December 1993 UILU-ENG-93-2250
CRHC-93-22

Center for Reliable and High-Performance Computing

ProperCAD II: A RUN-TIME LIBRARY FOR PORTABLE, PARALLEL, OBJECT-ORIENTED PROGRAMMING WITH APPLICATIONS TO VLSI CAD

Steven Parkes
John A. Chandy
Prithviraj Banerjee

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

ProperCAD II: A Run-Time Library for Portable, Parallel, Object-Oriented Programming with Applications to VLSI CAD

Steven Parkes
parkes@uiuc.edu

John A. Chandy
j-chandy@uiuc.edu

Prithviraj Banerjee
p-banerjee@uiuc.edu

Center for Reliable and High-Performance Computing
University of Illinois
1308 W. Main
Urbana, IL 61801

This research was supported in part by the Semiconductor Research Corporation under grant 93–DP–109.

Abstract

Despite the increasing availability of parallel platforms, their wide-spread use in the solution of large computing problems remains restricted to a fairly narrow set of applications. This is due in part to the difficulty of parallel application development which is itself largely the result of a lack of sophisticated environments for parallel application development. Further, though the number of parallel platforms is increasing, the convergence of parallel architectures and operating systems does not appear to be similarly increasing. Given that most development environments are targeted towards a particular architecture, it is difficult to amortize development costs over a wide base of installed machines.

In this research, we address these problems through the application of two significant technologies, object-oriented design techniques and the actor model of concurrent computation. Our approach is manifested in the ProperCAD II library, a C++ object library supporting actor concurrency on microprocessor-based parallel architectures and appropriate for applications demonstrating medium-grain parallelism. The development of the library has been driven in general by the requirements of large, unstructured problems, which have not proven amenable to techniques used in scientific computing, and in particular by the requirements of VLSI CAD applications. The efficacy of our approach and implementation is demonstrated through `ProperHITEC`, a parallel implementation of the `HITEC` algorithm for automatic test pattern generation, and `ProperPLACE`, a parallel implementation of the `TimberWolf` simulated-annealing algorithm for cell placement.

Table of Contents

Acknowledgments

The authors would like to gratefully acknowledge the support of the Argonne National Laboratory, the San Diego Supercomputer Center, and the National Center for Supercomputer Applications.

1 Introduction

The desire to employ multiple processors to solve difficult problems has, to date, remained largely unfulfilled for all but a restricted, albeit significant, set of applications, namely the numerical problems found in scientific applications and the database problems found in large transaction-processing applications. Although significant problems occur in many other fields, the techniques used in scientific and transaction-processing applications have not proven equally effective on more general, unstructured problems. Moreover, due to the restricted nature of previous problems, the solutions that have been developed have typically been designed for a particular architecture; for more general problems, in order to target the broad range of installed machines, a degree of portability not previously required becomes necessary. Similarly, to meet cost constraints, potential solutions must be compatible with preexisting code which has historically not been the case.

This chapter provides, as motivation for the work in the sequel, an introduction to both parallel computing and VLSI applications. It introduces the approach taken in this research and highlights the most significant contributions.

1.1. Parallel and Distributed Computing

Parallel and distributed computing, the process of using multiple processing elements to solve a computing problem, has existed almost as long as computers themselves. This section provides a broad overview of the hardware and programming models in existence today.

1.1.1. Hardware Architectures

Traditionally, parallel and distributed architectures could be cleanly divided into two classes, *shared memory* architectures and *message passing* architectures. In recent years, *logically shared, physically distributed* architectures such as the Kendall Square KSR-1[1] have been introduced. Additionally, with the rapid increase in interconnectivity via local- and wide-area networks, virtually all machines support some form of message passing interconnection, giving rise to so-called *hybrid* architectures.

This section considers the key characteristics of each of these architecture classes. It should be noted that these are *hardware* architectures and do not necessarily reflect the programming model as viewed by the application programmer. It is possible, via compilers, run-time libraries, and operating systems, to implement any of the application programming models of the next section on any of the hardware architectures presented below. We use the term *low-level programming model* to describe the model supported directly by the hardware and *high-level programming model* to

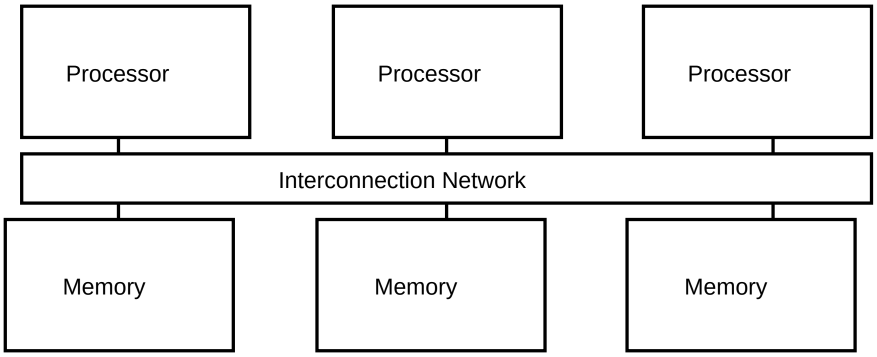

Figure 1.1 Shared memory multiprocessor.

describe the model provided to the applications developer. High-level models will be considered in Section 1.1.2.

