Program Structuring for Effective Parallel Portability

Gail A. Alverson and David Notkin, Member, IEEE

Abstract—The tension between software development costs and efficiency is especially high when considering parallel programs intended to run on a variety of architectures. In the domain of shared memory architectures and explicitly parallel programs, we have addressed this problem by defining a programming structure that eases the development of effectively portable programs. On each target multiprocessor, an effectively portable program runs almost as efficiently as a program fine-tuned for that machine. Additionally, its software development cost is close to that of a single program that is portable across the targets. Using our model, programs are defined in terms of data structure and partitioning-scheduling abstractions. These are activities identified as substantially affecting the performance of parallel programs; activities whose most efficient implementations can differ on the basis of the algorithm, multiprocessor, and even run-time data values and parameters. Low software development cost is attained by writing source programs in terms of abstract interfaces and thereby requiring minimal modification to port; high performance is attained by matching (often dynamically) the interfaces to implementations that are most appropriate to the execution environment. We include results of a prototype used to evaluate the benefits and costs of this approach.

Index Terms—Parallel languages and run-time systems, parallel programming, portability, software engineering.

I. INTRODUCTION

SOFTWARE development is a principal cost in the use of parallel computers. Software development costs can be significantly reduced by writing portable parallel programs. A portable program requires minimal source code changes to run on multiple targets. Thus, when the same solution is needed on different multiprocessors, a parallel programmer can substantially decrease the required work by writing one solution that is portable across these machines.

Despite this strong motivation to write portable programs, in practice there is relatively little inter-machine reuse of parallel applications. The key reason for this is performance.

Portability in the sequential domain does not generally sacrifice performance as the target of sequential programs and compilers, the von Neumann machine, defines a set of facilities that all sequential machines provide efficiently. Although several parallel analogues to the von Neumann machine have been proposed [44], [46], [49], none has been widely accepted. Given the wide range of parallel architectures and the lack of a unifying abstract machine, a program written or compiled for one architecture will often perform poorly on another. As a simple example, a portable congruence transformation program we developed achieved near perfect speedup on 16 processors of a Sequent multiprocessor but had a speedup of less than one using 20 processors of a Butterfly multiprocessor.

In contrast, a tuned Butterfly congruence transformation program achieved a speedup of nearly 19. Clearly, programmers unwilling to sacrifice performance will be driven to writing fine tuned codes for each target multiprocessor. But this specialization incurs a high software development cost.

Demonstrably, there is a tension between parallel software development costs and performance. Portable programs reduce software development costs but often at the price of performance. Fine tuned programs yield the desired performance but at a significant development cost. The challenge is to simultaneously attain the benefits of portable programming with respect to development costs and the benefits of customized programming with respect to performance. We call such programs, effectively portable programs.

Our approach to the challenge is in the form of an extensible model and system called Chameleon [2]. Chameleon encapsulates activities central to portable performance, namely data representation and partitioning-scheduling control, in source level programming abstractions. The abstractions enable the most effective implementations for the activities to be used without modifying the source program. Here, effectiveness is based on characteristics of the architecture, algorithm, and even run-time values of behavior. Low software development costs result because the source is portable. High performance results because the most effective implementation is used in each situation.

Chameleon is targeted to a specific domain: explicitly parallel programs, uniform and nonuniform memory access time (UMA and NUMA) shared memory architectures, and interval and tree structured algorithms. Chameleon provides several software engineering advantages such as extensibility, library reusability, and a clear separation between algorithmic and performance-improving program statements. These features, together with the balance of efficiency and portability, lead Chameleon to be an attractive medium for parallel programs.

This paper details the Chameleon approach to effectively portable parallel programming. We include results of a set of Chameleon programs run on a Sequent (UMA) and a Butterfly (NUMA) multiprocessor. The results concretely support the thesis of our work: a balance between software development costs and performance can be attained through exploiting

Manuscript received September 11, 1991; revised March 11, 1992. This work was supported in part by a Natural Sciences and Engineering Research Council of Canada scholarship, in part by NSF Grants CCR-8619663 and CCR-8858804, and in part by the Advanced Systems Laboratory, Boeing Computer Services. The research reflects work of the authors while at UW.

G. A. Alverson is with Tera Computer Company, Seattle, WA 98103.
D. Notkin is with the Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195.
IEEE Log Number 9210586.
user-level data and control abstractions. Moreover, the results confirm that the abstractions provide other important advantages corresponding to well engineered programs.

Section II discusses alternate approaches to effective portability. Section III describes our target domain in detail and motivates the approach we pursued. Sections IV through VII describe Chameleon. Section VIII concretely evaluates the performance of Chameleon, reporting results of our prototype. Section IX relates our work to other parallel languages and systems seeking portability. Section X highlights conclusions drawn from our experiences.

II. APPROACHES TO EFFECTIVE PORTABILITY

An effectively portable program attains high performance across a class of architectures while incurring low software development costs. Because it is a little unrealistic to achieve the best of both worlds without some change, we define what is meant by the terms high and low in practice.

High performance means that the execution time of an effectively portable program, on each target machine, is close to that of a fine-tuned program for that machine. Low software development cost means that the cost of creating an effectively portable program is close to the cost of creating a strictly portable program.\(^2\) The definitions are necessarily loose because actual percentages for how close is close to are subjective. The degree of performance and portability loss tolerable is largely dependent on the user and the application. The goal is thus to support effectively portable programs, where these programs may give up a little in the cost and performance dimensions, but will still retain a commendable placing in each dimension (Fig. 1).

There are at least four ways to approach effective portability. These approaches are not necessarily disjoint and often can be used together to further improve efficiency.

- Parallelizing compilers [9], [36], [39], [43], [51], also known as restructuring compilers, restructure sequential programs into parallel form. The target of a parallelizing compiler can range from the parallel language supplied by the multiprocessor vendor to the machine code of the multiprocessor. In the latter case, the compiler must identify, partition, and schedule the parallel tasks of the program.

Parallelizing compilers have the potential to support effectively portable programs. Software development costs are low both because of the relative ease in writing sequential code and the ability to use existing sequential code, which can be compiled to target multiple multiprocessors. Performance is high if the compiler is able to adequately detect and manage the parallelism implicit in the program, and if the compiler can change its management based on architectural features and overheads of the current target.

- Languages with parallelism implicit in their operational semantics form a second approach to effective portability. Applicative languages [5], [21], [24], [42] fall into this category. Applicative programs are collections of side-effect free functions and applications. Due to their independence, by definition these applications can be executed as soon as their inputs are ready. While parallelism is implicitly identified in the applicative program's description, compilers continue to be responsible for the management of data (function inputs and output), the scheduling, and sometimes the partitioning, of applications. These languages have the potential to support effective portability in much the same way as do parallelizing compilers.

- Operating systems have the potential to support effectively portable programs. An operating system can try to mask at run-time any inter-machine differences that affect performance. This masking would enable one program to efficiently execute on a set of different multiprocessors.

To use memory management as an illustration, work at
the University of Rochester [8], [11] has focused on automatic page placement strategies that hide the nonuniform memory access time characteristics of NUMA shared memory multiprocessors. The strategies assist an efficient program to be written without regard to the memory hierarchy. Suppose that a program could be ported to another multiprocessor with a different memory hierarchy. Software development cost would be low because the program is ported. Assuming the memory management unit of the target also concealed its memory access time characteristics, performance would be preserved.

• Parallel models and systems are languages and compilers or run-time systems for the expression and execution of explicitly parallel programs [10], [16], [22], [18], [26], [25], [33]. Models can hide the details of the target multiprocessor from the programmer through a virtual machine—a standardized interface. The programmer directly or indirectly maps the program to the virtual machine, and the model maps the virtual machine to the underlying multiprocessor. The separation of the program from the multiprocessor through the virtual machine allows the program to be portable. The program will perform well as long as the virtual machine can be efficiently mapped to the target.

