solving environments (PSE) and application-oriented compilers targeted to linear algebra and partial differential equations (PDE). PSEs and compilers based on this method will be architecture independent because the description of the processing environment is an independent parameter.

Although we focused on templates based on PDC in this article, the methodology presented here could be applied to other models as well. A case in point is the *sequential* divide-and-conquer (SDC), where

dependencies exist between subproblems [41]. Templates based on SDC may be used to generate *pipelined* parallel programs. There are architecture-problem combinations where this approach will give the best results. Dynamic programming (DP), like the divide-and-conquer method, is a problem-solving strategy that solves problems by combining the solutions to subproblems [42]. But the structure of a DP-based template will be very different from that of a PDC template. There are application domains that can benefit from the automatic parallelization that uses DP templates as the seeds. We plan to extend the scope and solving power of our methodology by designing templates based on other paradigms, such as DP and SDC.

Most of the future work will be focused on three areas: performance prediction, the database of algorithm templates, and code generation. The performance prediction needs to be improved to take into account memory effects. Automatic generation of code for a specific target machine based on the template with best predicted performance is an area which requires further work.

The database of algorithm templates will be organized with a hierarchical structure. At the bottom, we will have basic linear algebra kernels, data distribution and communication primitives, and divide/combine functions. On top of this layer, nontrivial applications—such as CG method, 2D FFT, banded-system solver, and eigensystem solver—will be built. Numerical models and PDE solvers will form yet another layer. We believe this layered approach will have the expressibility to solve most problems in scientific computing, including irregular and unstructured problems.

REFERENCES

[1] P. J. Hatcher and M. J. Quinn. *Data-Parallel Programming on MIMD Computers*. Cambridge, MA: MIT Press, 1991.

[2] Z. G. Mou and P. Hudak. "An algebraic model for divide-and-conquer algorithms and its parallelism." *J. Supercomput.*, vol. 2, pp. 257–278, 1988.

[3] M. J. Clement and M. J. Quinn. "Analaytical performance prediction on multicomputers." in *Proc. Supercomputing '93*, 1993, p. 886.

[4] M. J. Quinn and P. J. Hatcher. "On the utility of communication-computation overlap in data-parallel programs." *J. Parallel Distrib. Comput.* (in press).

[5] S. Hiranandany, K. Kennedy, and C.-W. Tseng. "Evaluating compiler optimizations for Fortran D." *J. Parallel Distrib. Comut.*, vol. 21, pp. 27–45, 1994.

[6] E. Polak. *Computational Methods in Optimization: A Unified Approach*, chap. 2. New York: Academic Press, 1971, pp. 44–66.

[7] S. Kumaran and M. J. Quinn. "Divide-and-conquer programming on MIMD computers." in *Proc. 9th Int. Parallel Processing Symp.*, 1995, p. 734.

[8] S. Kumaran, R. N. Miller, and M. J. Quinn. "Architecture-adaptable finite element modeling: A case study using an ocean circulation simulation." in *Proc. Supercomputing '95* (in press).

[9] Math Works, Inc., *The Student Edition of MATLAB: Version 4 User's Guide*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

[10] M. Parashar, S. Hariri, T. Haupt, and G. C. Fox. "Interpreting the performance of HPF/Fortran 90D." in *Proc. Supercomputing '94*, 1994, p. 743.

[11] K. M. Chandy. "Concurrent program archetypes." in *Proc. 1994 Scalable Parallel Libraries Conf.*, 1995, p. 1.

[12] R. Barrett, M. Berry, T. F. 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*. Philadelphia: SIAM Press, 1993.

[13] M. Metcalf and J. Reid. *Fortran 90 Explained*. Oxford, England: Oxford Science Publications, 1990.

[14] G. Alverson and D. Notkin. "Abstracting data-representation and partition-scheduling in parallel programs." in *Proc. Int. Symp. Shared Memory Multiprocessing*, 1991, p. 138.

[15] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's Guide and Reference Manual*. Oak Ridge, TN: Oak Ridge National Laboratory, 1994.

[16] Message Passing Interface Forum. "The MPI message passing interface standard." University of Tennessee, Knoxville, Tech. Rep. CS-94-230, 1994.

[17] R. Butler and E. Lusk. "User's guide to the p4 parallel programming system." Argonne National Laboratory, Argonne, IL, Tech. Rep. ANL-92/17, 1992.

[18] L. A. Crowl. "Architectural adaptability in parallel programming." PhD Dissertation, University of Rochester, 1991.