Shared Memory

Shared memory architectures are constructed from a number of processing and memory modules which are connected via an interconnection network (see Figure 1.1.) In the first generation of shared memory machines, processing modules generally lacked memory other than that represented by the registers in each CPU and the interconnection network was a bus. Both of these characteristics led to scaling problems. To improve performance, local caches were added to each processor; while this drastically cut the latency of memory references it introduced problems of cache-coherency. More advanced interconnection networks were also developed, including cross-bar and multistage networks.

Even though the addition of cache mechanisms has enabled the scaling of shared memory machines to larger sizes, the difficulty of maintaining cache coherence typically limits the feasibility of this approach to tens of processors.

Message Passing

In message passing architectures, each processing module consists of both a processor and local memory. Processor modules are interconnected by a network (see Figure 1.2.) Typically, network bandwidths are at least an order of magnitude less than those of the buses in shared memory machines. The most significant characteristic of message passing systems is that each processor has its own individual address space; access to the network is explicit, via primitives such as `send` and `receive`. The bandwidth of the network in these architectures has traditionally been less than that of shared memory machines and often varies between different pairs of processors. As a result, these architectures are considered more scalable than shared memory architectures. Machines of this type have been built with thousands of processors. Of course, the lower bandwidth of the network means that many algorithms that perform well on shared memory machines may perform poorly on message passing architectures.

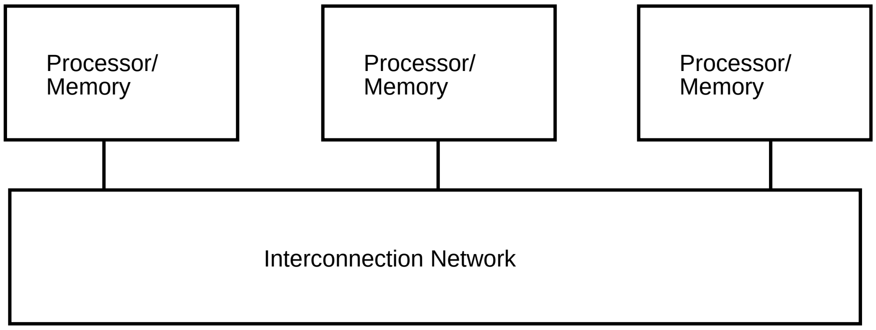

Figure 1.2 Message passing multiprocessor.

Logically Shared, Physically Distributed Memory

In the late '80s, a new architecture was developed using techniques borrowed from both the shared memory and message passing architectures. This architecture uses hardware components similar to those developed for message passing to implement a low-level programming model mirroring shared memory. Thus, though an interconnection network is used, the hardware does not support `send` and `receive` primitives. Instead, the low-level programming model is a uniform address space and the hardware detects accesses to non-local memory, sending the appropriate messages to gain access to the necessary data. Consistency in these systems is usually maintained via a combination of hardware software in a mechanism called a directory[2]. Contemporary machines in this class are Non-Uniform Memory Architecture (NUMA) machines such as the Stanford DASH[2] and Cache-Only Memory Architecture (COMA) machines such as the KSR-1[1].

Hybrid Shared Memory and Message Passing

Hybrid architectures combine both shared memory and message passing architectures. Some processors will share a uniform address space; otherwise communication requires explicit `sends` and `receives` (see Figure 1.3.) With the recent dual growth in workstation clusters and desktop multiprocessors, hybrid machines are becoming ubiquitous. Further, MPP manufactures are beginning to consider hybrid architectures for their machines; the INTEL Paragon supports configurations which have shared memory multiprocessors at each node within the mesh interconnect.

1.1.2. High-level Programming Models

In this context, "high-level" means the programming model as seen by the applications programmer, not the operating system designer. The overall programming model can be broken into two fundamental components, the *communication model* and the *thread model*. Though a complete model requires aspects of both components, considering the components separately helps clarify the issues involved while making more explicit the space of possible complete models. While only a few combinations are currently in use, almost any combination of a communication model and a thread model could form a new complete programming model. We reiterate that these models are high-level and thus with proper software support could be implemented on any of the hardware architectures described in the previous section. For current purposes, we are not concerned with