Each of the listed approaches has advantages and limitations. For instance, parallelizing compilers and applicative languages both require significant support from the compiler in order to execute efficiently. They both present a familiar programming medium for the programmer, however. Operating systems can work in conjunction with any of the other alternatives. Minimizing the effects of false sharing and excess copying are paramount to their success. The parallel model and system approach is quite flexible in the problems it can express and in the medium of their expression. Performance and programmability depend highly on a good virtual machine interface and mapping.

Each approach also better fits certain architecture, algorithm, and programming style domains. Before presenting the approach we take to effective portability, we thus describe our research domain in more depth.

III. RESEARCH DOMAIN

Because the space of parallel problems, architectures, and programming styles is so wide, solutions specialized to certain domains are often more successful at solving the problems they address. Our research concentrates on the use of shared memory multiprocessors in solving interval and tree-structured algorithms. Concentrating on the class of shared memory architectures is of practical benefit because, for a large set of users, these are the most popular and successful means for accessing medium-scale parallelism. Similarly, concentrating on interval algorithms is of practical benefit because they occur frequently in numeric computing. The algorithm class includes a large body of numerical, array, and iterative computations used by the scientific community to solve physical and engineering problems. Tree algorithms occur in nonnumeric as well as numeric computing. For example, the symbolic community commonly employs tree-structured search spaces for reasoning. Two large groups of tree algorithms are branch-and-bound/αβ search algorithms and divide-and-conquer algorithms.

Architecture: The key feature of shared memory architectures that all processors can physically reference the same address space. In general, a shared memory multiprocessor can be characterized as a set of independent processors connected to a shared memory through an interconnection network. The shared memory can be contiguous or it can be distributed. Often when the shared memory is contiguous, the interconnection network is a single high speed bus. In contrast, when the shared memory is distributed, the interconnection network tends to be multistage.

Shared memory architectures can be characterized by their memory access times: uniform or nonuniform. Ignoring the effects of possible hardware memory caches and possible network contention, if the time to address a location in memory is the same for all processors, then that multiprocessor is a uniform memory access time multiprocessor. If the time to address a location in memory depends on the accessing processor, then that multiprocessor is a nonuniform memory access time multiprocessor. Multiprocessors with distributed shared memory often exhibit nonuniform memory access times. Frequently each processor of a NUMA is directly connected to one section of shared memory, called local shared memory, and must access the rest of shared memory through the interconnection network. Nonlocal accesses can be an order of magnitude slower than local accesses.


Algorithm Classes: Interval algorithms apply a function to disjoint subintervals of a fixed and defined continuous domain, potentially in parallel. For example, matrix multiply is a data interval algorithm. Rows of one array are multiplied by columns of the other to compute the solution array. Each row and column is a continuous interval of data elements. Collectively, the set of rows and the set of columns are also intervals of data. Intervals, interestingly, do not always have to be defined by sets of data elements. For instance, Monte Carlo integration can be solved using a spatial interval algorithm. The region whose area is to be estimated is divided into subintervals, defined by endpoints, whose regions are estimated. The key characteristic of interval algorithms is that they can be partitioned along an interval into components that can potentially be executed in parallel.

Tree algorithms are the set of algorithms whose computational structure evolves in the form of a tree. A problem is solved by breaking it into several subproblems that are recursively solved. Because the subproblems tend to be largely independent, these algorithms can often take advantage of multiple processors. Typically, the majority of data used to
solve the subproblem is reflected in the description of the subproblem.

We assume that the algorithms have some uniformity in their data reference patterns. (If this assumption does not hold, then finding efficient parallel algorithms for even a single architecture can be at best problematic.)

Programming Style: We focus on programs that assume shared memory exists and whose parallelism is explicit. As noted earlier, one major alternative—implicitly parallel programs with parallelizing compilers—has some advantages. Parallelizing compilers can use existing sequential code; they allow a programmer to write in a familiar sequential form; and they have proven successful on regular, numeric applications in restricted architecture domains. Explicit parallelism, on the other hand, can encourage the programmer to think of parallel solutions that may be qualitatively better than those formed when thinking sequentially [3], [46]. Certain classes of algorithms, such as the tree algorithms in which we are interested, currently appear to be more successfully parallelized by the programmer than by the compiler.

IV. CHAMELEON BASICS

We pursue effective portability through the approach of parallel models. This choice was principally motivated by our interest in demystifying the interactions between source and system components of an effective parallel program. The flexibility of the approach, along with its lightweight and extensible nature and its applicability to different algorithm classes, further encouraged our exploration.

In general, an abstraction represents a concept. It is defined by an interface, which is a set of functions through which the concept can be accessed. An implementation is the realization of an interface. Our approach heavily exploits the ability of one interface to be independently realized by multiple implementations.

A Chameleon program consists of a source component and a library (or system) component that interact through a set of abstract interfaces (Fig. 2). The key to the model is that these interfaces abstract activities whose most efficient implementations may differ on the basis the algorithm, multiprocessor, and even run-time values of execution. When a program is moved to another multiprocessor, its source component remains unchanged as long as implementations of the abstract interfaces are available. Consequently, software development costs are low. High performance results because different effective library implementations, chosen on the basis of each execution environment, can be automatically bound to the interfaces. The sharing and extensibility of these libraries across applications within a multiprocessor further reduces overall software development costs.

To define Chameleon’s critical abstractions, we identified activities that substantially affect the performance of parallel programs.

Data-Representation: Layout and distribution characterize the data-representation of a data structure.

Layout specifies how the elements in a data structure correspond to their positions in memory. For example, a matrix can be laid out in row or column major form, or even in blocks or diagonals. To best exploit cache prefetching, software caching, and compiler address optimizations, the layout of a data structure should match the application’s access pattern of the structure.

Distribution specifies how a data structure’s elements are dispersed amongst the memory modules of a multiprocessor. For example, a structure can be spread across several modules, it can be replicated in each module, or it can be stored entirely in one memory module. Distribution is critical for machines with distributed shared memory, since the cost of accessing a certain memory module can vary significantly among processors. Distribution in these machines generally affects the contention for and the cost of remote accesses to data.

Initial studies we conducted validated these assertions [23]. As an example, a matrix multiply program (AB=C) achieved near-linear speedup on a Sequent when A, B, and C were stored in row major order. The program achieved super-linear speedup when B was switched to column major order, due to a high cache hit rate caused by the layout decision. On a Butterfly, moving contiguously stored matrices to scattered representations (changing the distribution decision) improved performance by a factor of four. Using the best layouts for the matrices, caching the row and column pointers in local memory, and then caching the top row and column of data when it was to be multiplied, each improved performance further. While caching data reduced remote memory references, caching pointers served also to reduce contention for the contiguous pointer array.

Our experiences with data-representation are not isolated. Scientists programmers, for example, often transpose their matrices to better match access patterns and layout. Many languages and systems for nonshared memory multiprocessors include support for distributed structures [9], [20], [40]. Other empirical studies, by comparing the performance of programs whose data structures have different implementations, have also demonstrated the importance of locality of reference and of reduction of memory contention on performance [29], [41].