[19] S. Pakin, M. Lauria, and A. Chien. "High performance messaging on workstations: Illinois fast messages (FM) for myrinet," in *Proc. Supercomputing '95*.

[20] B. K. Szymanski. "EPL—Parallel programming with recurrent equations." *Parallel Functional Languages and Compilers*, in B. K. Szymanski, Ed., New York: ACM Press, 1991, pp. 51–104.

[21] M. Chen, Y. Choo, and J. Li. "Crystal: Theory and pragmatics of generating efficient parallel code." in *Parallel Functional Languages and Compilers*, B. K. Szymanski, Ed., New York: ACM Press, 1991, pp. 255–308.

[22] P. Hudak and L. Smith. "Para-functional programming: A paradigm for programming multiprocessor systems." in *Thirteenth Annual ACM Symposium on Principles of Programming Languages*. New York: ACM Press, 1986, pp. 243–254.

[23] A. P. W. Bohm, D. C. Cann, J. T. Feo, and R. R. Oldehoeft, "SISAL 2.0 reference manual," Computer Science Department, Colorado State University, Fort Collins, CO, Tech. Rep. CS-91-118, 1991.

[24] K. Ekanadham, "A perspective on Id," *Parallel Functional Languages and Compilers*, B. K. Szymanski, Ed., New York: ACM Press, 1991, pp. 197–254.

[25] P. Hudak, "Para-functional programming in Haskell," in *Parallel Functional Languages and Compilers*, B. K. Szymanski, Ed., New York: ACM Press, 1991, pp. 159–196.

[26] A. S. Grimshaw, "Easy to use object-oriented parallel programming with Mentat," *IEEE Comput.*, pp. 39–51, May 1993.

[27] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Yang, "Distributed pC++: Basic ideas for an object parallel language," presented at the Object Oriented Numerics Conf., 1993.

[28] Y. Yokote and M. Tokoro, "Experience and evolution of concurrent smalltalk," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, New York: ACM Press, 1987, pp. 406–415.

[29] A. A. Chien, "Efficient concurrent object oriented programming: Lessons from the Illinois concert system," in *Proc. Parallel Object-Oriented Methods and Applications Conf.*, 1994.

[30] L. V. Kale, "CHARM: Modularity and reuse in parallel object-oriented software," in *Proc. Parallel Object-Oriented Methods and Applications Conf.*, 1994.

[31] C. Kesselman, "Compositional C++: A parallel object-oriented programming language," in *Proc. Parallel Object-Oriented Methods and Applications Conf.*, 1994.

[32] G. Agha, *Actors—A Model of Concurrent Computation for Distributed Systems*, Cambridge, MA: MIT Press, 1986.

[33] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley," "ScaLAPACK: A portable linear algebra library for distributed memory computers—design issues and performance," University of Tennessee, Tech. Rep. CS-95-283, March 1995.

[34] D. Quinlan and M. Lemke, "P++, a parallel C++ class library," presented at the Object Oriented Numerics Conf., Mississippi State University, MS, 1993.

[35] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka, "Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools," presented at the Object Oriented Numerics Conf., Mississippi State University, MS, 1994.

[36] R. Armstrong, "Use of frameworks for scientific computation in a parallel distributed environment," in *Proc. Parallel Object-Oriented Methods and Applications Conf.*, 1994.

[37] J. F. Karpovich, M. Judd, W. T. Strayer, and A. S. Grimshaw, "A parallel object-oriented framework for stencil algorithms," in *Proc. HPDC-2*, 1993, p. 34.

[38] J. Brown, "MaTRiX++: An object-oriented environment for parallel high-performance matrix computations," in *Proc. Parallel Object-Oriented Methods and Applications Conf.*, 1994.

[39] V. M. Lo, S. Rajopadhye, S. Gupta, D. Keldsen, M. A. Mohamed, B. Nitzberg, J. A. Telle, and X. Zhong, "Oregami: Tools for mapping parallel computations to parallel architectures," *Int. J. Parallel Prog.*, vol. 20, pp. 237–270, 1991.

[40] H. El-Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *J. Parallel Distrib. Comput.*, vol. 9, pp. 138–153, 1990.

[41] Z. G. Mou, S. Anderson, and P. Hudak, "Parallelism in sequential divide-and-conquer," Yale University, Department of Computer Science, Tech. Rep. YALEU/DCS/TR683, Feb. 1989.

[42] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Reading, MA: Addison-Wesley, 1974.