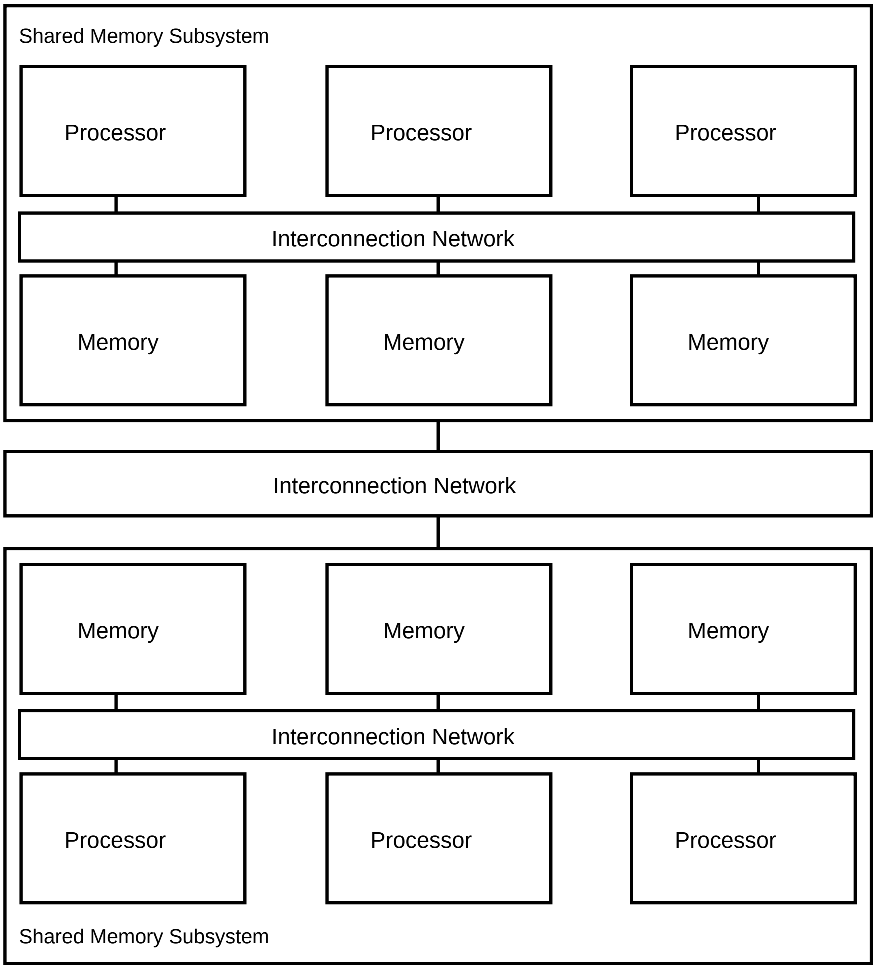

Figure 1.3 Hybrid multiprocessor.

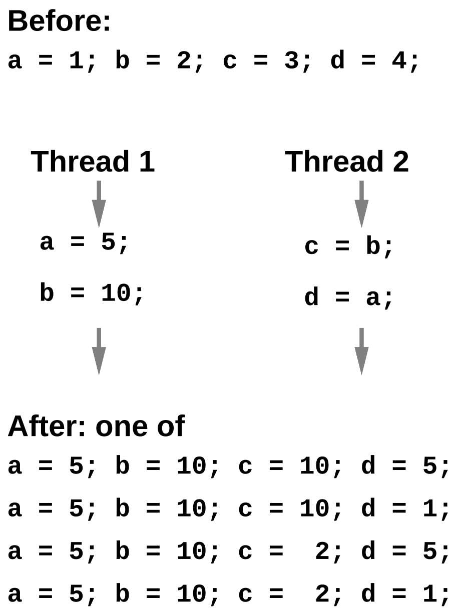

Figure 1.4 Communication and Consistency in Shared Memory.

the cost of high-level models on dissimilar low-level models.

Communication Model

The communication model defines how the "threads" of a program coordinate data interchange amongst themselves. Though we use the term 'thread' in this section, we defer precise specification of a thread to the sequel. For the purposes of this section, threads may be considered an active execution environment (stack) which vies for processor time with other threads.

Shared Memory In the shared memory programming model, the application sees a single flat address space. Communication is implicit, through access to shared variables. To fully support such a model, a method of interprocessor synchronization is necessary, usually implemented at the lowest level via atomic operations such as `test-and-set` or through higher-level abstractions such as semaphores. The exact semantics of a semaphore, i.e., busy-wait versus rescheduling, cannot be defined precisely without reference to a thread model.

Specifying the exact semantics of shared memory machines is complicated by the existence of different *consistency models.* With hardware support for caching and load/store reordering, the most conservative model of shared data consistency, *sequential consistency*[3], is prohibitively expensive. Thus in addition to the uniform address space, a shared memory model must explicitly define the aspects, both deterministic and non-deterministic, of access to shared memory.

An example of shared memory communication is shown in Figure 1.4. In the figure, two threads access two shared integer variables, `a` and `b`. Because the model specifies a shared address space, the variables exist at the same addresses in both threads. The figure demonstrates the consistency problem. Though the first thread stores `5` into `a` followed by storing `10` into `b`, it

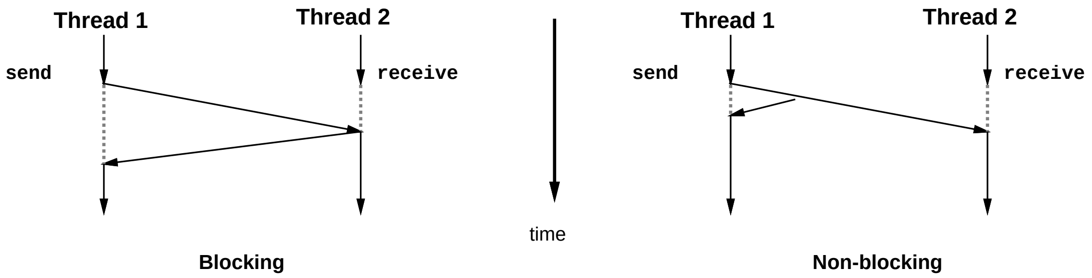

Figure 1.5 Communication through Message Passing.

is possible, under certain consistency models, for the second thread to see the change to `b` before seeing the change to `a`.