Dynamic Partitioning-Scheduling: Partitioning-scheduling is best described in terms of our program execution model. A program consists of a set of tasks, where a parallel task is composed of units that can be processed in parallel. When a task is to be executed it is placed in a work pool. Each
1 #include chameleon.h
2 RowwiseMatrix *AMatrix, *BMatrix;
3 ColwiseMatrix *CMatrix;

/* multiply an interval of A by an interval of B, storing result in global C */
4 void
5 multiply(RowwiseMatrix *A, int startA, int stopA, int startB, int stopB, int startC, int stopC) [...] 
6 ColwiseMatrix *B, int startB, int stopB, int startC, int stopC) [...] 
7 main() {  
8 Chore *multChore;
9 IntervalTask *task;

/* create and initialize data */
10 AMatrix = new [shared] RowwiseMatrix(100,100,ReadOnly,Freq);
11 ... 
*/
12 /*describe computation*/
13 multChore = new [shared] Chore(multiply, 1, Regular);
14 multChore->execute(1, ReadOnly, Freq, ByRow, 2, ReadOnly, Freq, ByCol);
15 task = new [shared] IntervalTask(multChore, 
16 AMatrix, 0, AMatrix->rows(),
17 BMatrix, 0, BMatrix->cols(),
18 Chunk);

/* compute */
19 task->activate();
20 task->synchronize();

/* print results */
21 CMatrix->print();
}

Lines 2,3,10 define and allocate three abstract matrices, A, B and C. Lines 8,12,13 define the chore that describes the inner loop of matrix multiplication. Lines 9,14-19 define and invoke the parallel matrix multiplication task to be partitioned and scheduled according to a chunking policy.

Fig. 3. Chameleon source program for matrix multiplication.

processor runs a single process called a worker. Workers repeatedly select and execute tasks from the pool. Several workers may simultaneously operate on a given parallel task. A worker coordinates its activities with other workers by invoking a task’s self-directing partitioning-scheduling policy. The policy guides the worker in choosing the size and identity of the task unit to execute next. For simplicity, we often call a worker executing a partitioning-scheduling policy a partitioning-scheduler.

One facet of partitioning-scheduling is the dynamic partitioning of parallel tasks, where the granularity at which a task is executed can be determined by the run-time environment. By granularity, we mean the time a worker spends executing a task before synchronizing for more work. If granularity is too fine, a worker’s synchronization overhead can dominate execution time. If granularity is too coarse, load balancing can be compromised: several workers may be idle while others operate on large final task units. Moreover, if granularity is fixed for the entire execution of a task or program, the execution cannot be responsive to changes in the number of active workers and the amount of pending work—which may harm performance.

The other facet of partitioning-scheduling is dynamic work choice. A worker can minimize its execution time by selecting a task or task unit wisely. For example, to increase the amount of work available and minimize starvation, units generating more work can be processed before terminating units.

There is a natural connection between the partitioning and scheduling activities. For example, a worker on a NUMA multiprocessor may prefer to take large portions of local work but small portions of remote work. For efficiency reasons on NUMA machines there is also a connection between data-representation and partitioning-scheduling. Because these operations are conceptually independent, the Chameleon structure strives to keep their definitions independent while allowing their implementations to interact as necessary.

An Example: Matrix Multiplication: At this point it is useful to make things more concrete. Fig. 3 presents a simple Chameleon source program that computes matrix multiplication. The program is written in the Chameleon prototype, which is built using the C++ language [47]. As we will explain, many of the C++ methods are inline for performance.

The details of this program (as shown in Figs. 4 and 5 and further explained in Section V) are outlined here to illustrate three main points about the approach.

First, data representation—how the arrays are represented in memory—is abstracted. The arrays are defined and allocated through methods of the Chameleon matrix interface (lines 2,3,10).

Second, the computation of the program is defined in terms of chores (lines 8,12,13) and tasks (lines 9,14-19). A chore describes the work of a task—in this case, the inner loop of the matrix multiply (lines 4-6). The task then identifies the interval on which to apply the work—in this case, the bounds of the matrices.

Third, because the task describes a parallel computation, a policy to partition and schedule the work must be identified. The policy is indicated in the example by the keyword, Chunk...
class RowwiseMatrix {
  public:
  RowwiseMatrix(int r, int c, DataAccess style, DataFreq afreq); // constructor
  int* ref(int i, int j); // accesses element i,j
  int row(); int col(); // return matrix dimensions
}
}
}

The matrix constructors create a logical matrix with the defined number of rows and columns. Rowwise matrices are stored for efficient rowwise accesses (i.e., their layouts optimize access by row). Columnwise matrices are stored for efficient columnwise access. Parameters to the constructor define how the matrix is globally accessed by the whole program. The data access value indicates how destructively the program will access the object. The data frequency value indicates how often the object will be accessed. These data provide information to the system to aid its efficient execution of the program.

The ref method returns the value at position i,j of the logical matrix. The row and col methods return the number of rows and columns of the logical matrix, respectively.

![Fig. 4. Partial data interfaces used for matrix multiplication.](image)

class Chore {
  public:
  Chore(PParallel worldproc, int unitscost, Behavior behavior); // constructor
  void accessFormal(int paramnum, DataAccess da, DataFreq df, DataPattern dp); // registers info about forms
}

The chore constructor creates a chore object that describes the work procedure. A work procedure is a sequence of instructions to be executed sequentially. The procedure is parameterized for variable execution time, its parameters define a partition over which to operate. A procedure's unit cost is a relative value that estimates how large a work procedure's partition must be for the execution time of the invocation to exceed the overhead incurred by the invocation. Although this value may change between multiprocessors, the source code modification required by such a change is small and consequently of low cost. In addition, it is easy to store this value as a data in a file separate from the source and read in (and potentially derived) by the system. A procedure's behavior is a value that describes the regularity in execution time of the procedure. In contrast to the unit cost, the behavior of a procedure does not typically change across architectures.

The chore's accessFormal method identifies how a formal object parameter of its work procedure will be accessed by the procedure. The data access value indicates how destructively the procedure will access the object. The data pattern value indicates the access pattern the procedure will make on the object. The data frequency value is an indicator of how often the object will be accessed during a particular invocation.

class IntervalTask {
  public:
  IntervalTask(); // constructor, one dimension two intervals
  Chore *c, // work procedure
  Data *d1, // first object
  int e1, int stp1, // partitioning interval
  Data *d2, // second object
  int e2, int stp2, // partitioning interval
  FType parttype, // partitioning strategy
  int numworkers, // max num processing resources
  ...);

  void activate(); // places task in work pool
  void synchronize(); // returns when task is done
}

The constructor of a task creates a task object. An interval task is identified by: a chore (its work procedure), one or more data objects and their boundaries (intervals over which the work procedure is to be applied); a partitioning/scheduling strategy identifier (identifying a strategy to use to partition and schedule the work defined by the intervals); (optional) maximum number of workers to use.

The activate method starts the asynchronous execution of the task. The synchronize method suspends the caller from executing its current stream of statements and starts it executing the task along with the other workers. When the task is finished, the caller returns. By immediately following a task's call with a call to synchronize, a task becomes a synchronous entity.

![Fig. 5. Partial control interfaces used for matrix multiplication.](image)

(line 17). The implementation of the policy, like the data representations, is abstracted from the source program.

This hiding of information highlights the central purpose of the example. As motivated earlier, different representations of data structures and partitioning scheduling policies can be necessary for high performance on different multiprocessors.
Abstracing these operations enables their representations to change independently of the source. It enables the source program to port without modification.

V. CHAMELEON MODEL AND SYSTEM

The Chameleon model is analogous to a programming language. It defines the primitives in which the applications programmer writes a parallel program. The primitives central to our work are data and control interfaces that can be added to a sequential language. The Chameleon system is analogous to a compiler for the model. It includes a set of implementations of the interfaces together with run-time support for their coordination.

In practice, system programmers will develop their instances of the Chameleon model and system in computational areas of primary use by their application writers. For example, numerical analysts will require rich support of array data structures and interval control strategies. The initial model and system will then be extended—by the systems programmer or, if the extension is a customization, by the application writer—in response to changes in the operating environment.

The general set of Chameleon abstractions includes data-structures, chores, tasks, and partitioning-scheduling policies (ps-policies). Before we explain how these abstractions are tangibly realized, we identify several properties suggested for the sequential language on which a Chameleon package is built.

**Base Language:** We exploit the following four mechanisms, which are frequently found in object-oriented languages: 1) A mechanism for information hiding [35], necessary to separate the interface and implementations of an abstract entity. 2) A mechanism that allows a representation to be dynamically bound to an interface. Most languages, including C++, our prototype language, do not provide such a mechanism. We have solved this problem, as we describe next, by creating separate objects for the interface and for the representation itself and then by linking these objects dynamically. 3) Inheritance of specifications, which assists in realizing the specialization structure of the programming model and its extensions [38]. 4) Implementation (or code) inheritance, to help reduce software costs.

We outline the Chameleon abstractions, focusing on the interval algorithm domain, when necessary, for clarity. Section VII discusses aspects of Chameleon that are specific to tree algorithms. It will become clear that each abstraction is actually structured as a growing family of interfaces and implementations.

**Data Structures:** A Chameleon source program must allocate and access each of its data structures through a fixed interface to that structure. An array interface, for example, includes methods for creation and destruction; it includes methods to access its elements and to query about its size. The interface separates the logical data structure from its physical representation in memory. Consequently, a program interacts with a logical entity that is less complex than its tuned physical representation, and the physical representation can change without affecting the rest of the program.

There are a variety of ways in which a representation can be chosen for, or bound to, an abstract data structure. One approach is for the programmer to state a binding in the source program. The advantage of the static choice is that the choice is explicit in the program text, and the programmer can consciously experiment with different representations. The disadvantage is that the program must be recompiled and relinked when the binding is changed. Another approach is for every data abstraction to include a routine to select a representation. Run-time values, such as the size of the structure, can then influence the representation chosen. As an example of the latter, on a distributed memory machine a replicated representation could be chosen for a small read-only array, yet a distributed representation chosen for a large read-only array. The advantage of the dynamic choice is that there is added flexibility at run-time. The disadvantage is the run-time cost of selecting the representation, and the potential cost of having to access it indirectly.

We use the dynamic approach without precluding a variant of the static approach. Arrays are the most common data structure used by interval algorithms. As we explain in Fig. 7, while Chameleon selects distribution, selecting layout is left to the application writer. The layout decision is difficult to make automatically; moreover, binding it early enables compilers to potentially optimize memory addressing. Unlike distribution, generally this choice is based on the characteristics of the algorithm and does not change between machines.

To permit different representations for different abstract arrays of a program, and to permit the dynamic selection of a representation, two objects are used to represent an array: an interface object and a representation object. When a source program creates an array through a call to an abstract array constructor, an object representing an interface to that array is created.

If the constructor is passed a representation identifier, it then binds a representation object of that type. Although our current prototype does not support it because of limitations of the compiler, these statically selected bindings could occur at compile-time. Logically, this option is equivalent to the static approach above.

If the constructor is to dynamically select a representation, it then invokes the select method provided by the array abstraction. This method selects and binds a representation object (Fig. 6) using information provided to the abstract array constructor, as well as information about the memory hierarchy of the target machine. Information provided to the constructor includes the size, overall access frequency and access type of the array (Fig. 6). While defaults exist for the access hints to the selection method, the supply of more accurate information can improve the performance of the program.

With both static and dynamic selections, subsequent calls made by the program to the methods of the interface object are handled as calls to the methods of the representation object. The critical point is that the source program interacts with the array only through the interface object. Thus, the representation object can change without affecting the source. (By careful definition, primarily by sharing the same interface,
critical interface and representation routines can be inlined without disabling the dynamic choice of representation. The matrix ref method of the prototype, for instance, is inlined yet the representation dynamically bound, as all of the prototypes' representations share the same indexing protocol. Section VIII-C explores the cost of inlining without the compromise of dynamic choice.)

All together then, an abstract array has a constructor that binds a representation object either directly or by invoking a select method, a destructor that releases the memory first of the bound representation and then of itself, and access operators that reference the associated access operators of the representation object. A representation has a constructor that allocates the array's memory in a particular layout and distribution, a destructor that releases this memory, and access operators (associated with those of the abstract array) that access the memory structure.

Fig. 7 illustrates how a family of array interfaces and representations can be structured as hierarchical libraries in Chameleon. The interfaces form part of the Chameleon model and are available to the application writer for use in a set of parallel programs. The implementations and their coordination form part of the Chameleon system.

Chores: Encapsulating the partitioning-scheduling activity requires several abstractions. To allow the ps-policy to control the program's granularity, parallel work must be dynamically decomposable. The chore is the Chameleon abstraction that describes a dynamically sizable unit of computation. The task, as we will soon detail, specifies the bounds over which a particular chore is to be applied; thus, a task specifies a decomposable composite of parallel work.

The principal component of a chore is its work procedure, which defines a sequence of instructions to be executed sequentially. A work procedure has parameters that control its length of execution. Hence, by calling the work procedure with different actual parameters, partitioning-schedulers (workers executing ps-policies) can vary the granularity at which the program is run. A work procedure, for example, might contain a loop and be parameterized by the bounds of the loop.

Because of the interaction between work procedures and ps-policies, the parameter list of a work procedure must conform to a signature known to the ps-policy. The signature of this simple loop example, for instance, could be (Vector* vec, int start, int stop, int globalstart), where start and stop indicate the interval of vec to be operated on by the procedure. The globalstart integer denotes the global name of the start index. This value is necessary since, as we will soon explain, the start and stop indexes may refer to a cached object, yet the name of the original section is needed by the computation.

Tasks: A Chameleon task describes parallel work. Specifically, it is a decomposable composite of work that can be executed in parallel at various granularities and according to a preferred schedule of execution.

When a task is created it binds to itself a ps-policy object instantiated with task-specific information. This information includes a chore (the task work descriptor) and the bounds over which the chore is to operate (the task scope). On activation of the task, each cooperating worker independently executes the partition operation defined by the ps-policy object. This routine directs workers in the partitioning and scheduling of the task's parallel work.

A task contains a constructor, an activate method, and a synchronize method.

The constructor creates a task, creates an associated ps-policy object, and associates the ps-policy object with the task. When the ps-policy object is predefined, the constructor need only create the task and bind the provided ps-policy. As an example, a task describing an interval algorithm is constructed by passing the constructor either: 1) a chore, the endpoints of the interval(s) to be partitioned, data objects corresponding to the partitioning intervals, and a partitioning-scheduling keyword, such as Chunk (uniform partitioning) or Dynamic (partitioning based on the amount of pending work), used to identify the ps-policy of choice; or 2) a ps-policy object instantiated with this information. To change the ps-policy used to execute a task in the former case, the keyword is modified; to make the change in the latter case, a new ps-policy object is supplied.

The activate method of the task starts its asynchronous execution. The method places the task in the system workpool for workers to find and execute. For synchronous execution of a task, the activate call can be immediately followed by a call to synchronize, which is defined to complete when the work of the task is complete.

Fig. 5 provides examples of a Chore and Task interface of Chameleon.

PS-Policy: The function of a ps-policy is to guide workers in their parallel execution of a task. Typically, a Chameleon source program interacts with a ps-policy only indirectly, by
passing a keyword identifier to a task. The Chameleon system then creates the suggested policy object. The application program can create customized policies, however, if the general library policies are not adequate.

A ps-policy provides several key methods.

The constructor of a ps-policy object instantiates the object with its task-specific information. For interval algorithms, this information includes a chore, the endpoints of the interval(s) to be partitioned, and the data objects corresponding to the partitioning intervals.

The partition method is called by workers to self-direct their scheduling and partitioning of the work composite. Specifically, it defines the parameters with which the workers invoke the chore work procedure. Because the workers synchronize on the common synchronization data included in the ps-policy object, workers do not interfere with the work that another has claimed. The outermost loop of the partition method is shown in Fig. 8.