Message Passing In the message passing model, a `send` primitive is used to send data from one thread to another thread. An explicit `receive` operation must be executed before the data is available to the receiving thread. Figure 1.5 shows two possible ways communication can occur in a message passing model. In the blocking case, the `send` operation does not complete until the corresponding `receive` operation has begun. This style of communication is one of the types supported by the Thinking Machines CM-5 CMMD library. In the non-blocking case, the `send` operation completes when the necessary data are copied out of the application buffer; it is not necessary that the corresponding `receive` be executed. This type of message passing is common on the INTEL iPSC hypercubes. Variations on message passing provide for broadcasting, multicasting, synchronous and asynchronous communication, and typed messages.

Thread Models

There are two principal thread models, which we will call *physical* and *virtual.*

Physical Threads In a physical thread model, the thread, as seen by the application program, is a processing element and generally available for "exclusive" use by the application. If the code running on a thread executes a blocking call, the entire thread is blocked for the duration of the call.

The physical model includes those systems which provide more threads than processors but for which the developer cannot rely on the underlying run-time library efficiently handling a number of threads vastly greater than the number of available processors.

Virtual Threads In a virtual thread model, the number of threads visible to the program is typically not related to the number of processing elements. Applications developers think in terms of a number of threads convenient for the application at hand. It is the responsibility of the underlying language and run-time support to map these virtual threads to physical processors. When a virtual thread blocks, it is expected that the underlying run-time support will find another virtual thread to schedule. This model is often referred to as *light-weight threads.*

Complete Models

A complete model is created by combining a communication model with a thread model and then specifying the semantics of the interaction between the two components. We consider three complete high-level models. Many more are possible, but the examples serve to demonstrate the issues involved.

Unix Shared Memory Multiprocessor The model of a Unix shared memory machine combines the shared memory and physical thread components. The underlying consistency model is that of the underlying hardware and varies from architecture to architecture. The model is considered physically threaded because the Unix operating system—in particular, the scheduler—does not perform well when the number of kernel-scheduled threads is much greater than the number of processors (e.g., 100 times or more.)

MPI The message passing interface (MPI) model is a send-receive model currently being standardized by a consortium of manufactures and users. The model is similar to the programming interfaces on the INTEL iPSC and Paragon multicomputers and supplied by libraries such as PVM[4] and Express[5].

The basic model provides a send/receive interface over a physically-threaded model. Programs written for this model are written for a fixed number of processors. While they can be parameterized by the size of the machine, the model provides little or no support for context switching a physical thread over multiple virtual threads.

In the MPI interface, the basic send/receive model is augmented with such concepts as process groups and environments, making the interface more applicable to virtual threads. Experience with the advanced virtual thread interface for message passing systems is, to date, limited.

Actor An actor model is based on continuation passing[6] using virtual threads. From the programmer's point of view, an actor is an object with its own thread of control that at most times is blocked waiting for a message—a continuation execution of one of its methods.

Continuation passing is similar to message passing but omits the explicit receive operation. Instead, each message sent contains enough information to determine the action to be invoked by the receiver. This model is much like the active messages model[7]. Figure 1.6 shows how continuation passing might occur in a C++-like language. In continuation passing style (CPS), execution of a member function is initiated through the execution of a continuation. In the figure, execution of the member `p` for actor `a` is initiated though calling a continuation and is represented by the arc labeled `p(c)`. In response, `a` executes a member function of `b` by creating a continuation, `x`, from the actor address and member function pointer and using function call syntax to call the continuation. The figure demonstrates one of the important characteristics of CPS; control flow need not be stack-based as it is in function call mechanisms. In the example, `b` need not return a value to `a` in order for it to return to the caller of `a::p`; instead, using the continuation `c` passed to `a::p` and then to `b::q`, the result is communicated directly to the initial caller.

The actor model combines continuation passing with a virtual thread model. Most of the time the thread associated with an actor is blocked waiting for the continuation execution of one of its methods. When a continuation destined for the actor is executed, the appropriate member

```
class Actor {

  int data;

  int p(Continuation c)
  { ...
    Continuation x ( b, q );
    x(c);
    ...
  }

  int q(Continuation c){ ... c(5) ... }

};

Actor a, b;
```

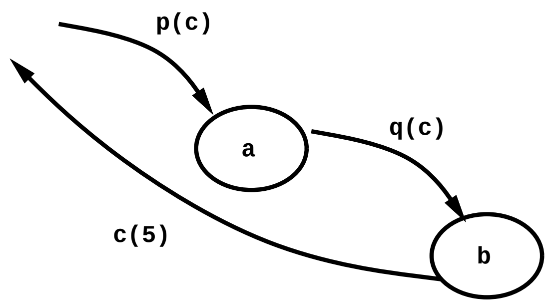

Figure 1.6 Communication through Continuation Passing.