To manage the granularity of a task’s execution, the ps-policy includes a method determinechunk that identifies a partition size. When a task is to be executed with fixed granularity, this method is called in the partition routine only once. For variable granularity, it is called each time a worker synchronizes. To make its granularity decision, determinechunk can refer to values such as the size of the initial interval, the amount of pending work, the unit cost of the procedure, the number of active workers, and the overheads of the target. Consequently, the granularity at which a program is executed can be easily modified by changing the determinechunk method for a given policy.

Similar to the data structures, Chameleon hierarchically structures its library ps-policies. Fig. 9 displays a segment of our prototype library. The hierarchies aid in both the organization and extension of the set of available activities, which in turn increases software reuse.

To facilitate portable performance, the ps-policy interface hierarchy can be paired with a different hierarchy of implementations on each machine. Here, the binding of implementations to interfaces occurs at load time. Multiple implementation hierarchies allow each ps-policy implementation to be tailored to its target. A simple policy can thus be used on a UMA machine but a more complex policy on a NUMA machine, where it is needed for efficiency. The common ps-policy interface ensures the source program is not affected by which implementation is bound.

VI. INTERACTION BETWEEN THE ABSTRACTIONS

Data representation and partitioning scheduling activities are often related, especially on distributed shared memory multiprocessors. Two common approaches, data-directed scheduling and software caching, can be employed to reduce the cost of accessing remote data. Use of these techniques, while not necessary for strict portability, is necessary for effective portability [1].

Data-Directed Scheduling: Using data-directed scheduling, whenever possible a worker chooses task units with locally resident data. The technique exploits data locality. Because local data are faster to access, the worker can complete these units more quickly than it can complete units whose data are remote. Knowledge of the data associated with each task, along with the location of that data (the data-representation), is necessary to determine which workers prefer which task units.

To assist with data-directed scheduling on NUMA machines, each ps-policy includes a preferred method. The method identifies, for each worker, a set of work that will maximize its local and minimize its remote memory references. The work sets are disjoint and include all possible work of the task. Together, they serve as a list of pending work that is updated by workers as they execute. Because workers can prefer to take large portions of preferred work but small portions of nonpreferred work, the partitioning and scheduling activities are connected.
Knowledge of how the data structures of a task are represented in memory is required to create the preferred work sets. Each abstract data structure thus includes a method, mirrored in each representation object, that returns a description of its location. When a ps-policy object is created, its preferred method is executed to develop the worker preference sets. The method queries the tasks' data structures about their location and uses this information in its development. The preferred method must, of course, execute quickly. Consequently, the preference methods we have developed refer only to the distribution information associated with those data structures passed in the task/ps-policy constructor—those objects that correspond to a partitioning interval. Fig. 10 outlines the construction of the preferred sets for a simple interval task.

Notably, although the operation of data-directed scheduling requires that the implementations of the data-representations and partitioning-scheduling abstractions cooperate, the abstractions themselves are independent.

Software Parameter Caching: When the data of a task are located in several memory modules, or when no pending task has data local to the worker, data-directed scheduling is not sufficient for reducing all remote memory reference costs. To augment data-directed scheduling, a worker can, under certain conditions, use software caching techniques to make a local copy of the remote data on which it operates. Specifically, software caching must be safe and worthwhile.

Caching is safe if making a copy of the data does not jeopardize the correctness of the program. For example, data that are only read are always safe to cache. Caching is worthwhile if it would provide overall performance improvement. In other words, the cost of copying and accessing the cached data must be compared to the cost of accessing the data remotely. On the Butterfly, for example, it is often more efficient to copy remote data into local memory and then access it there. Because the block transfer operation of the Butterfly is fast, time savings can result even when the data are accessed only once.

Many existing parallel programs interleave code for software caching with the algorithmic code of the program. Our goal was to remove these tailoring details from the source program, leaving the logical function of the source program clear. Encapsulating the caching statements in the data and ps-policy achieves this goal. Additionally, the encapsulation
Fig. 11. Possible work procedure parameter bindings. The upper diagram illustrates an invocation of the chiple work procedure with an actual data object. The lower diagram illustrates an invocation with a cached object. Because caching statements are encapsulated in, and directed by, the $ps$-policy, the source program is unaware as to whether it is operating on an actual or cached object. Moreover, the function of the program is unobscured by these tailoring details.

1) enables a data-object itself, to decide how it is cached,
2) enables the granularity of caching to be influenced by the semantics of the policy, 3) enables the data-directed schedule to influence the presence or absence of caching, 4) and collects performance-related functions—functions whose most efficient implementation may differ between targets—together.

Consider the general signature of an interval work procedure operating on one interval: (Object* array, int start, int stop, int gstart). Essentially, the partitioning-scheduler first operates on its preferred (local) work, passing pointers to the uncached object as an argument. It then operates on nonpreferred (remote) work. The partitioning-scheduler caches the remote data and passes a pointer to the cache object as an argument. Note that the work procedure, the code of the source program, need not be aware of whether its formal parameters are original or cached objects (Fig. 11).

To foster independence between the data representation and partitioning-scheduling decisions, each abstract data structure (and the associated representation) includes a method to clone a full or partial copy of itself in the local memory of the caller. This method can be used by partitioning-schedulers to construct the cached object. The method is defined to succeed only if caching is safe and worthwhile. If the method fails, the original object is used by the partitioning-scheduler.

It is important to note that Chameleon’s primary goal is not to provide novel data-representation or partitioning-scheduling schemes. Instead, the Chameleon structure is intended to enable parallel programmers to encapsulate and reuse schemes that have been developed in the course of researching and writing parallel programs.

VII. APPLYING CHAMELEON TO TREE STRUCTURED ALGORITHMS

Our research focuses on applying Chameleon to interval and tree algorithms. While many interfaces and much of the source-system program structure described in the previous sections apply to both of these algorithm classes, this section collects the specializations particular to trees. In general, Chameleon can be applied to most algorithm classes that describe their computations as decomposable composites of parallel work.

Data Structures: Nodes and, at a higher level, trees, are the data structures specific to tree-structured algorithms. Unlike arrays, the attributes of computational nodes tend to be very specific to the computation of the program. For example, a node for an adaptive quadrature algorithm—representing an interval to integrate—might have fields for the interval endpoints, midpoint, and the integral value. A node for a knapsack algorithm, representing a state of the search space, might have fields to denote the item, the current value of the pack, and the capacity left in the pack. Given the customized nature of nodes, it is easiest for each source program to extend a base node interface with its specialization, and an extensive library is not provided. The basic Chameleon node is a placeholder, with a trivial constructor and destructor.

Consider a sample extension, denoting a binary tree node, includes: a constructor, which optionally stores a value for the node, together with information about the type and frequency of access; methods to access the value, the parent, and the child and; a trivial destructor.

While the application programmer specifies the representation (the contents) of a tree node at program development time, the system binds the computational tree to a distribution at runtime (Fig. 12). Again, Chameleon supports a separation between a logical object (the tree) and how it is represented in memory.

More specifically, the dynamic, node-by-node growth of the trees is efficiently supported by careful definition and use of the system’s memory allocator. Chameleon’s memory allocator, new, can be parameterized with one of several pragmas, including shared and local shared. Most data is allocated as shared data that are to be generally accessed by all workers. The system stores this data in a random memory module. For tree computations, it is important to allocate nodes as local shared data that are principally accessed by the allocating worker. This causes the system to allocate the node in a memory module close to that worker. Because workers are close to different memories on distributed memory machines, the computational tree is automatically distributed as it is created. (The shared hint could also be used to create a distributed tree. However, true work procedures and $ps$-policies typically direct workers to work first on nodes that they allocate. Thus, the local placement induced by the local shared directive simply causes all nodes, and consequently, the tree, to be placed in the contiguous shared memory.

PS-Policies: A chore of a tree algorithm differs from a

---

4This ability is similar to that provided by Munin [27], a run-time system that employs object-specific memory management policies. The ability to have adaptive memory management policies is important because different memory management policies are best suited to different algorithms [28].
chore of an interval algorithm only by its work procedure. A work procedure evolving a tree computation has the signature:
(Node* topnode, Task* caller, int depth, int argc, char* argv). The procedure expands a node to some depth or height cutoff that is identified by the depth integer parameter and can return some nodes to the calling task. The final two parameters denote the procedure's other arguments. By calling the procedure with a different depth or height cutoff, the partitioning-scheduler can vary the execution granularity of the program. Again, where the depth hint could be omitted by the programmer, its provision typically improves performance.

A tree task, on the other hand, differs more significantly than its interval counterpart. Instead of intervals defining the bounds of computation, the constructor of a tree task requires a starting node—the root of the computational tree to be expanded and evaluated in parallel. Because tree tasks can grow during their execution, they also include methods for enqueuing work. The work procedure of a task can add a node or list of nodes to that task for evaluation using the methods addnode(Node* n), and addnodes(NodeList* l), respectively. Usually a work procedure expands its given node to a subtree of the specified depth (or height). The procedure then returns the leaves of the subtree to the task to be explored further.

In contrast to interval partitioning policies, which partition static intervals into subintervals of a certain length, tree partitioning policies partition growing trees into subtrees of a certain height or depth [50]. The distinct tree and interval branches of the ps-policy interface hierarchy shown in Fig. 9 reflect this difference. Distinct implementation hierarchies can exist on each target, since the implementations are customized to their target. Because of the role the memory allocator plays in effectively realizing the tree abstractions, there is often significant inter-architecture commonality between tree implementation hierarchies.

Despite the algorithmic differences the structure of a Chameleon tree ps-policy object parallels that of an interval ps-policy object. The central methods used by the system for achieving performance are partition and caledepth.

The partition method, exported by every ps-policy, encodes the actual partitioning-scheduling algorithm. The caledepth method parallels the determined method of interval ps-policies. It assists with the partitioning decisions by identifying a measure of granularity.

The Memory Hierarchy: In order to minimize memory latency, tree ps-policy NUMA implementations incorporate data-directed scheduling through their list organizations. Essentially, each worker (or possibly set of workers) has a personal list of computational nodes to process. Workers process nodes of their own lists unless the list is empty. Then they process nodes of another's list. When directed by a work procedure, a worker can add some nodes to the task. As workers allocate nodes in their local memory and add them to the task by adding them to their own work list, the new computation is implicitly data-scheduled.

The per-worker node lists are also employed by UMA ps-policy implementations. Their tangential benefit is the ability to reduce contention for the scheduling structure.

Although a partitioning-scheduler generally prefers work from its own local work list, it may process a node from another worker's list to balance the computational load. If the node were remotely located, as it may be on a distributed memory machine, the node could be cached in local memory to improve performance. Instead of passing a remote computational node to its work procedure, the partitioning-scheduler passes its cached copy. Note that caching may only be beneficial if the node were frequently referenced by an invocation of the work procedure.
Similar to arrays, nodes have clone methods associated with them to assist in the caching decision and operation. As a node is defined by the programmer, the clone method must also be defined. A template method found in the system node definition can often be replicated in the subclass node and used for this definition.

An Example: Adaptive Quadrature: Fig. 13 presents a segment of a Chameleon source program for adaptive quadrature using Simpson's rule. Nodes represent regions to integrate. If the approximation of a region is outside a specified tolerance, the region is divided into two smaller regions (child nodes) for future approximation. If the approximation is within the tolerance, the approximation is posted, possibly recursively, in the parent node.

Although the example is incomplete—the work procedure and node and ps-policy representations are not given—it illustrates use of the tree abstractions just defined. LIFOFOFIXed indicates a scheduling strategy where workers process nodes on their queue in a LIFO manner and nodes on queues of others in a FIFO fashion. The partitioning is based on a depth cutoff determined once and used repeatedly.

VIII. EVALUATION OF CHAMELEON

Let us quickly summarize the material that has been presented thus far. The problem is to find a better balance in the tradeoff between software development costs and software performance than is commonly found in parallel programming. Feasible approaches evolve around the notion of an effectively portable program, where an effectively portable program performs well on each target machine while incurring low software development costs.

Chameleon, our approach, is an extensible programming model and system based on abstractions critical to performance—abstractions whose most effective implementations can differ with the execution environment. This section describes how Chameleon programs, due to their source-system component separation, ease maintenance and experimentation while sacrificing little performance. It demonstrates a major advantage of Chameleon programs, their effective portability.

A. Maintenance

Many explicitly parallel programs intertwine statements concerning data-representation and partitioning-scheduling with statements implementing the program's basic algorithm. An example of this interleaving is shown in Fig. 14. In general, the intertwined approach makes it difficult to determine if a source statement is part of the basic algorithm, the data-representation, or the partitioning-scheduling scheme. In other words, the approach can obscure a program's true function.

Lack of clarity makes even basic maintenance and understanding of the program more difficult, because it is hard to change or understand a single aspect in isolation. Collecting the tailored partitioning-scheduling and data-representation statements in the library component of a Chameleon program clarifies the function of the source component. Consequently, the maintenance programmer can more easily determine the purpose of the program (readability is eased), debug the algorithmic operation of the program (correctness debugging is eased), and modify the capabilities of the program (evolution is eased).
This function, for use on a Butterfly multiprocessor, is called in parallel to execute the inner loop of a matrix multiply algorithm. a and c are global, distributed matrices whose elements are stored in row major order. b is a global, distributed matrix whose elements are stored in column major order. Given row and column identifiers, the function computes c[row,column] by calculating the dot product of a[row, column] and b[column, ].

```c
int * a, * b, * c;
int rows_a, cols_a, rows_b, cols_b;

double multiply(int row, int col)
{
    int i, sum, *rra, *rrb;
    int transa, transb;
    // ensure data is in local (fast) memory
    if (Where.Is(a[row]) != Proc.Node)
    {
        rra = (int *) malloc (cols_a * sizeof(int));
        btransfer(a[row], rra, cols_a*sizeof(int));
        transa=1; }
    else {
        rra = a[row];
        transa=0; }
    if (Where.Is(b[col]) != Proc.Node)
    {
        rrb = (int *) malloc (cols_b * sizeof(int));
        btransfer(b[col], rrb, cols_b*sizeof(int));
        transb=1; }
    else {
        rrb = b[col];
        transb=0; }

    // multiply
    for (i=0; i<cols_b; ++i)
    {
        sum = *(rra) * *(rrb);
        c[row][col] = sum;
    }
    // release temporary storage if necessary
    if (transa) free(rra);
    if (transb) free(rrb);
}
```

Fig. 14. Inner loop of a matrix multiply program.

The intertwining of data-representation and partitioning-scheduling decisions is visible in the statements that cache the topic row and column in local memory (knowledge of data-representation) and the operation of the function on a single row and column (knowledge of partitioning size).

B. Experimentation

Lack of clarity also makes it hard to improve performance on a given architecture, because this typically requires experimentation with data-representation and partitioning-scheduling schemes, and these aspects may be confusingly interleaved.

The Chameleon abstractions isolate performance critical decisions. The identification of these decisions, together with their independence from the rest of the code, eases both performance debugging and experimentation.

Performance debugging, locating and recoding bottlenecks, is eased because the abstractions identify two critical activities that affect performance. Isolation of these implementations enables the functions to be directly tailored and easily monitored. Moreover, specific methods used to determine grain size, preference sets, and to select data distribution, further ease performance tuning.