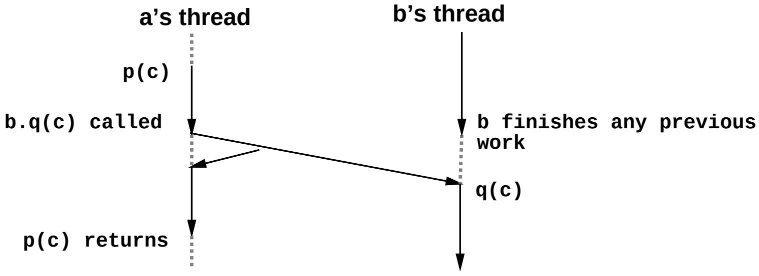

Figure 1.7 Multi-threaded continuation execution.

function is executed. As part of the execution, the actor may execute other continuations, create new actors, or perform computations, possibly with side effects.

In a parallel context, continuations express parallelism because the call to a continuation returns after scheduling the future execution of the continuation body rather than synchronizing with the actual execution. Figure 1.7 shows how the execution of the code in Figure 1.6 might occur. `a.p(c)` is called via a continuation (not shown). In response, it does some processing and executes `b.q(c)` via a continuation. While processing the continuation call, the parameters are copied and a message is sent to the processor responsible for scheduling `b`'s thread. As soon as the message send has started, `a` resumes its processing. At some later point, `b` finishes any previous continuation and becomes available to process the continuation from `a`. Each actor has a single virtual thread so only one message can be delivered at a time. The figure emphasizes the particular communication and thread models comprising the actor model. The communication style natural to actors is asynchronous; basing an actor implementation on a synchronous model would in most cases result in performance degradation. Since every actor has a thread and an actor is simply a concurrent object, actor programs typically have thousands if not hundreds of thousands of virtual threads. Thus, the virtual thread mechanism used by an actor implementation must be extremely efficient.

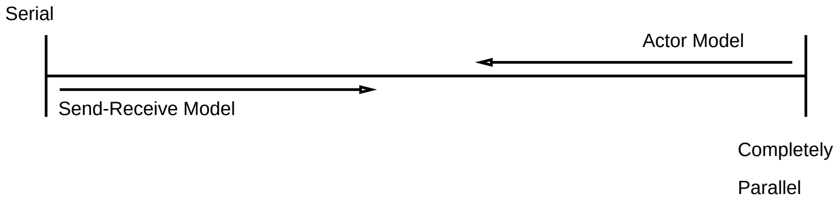

Figure 1.8 Spectrum of Parallelism. Arrows indicate increasing programmer effort.

Comparison of MPI and Actors With the MPI style interface much more well established, the actor model must provide significant benefits in performance or programmability to be considered useful.

Expression of Parallelism One can view the gamut of parallelism as a one dimensional space with extremes representing completely serial and completely parallel programs (see Figure 1.8.) If we consider the MPI and actor programming styles, we see that they lie on opposite ends of the parallelism spectrum. For example, if we consider a send-receive code that sends two messages from a thread A to a thread B, in the most straightforward, synchronous case, parallelism is decreased by the imperative `receive` primitive:

```
// Thread 1                          // Thread 2
send( 2, type1, data1 );             receive( type1, data1 );
send( 2, type2, data2 );             receive( type2, data2 );
```

Though the semantics of the application may allow the messages to be processed in any order, the underlying model requires that the messages be received in the order sent and the serial nature of code execution implies that one receive command must precede the other. Though it is possible to express unordered reception via asynchronous `send` and `receive` primitives, such code is significantly more difficult to write. Thus, the basic MPI interface is represented at the left of the parallelism spectrum; the developer must apply *more* effort to express *more* parallelism.

In the actor model, message reception may occur as long as the actor is not processing a previous message. In contrast to the MPI case above, the actor will receive whichever message arrives at the actor thread first. There is, however, a dual to the MPI case; if there is a semantic dependency on message processing, the actor must handle the case where message reception order is reversed. In this case, the actor must delay processing of the second message until the first is received and processed. Thus, the actor model starts at the completely parallel end of the spectrum and requires that the programmer apply *more* effort to express *less* parallelism. The extreme amount of parallelism in an actor program can sometimes lead to difficulty in programming. To remedy this, most actor languages include constructs for shifting order maintenance from the programmer to the run-time system. As will be seen in Chapter 3, combing the actor model with an imperative language such as C++ also serves to simplify the expression of parallelism without putting an unacceptably large burden on the developer.