Experimentation is enhanced because the source program, being written in terms of interfaces, is unaffected by changes in implementations.

C. Effective Portability

To evaluate the effectiveness of our approach to limiting the software development-performance tradeoff, we developed a concrete implementation of Chameleon on a Sequent Symmetry 81 and a BBN Butterfly Model GP1000 multiprocessor. The Chameleon prototype is written in C++ and contains approximately 2000 executable lines of code.5

The Sequent is configured with 20 Intel 80386 processors. Each processor has a 64 Kbyte cache; the write-back strategy is copy-back. All processors access a 32 Mbyte memory through a shared bus. The Sequent is a UMA machine as the time to address a location in memory is the same for all processors. The Butterfly is configured with 32 processing nodes connected by an omega interconnection network. Each Butterfly node contains a MC68020 processor, 4 Mbytes of

5An executable line of code is a line of code that contains a programming language verb (i.e., +, -, for).
shared memory, and a node controller. The Butterfly is a NUMA machine as local references are substantially less expensive than remote references. The local/remote memory access ratio on our machine is 1:12. Remote references require communication across the network between node controllers. The operating system on the Sequent is DYNIX Version 3.0.12. On the Butterfly, the operating system is Mach Version 2.0.1. The prototype uses several routines from the parallel libraries provided with the machines [34], [48] to create heavyweight worker processes and access spinlocks and shared memory.

The Applications: We examine the results of four representative applications. Three applications are interval algorithms. The first, matrix multiply, computes the product $C$ of two matrices $A$ and $B$. We applied the programs to dense matrices of size $(200 \times 400)$ and $(400 \times 300)$. This straightforward computation is interesting because it is a component of many numerical applications and has the potential to speed up linearly in the number of processors. The second application takes two $n \times n$ matrices, $A$ and $Q$, and computes the congruence transformation $C = QAQ^T$. This computation is often a component of larger numerical applications and can be executed in two phases: one, calculate $B = QAQ^T$; two, calculate $C = QBX^T$. The programs were applied to two $(500 \times 500)$ dense matrices. The third application, Gaussian elimination, is a technique for solving a set of linear equations. To solve $Ax = b$ using Gaussian elimination without pivoting, $A$ is put in upper triangular form with $b$ reflecting the reduction operations. $x$ is found by backsubstitution. While the triangulization phase can be parallelized, the backsubstitution phase is generally executed sequentially. The Gaussian elimination programs were applied to a set of 500 linear equations, and the triangulization phase was timed.

The fourth application is a tree algorithm that approximates the integral of a function using the adaptive quadrature method with Simpson's rule. The algorithm was outlined at the end of Section VII. The programs were invoked to approximate the integral of a trigonometric function.

Performance: We report speedups, where the speedup of a parallel program is defined as the time to execute a sequential solution to the problem divided by the time to execute the parallel program. The timings are based on dedicated processors. They are for the central computation of each application and do not include initialization and completion operations, such as starting and releasing heavyweight processes; these costs would normally be amortized over the cost of a large application, of which each studied program would be a small part. On the Butterfly, only the memory modules in the nodes of the participating processors are used to store program data.

Performance is evaluated by comparing the speedups of the Chameleon programs on each machine to the speedups of fine-tuned (unabstracted) programs on each machine. Figs. 16 and 17 summarize our measurements. The "Sequential" programs are written in C++ and contain no statements for parallelism. The "Chameleon" programs are written using the interfaces provided by the Chameleon prototype. These programs are executed on each multiprocessor using the prototype libraries of that machine. The "Ported(X)", "Chameleon", and "Fine-tuned" programs are written using the interfaces provided by the Chameleon prototype, and executed with prototype libraries associated with machine $X$. The "Ported(X)", "Chameleon", and "Fine-tuned" programs are written in C++. On the Butterfly these programs call the Uniform System [48] for their parallel functions; on the Sequent they call the Sequo parallel programming library [34].

The initial results of Chameleon programs reminded us that abstraction, particularly at a fine grain, is not free. Most noticeably, the indirection caused by making each reference to an array element a procedure call made a significant impact on run-time. Fortunately, by inlining these procedure calls we were able to greatly reduce the cost of the references; the numbers of Fig. 17 reflects this optimization. Inlining the call to the array representation object's ref method, as well as the call to the abstract object's ref method, reduces the cost of the abstraction to an extra load. There is a load for the start of the representation object plus the normal load for the actual array element. In a tight loop, our C++ compiler moves the first load outside, incurring its cost once.

Collecting the Chameleon results, on the Sequent the Chameleon programs are within 1% of the corresponding fine-tuned speedups. On the Butterfly, the speedups of the

---

Fig. 15. Inner loop of a Chameleon matrix multiply program.
Chameleon programs are within 8% of the fine-tuned. These results indicate that Chameleon programs achieve acceptably high performance on each target multiprocessor.

It is interesting to also examine how far the Chameleon programs improve on the performance of the Ported(V) programs. These programs approximate the performance of strictly portable programs—programs that can execute on a new architecture, but that do not adjust their operation to better adapt to that new architecture. Clearly for the Butterfly, selecting the more appropriate implementations for the data-representation and partitioning-scheduling activities significantly aids performance. The poor results of the Ported(Seq) program on the Butterfly reflect the cost of remote references and memory contention incurred by their contiguous data structures and naive partitioning-scheduling. For the Sequent, the performance improvement over the Ported(Bfly) programs was not as significant. An exception is the Gaussian elimination application, where the sophisticated ps-policy used for the Butterfly only caused unnecessary overhead when used on the Sequent. We anticipate other such exceptions to be found with communication intensive algorithms and when porting between distinct distributed memory architectures. Here, different distributed memory and communication structures can require different representations to combat latencies.

Software Development Costs: Software development costs are more complicated to assess. Qualitatively, we are satisfied that the structure of Chameleon programs leads to software development costs that are in line with those for strictly portable programs. To confirm this observation, we made some quantitative measurements of program size and structure.

In terms of size, we counted the number of executable lines in the programs. We separated the counts into lines in the source component—these are written by the application programmer and tend to be application specific—and lines in the library component—these are written by the system builder for use in a set of applications. The sequential program was used as a conservative estimate of a strictly ported program. Fig. 18 diagrams the results.

From the counts we draw two careful conclusions. Examine the nonreusable entry in the table: If libraries exist, then the size of the Chameleon program set is close to the size of the strictly ported program set, and is much smaller than the sum of all the fine-tuned programs. Examine the total line-count entry: Although the total lines of Chameleon code is close to that of the fine-tuned code, 60% of the Chameleon lines are reusable while only 30% of fine-tuned lines are reusable. The development cost of this small Chameleon set appears dominated by the cost of developing its libraries; once the libraries exist, the marginal cost of developing a Chameleon program will be lower. As a concrete example of the latter, the Matrix Multiply application incurred 64 nonreusable lines and 447 reusable lines. The reusable lines were due to its creation of matrix and interval ps-policy library implementations. In contrast, the Gaussian Elimination application, which shares libraries with Matrix Multiply, incurred only with 91 nonreusable lines, total.
<table>
<thead>
<tr>
<th>Version Set</th>
<th>Non-Reusable (Source)</th>
<th>Reusable (Library)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>447</td>
<td>-</td>
<td>447</td>
</tr>
<tr>
<td>Chameleon</td>
<td>665</td>
<td>1088 (40%)</td>
<td>1753</td>
</tr>
<tr>
<td>Fine-Tuned</td>
<td>1265</td>
<td>571 (20%)</td>
<td>1836</td>
</tr>
</tbody>
</table>

Fig. 18. Executable lines of code (sum of five applications).

In sum, the line-count figures indicate that as the number of Chameleon client programs grow, the cost of the libraries can become a small fraction of the total cost of the program set. Moreover, if the Chameleon libraries exist, then the cost of developing a Chameleon program is arguably close to the cost of developing a strictly portable program. (We assumed strictly portable libraries exist without cost for our analysis.)

Such quantitative measurements of software development costs are known to be imprecise. However, together with experiences noted in Sections VIII-A and VIII-B, they appear to confirm our belief that the Chameleon structure helps keep software development costs comparable with respect to strictly portable programs, especially when considered across a growing collection of effectively portable programs over time.

So, overall, while the performance results indicate that the performance of Chameleon programs is high, the software development costs results indicate that the cost of Chameleon programs is low. In other words, the results demonstrate that Chameleon supports effectively portable parallel programs.

IX. RELATED WORK

The goal of Chameleon is to decrease the tension between the software development cost and performance of parallel programs. This section investigates how ten representative parallel models also address this goal. The languages are classified by the memory abstraction they present to the application programmer: shared memory, nonshared memory, or mixed memory—both shared and nonshared. We look at the shared memory models in most detail, as they reflect the same memory abstraction as Chameleon.

Shared Memory Programming Models: The Argonne tools [33] are a set of constructs (macros) for parallel programming. The constructs hide operating system specific functions, such as locks, process creation, and monitors, from the source program, so that the source program is portable. Unlike Chameleon, the tools do not include abstractions for data structures. System implementations of the constructs also do not address the memory hierarchy of the target.

The Force [25] is another package for portable programming. The language is based on dynamic and statically scheduled parallel loops. Similar to the Argonne tools and unlike Chameleon, the package does not address the memory hierarchy of the target.

While the previous two languages can be thought of as languages for strictly portable programs (providing effective portability between machines of one specific architecture, but requiring (possibly substantial) source tailoring for high performance on different architectures), the next three languages seek to address performance across several different architectures.

Linda [18] is a parallel model and system developed at Yale University. Linda presents a shared memory abstraction called a tuple space to the programmer. Processes communicate by asynchronously moving data (often called tuples) in and out of the tuple space. To foster effective portability, the system represents its tuple space differently on different targets. The implementation can cause elements of the space to be distributed and/or replicated. Linda’s tuple space thus shares some characteristics of Chameleon’s data structures. The tuple space can support more general memory management, such as data migration, than can Chameleon. However, unlike Chameleon, the granularity of data distribution is based on tuple size and thus controlled by the application programmer.

Par [10], designed and undergoing implementation by Coffin at the University of Waterloo, is an object-oriented language based on SR [4] for expressing architecture-independent parallel programs. Like Chameleon, a goal of Par is to enable the purpose of the source program to be clear. An application programmer enters a source program without regard to an architecture. Then, to tailor the program for high performance on a given machine, the programmer adds scheduler and mapping (data representation) annotations. Effective porting requires only changing the annotations as necessary. The decisions of representation made often dynamically and/or implicitly by the system in Chameleon are thus made explicitly by the programmer in Par. While the decisions can interact in Chameleon, they appear to remain disjoint in Par.

Recent work by Crowl and LeBlanc of Rochester has generalized the notion of Chameleon’s ps-policy abstractions to all control constructs of a language [12], [13]. In the Matroska model, a control construct can have multiple implementations. The programmer annotates each construct used in a program, to indicate which implementation should be bound at compile-time. As an example, forall _DIVIDED defines the for as a parallel divide-and-conquer loop, and forall _GROUPED defines it as a loop that executes by chunks in parallel. For a distributed memory machine, a forall _MODULAR can be used to pair control with data. This is similar to the data-directed scheduling of Chameleon. Matroska’s set of control constructs is extensible, allowing additions and customizations as necessary. The compiler packages the body of work to be controlled (i.e., the loop body) in Matroska. This differs from Chameleon, where the work body is packed explicitly in the chore definition. Although support for data abstractions is sketched in Matroska, the bulk of the research consists of support for control abstractions.

Nonshared Memory Programming Models: Although shared memory programs—the focus of this work—cannot be expressed directly in nonshared memory models, it is interesting to investigate how several of these models address effective portability.

Portability (and case of programming) is encouraged in Poker [45], Dino [40], and Ensembles [20] by an intermediary, an abstract machine, to which the application programmer maps the program. The system maps the abstract machine to the target machine. Dino and Ensembles address effective portability by enabling the code and data structures that sit on the abstract machine to be parameterized by the size
of the problem and the size of the target. Consequently, the granularity of computation and distribution scale automatically, fostering high performance on different platforms. Ensembles and Poker, in fairly modularized ways, enable the communication graph of the abstract machine (and thus, of the program) to adapt to that of the target. Well matched graphs foster lower routing costs.

**Mixed Memory Programming Models:** Mixed memory models, including Pisces2 [37], VVMP [16], and Chare [26] provide both shared and nonshared views of memory to the programmer. Generally, processes communicate by passing messages, although there also exists some portion of globally addressable data. The mixed models address effective portability as a mixture of the approaches used by shared memory models (different implementations of certain interfaces, depending on the target, VVMP and Chare) and those used by nonshared memory models (abstract machine mapping, Pisces2). Yet to be demonstrated is the high performance of the shared memory primitives on nonshared memory architectures. The models encourage minimal use of these structures for effective portable performance, as they are difficult to implement efficiently on all machines.

**X. Conclusion**

We have developed an abstraction-based approach to the problem of balancing software development costs and performance in parallel programming. Although Chameleon is successful in practice, the programming structure it defines does have several limitations. These limits serve to identify areas of future research.

The indirection introduced by using separate objects for interfaces and representations can incur costs in practice. Unless inlining can be used, the costs can be especially noticeable for small, frequently called operations. Consequently, it would be valuable to discover and/or employ run-time or compile-time optimizations to limit these costs. Similarly, our use of virtual (indirect) function calls is expensive; approaches to reduce these costs without sacrificing the clarity of Chameleon programs are needed.

Chameleon only caches program data that is passed as parameters to a chore work procedure. Caching other data may further improve performance. Future directions include investigating the possible interaction between the Chameleon abstractions and the memory management routines that employ caching. While the memory management routines can provide generality, the abstractions can provide semantic information about the data objects to be cached.

Chameleon, by choice, has a focused algorithm and architecture scope. Having made progress in this area, we look forward to broadening the dimensions. Certainly, effective portability across shared and nonshared memory multiprocessors is desirable. An open question is whether the shared-memory programming model (or, indeed, any shared-memory model) will be able to perform close to the that of the nonshared memory model on nonshared memory machines [31]. And if not, whether the delta in performance is still warranted by any savings in programmer cost.

To close, our experiences with the Chameleon program structure are positive. The structure fosters high performance across different architectures. It uses fundamental software engineering techniques to limit development costs. An important aspect of Chameleon is the separation of concerns it provides to the parallel programmer. Performance related statements are encapsulated in abstract representations, clarifying the function of the source program. Even for programs running on only one target, this separation of concerns can aid program development, maintenance, and tuning.

**References**


Gail A. Alveson received the B.Sc. (Honours) degree from Queen's University, Kingston, Ont., Canada, in 1986 and the Ph.D. degree from the University of Washington in 1990.

She has research interests in software engineering, parallel languages, systems, and environments. She is currently a computer scientist in the compiler group of the Tera Computer Company, Seattle, WA.


He has research interests in software engineering, software evolution, software restructuring, programming environments, and parallel and distributed systems. He is an Associate Professor in the Department of Computer Science and Engineering, University of Washington, Seattle. In 1990–1991, he was a visiting faculty member at Tokyo Institute of Technology and Osaka University.

In 1988, Dr. Notkin received a National Science Foundation Presidential Young Investigator Award. He is the secretary-treasurer of ACM SIGSOFT.