Composability Another point of comparison is the ease of *composability*, the process of taking two existing modules for solving unrelated problems and combining them into a single application to solve a more complicated problem. For example, an application might require the processing of independent linear system solver and matrix multiplication problems. Composability, then, refers to the ability to take existing modules for solver and multiplication and to use them, without modification, to solve the appropriate subproblems.

In composability, physical thread-based models tend to fall short of the virtual techniques because partitioning, usually static, of available processors is required. Designers of message passing interfaces such as MPI are trying to implement ways of solving this problem but as yet there are no practical examples of such features.

Implementations of the actor model, being based on a virtual thread model, implicitly perform load balancing when a new actor is created. Thus, if an actor type exists for each of the solver and multiplication problems, an actor can be created for one without interfering with the other. It is the task of the run-time system to load balance and schedule the actors to take advantage of available processing elements.

Figure 1.9 shows graphically the difference between typical message passing and actor implementations of the linear system solver and matrix multiplication problem. In the message passing case, we assume a function exists for each subproblem and that each function assumes it has full use of the multiprocessor. Thus, individually each of the two subproblems is solved in parallel but the two subproblems are sequentially ordered with barriers. If the matrices are sparse, as is typical, it is often difficult to use the processors efficiently. In the actor case, since each subproblem is represented by an actor, the two actors, and by extension, the subproblem actors, are created in parallel and only synchronized when the two operations are completed. The result is higher processor efficiency. (The barriers shown in the actor example are symbolic; though an actor model does not have blocking primitives, the combination of the code effectively creates barriers.)

Integration of Object-Oriented Design Techniques Since continuations are a language-based concept, their integration into object-oriented languages is relatively straightforward. In an object-oriented language, a continuation conceptually contains both the address of the target object and a pointer to the method to be invoked on the target object. The dynamic binding mechanism (virtual functions in C++) is invoked in exactly the same manner as it is for serial languages.

On the other hand, in the send-receive model communication looks more like an input/output operation. When a number of possibilities exist for the type of a received message, some sort of type-globbing mechanism is necessary in the `receive` primitive. Subsequently, a type switch, e.g.,

```
if ( type == a )
     ...
else if ( type == b )
     ...
```

is required. Type switches are difficult to maintain and anathema to object-oriented design. The elimination of the need for such constructs was one of the main impetuses for the development of the object-oriented methodology. Any technique, such as continuation passing, which obviates the need for type switches is considered highly desirable.

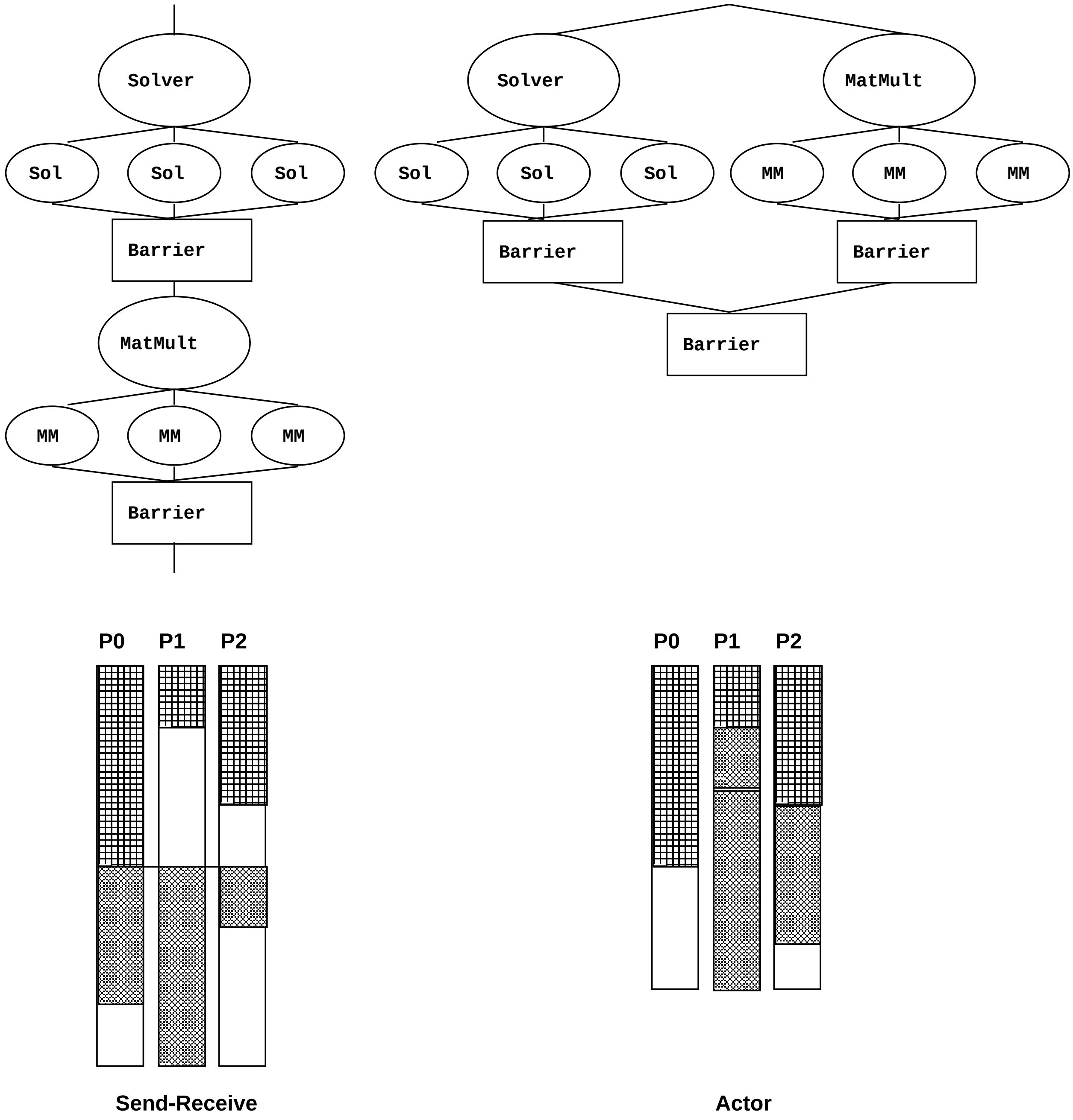

Figure 1.9 Linear System Solver and Matrix Multiplication on Send-Receive and Actor models.

1.2. VLSI Computer-Aided Design

The types of large, unstructured problems encountered in general computing are typified by those in electronic design automation for VLSI CAD. Examples of VLSI CAD problems are:

ATPG Automatic test pattern generation (ATPG) for VLSI circuits is a difficult problem (NP-complete) which is nonetheless critical if the quality of ever larger VLSI devices is to be maintained.

Synthesis Owing to increasing circuit densities, the last decade has seen a considerable increase in interest in algorithms for the automatic synthesis of VLSI circuits. Industry consensus is that only through synthesis will it be possible to manage the design complexity of the current and coming generations of VLSI chips. Most synthesis algorithms are both memory and processor intensive and display a quality of results tightly coupled to the resources applied to the problem.

Extraction Extraction is the process of taking a VLSI mask-level layout and extracting circuit connectivity and parametric values. The results of extraction are used to verify both design correctness and performance requirements, usually after automatic cell placement and routing. Extraction is typically performed in terms of rectangles on various masks of the layout; the number of these rectangles can approach 100 million on contemporary microprocessor designs. Few processors in industry have the resources to handle such a design efficiently. Given the frequency of this problem (extraction is usually iterated with design changes to verify changes and to update extracted parametric information), techniques to take advantage of all available resources are valuable.

Cell Placement and Routing When logic design for a VLSI circuit has been completed, cell placement and routing are performed. With chips approaching tens of millions of gates, the complexity of these problems is enormous. Moreover, one of the most popular techniques for placement, simulated annealing, is an inherently expensive operation. As in most other CAD tasks, the quantity of resources applied to the problem has a direct impact on the quality of results.

With the preponderance of evidence indicating that virtually any method of managing the algorithm development process and any technique for improving quality through the application of additional resources would appear promising, it is perhaps surprising that neither parallel processing nor object-oriented techniques are well represented in the CAD development community. To date, with few exceptions, the only use of either technology is almost exclusively in the area of graphical user interfaces(GUIs). This situation is not without justification:

- For more than a decade, most CAD development has been performed in the C programming language and until the advent of C++ use of an object-oriented language implied the sacrifice of existing code, an unacceptable alternative. With the availability of C++, adoption is slowly occurring. C++ is easily an order of magnitude more complicated than C and is still undergoing rapid development; development tools are only now attaining the degree of stability needed for even the most aggressive commercial development.

- Due to a lack of widely available libraries (again, outside the GUI area) much of the promise of C++ remains to be realized. Given the ability of C++ to trade flexibility for efficiency, the process of generating reusable code is made more complicated by the fact that the set of design choices, in terms of flexibility versus overhead, for one application may not be acceptable for another; CAD problems, with their inherent complexity and size, are known to be sensitive to overheads in area or space.

- Until recently, the wide-spread availability to CAD users of parallel platforms has been severely limited. Supercomputers have in general been limited to the restricted applications mentioned previously and thus the techniques developed for these platforms have had no place for application in the CAD community. This situation leads to a chicken-and-the-egg problem: with little availability of parallel platforms, interest in parallel solutions to CAD problems has been relatively low. With only limited development of parallel applications, little impetus exists for CAD users to explore the cost-benefit tradeoff of parallel platforms, even though the costs associated with VLSI development and fabrication are high and time-to-market is a critical factor.

- The generation of *useful* parallel algorithms has also been impeded by concurrent rapid improvement in serial algorithms. Often by the time a parallel CAD algorithm is completed, it lags significantly behind the quality, and sometimes even the performance, of contemporary serial algorithms.

1.3. Approach

Based on past experience, a strong argument can be made that any practical, parallel, unstructured application—and CAD applications in particular—must be based on models and implementations that are portable across the widest variety of parallel machines, scalable to the maximum degree feasible, and based initially upon the best available serial algorithm. These constraints are explicitly addressed in the approach taken in the ProperCAD project. The goal of the project is the development of a library which provides a seamless, high-level programming model across a variety of parallel architectures along with a set of applications based on that model that address the most significant tasks in electronic design automation [9, 10, 11].

The main portion of the work reported here is the development of a layered library which can be used as the basis for porting serial algorithms to an abstract parallel machine. The library is divided into two layers. The lower half, the abstract parallel architecture, interfaces with underlying hardware and system software to provide a consistent interface across a wide variety of parallel machines including heterogeneous combinations of parallel architectures. The upper half, the actor interface, is an implementation of the actor-model of concurrency which, through abstraction, simplifies the task of writing parallel applications. The actor interface helps the application designer avoid dealing with low level, architecture-dependent issues.

Specific contributions of this work are:

1. A library approach to portable, actor-based parallelism in a strongly- and statically-typed language.

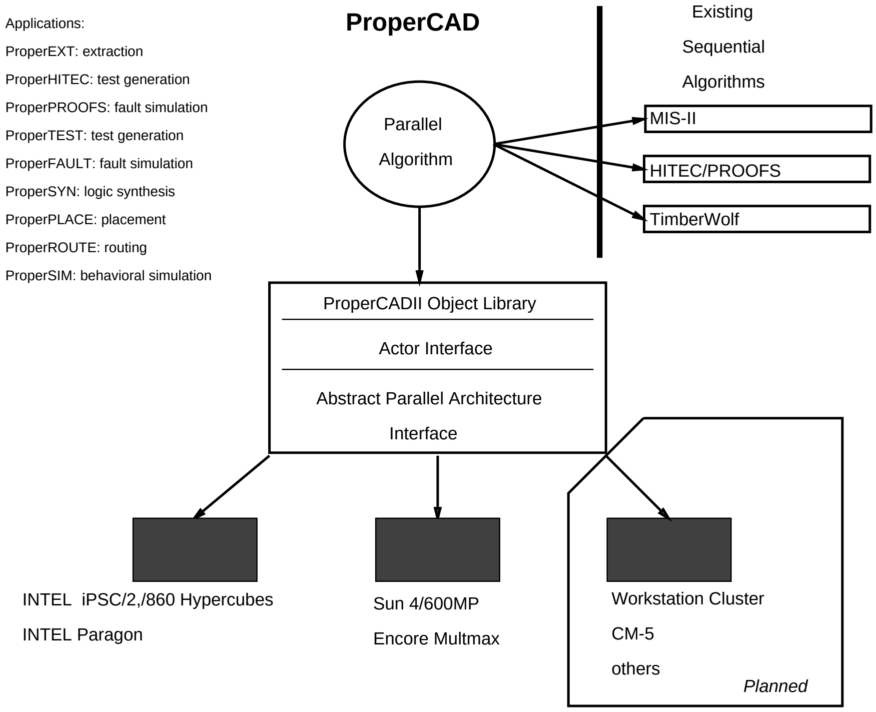

Figure 1.10 An overview of the ProperCAD project [8].

2. A set of classes for controlling, or meta-programming, an actor system on contemporary medium-grain machines.

3. An implementation of aggregates with additional features applicable to medium-grain computation.

4. An abstract parallel architectural model, with an object library-based implementation, which is capable of describing the full range of contemporary parallel architectures.

5. A parallel algorithm for ATPG, incrementally derived from a well-known top performing algorithm, which achieves significant multiprocessor utilization without degradation of resultant quality.

6. A parallel algorithm for cell placement which achieves significant multiprocessor utilization without degradation of resultant quality.

1.4. Report Overview

The sequel is broken into two parts. The first part presents our approach to concurrent object-oriented computing. A small numeric kernel, the Gauss-Jacobi algorithm for the solution of partial differential equations, is used to exemplify the approach. Individual chapters present an overview of the library, the low-level abstract parallel architecture, the high-level actor interface, and the results of the Jacobi kernel on a number of parallel platforms.

The second part details two CAD applications, ATPG and cell placement. Each problem is described along with a description of the serial algorithm which forms the basis for the parallel application. The parallel application is then described and performance results presented.

Part I

Run-Time Library

2 Overview

2.1. Approach

The approach to parallelism taken in this research is the development of a library which provides simultaneously a high-level, object-oriented abstraction to application programmers and an underlying run-time implementation of that interface tuned to the highest efficiency possible. The library enables an application designer to incrementally apply object-oriented programming mechanisms to develop a parallel application while maintaining compatibility, i.e., sharing code, with an existing serial application. This approach differs from previous work in a number of ways, most notably in the sets of target architectures, applications, and languages.

2.1.1. Targets

In this work, we specifically target those architectures developed atop contemporary microprocessor technologies, supporting medium-grain parallelism. Likewise, the target application set consists of those applications which require medium-grain parallelism. The term *medium-grain* requires some clarification.

Medium-grain Parallel Architectures

It is fairly simple to quantify the difference between fine-grain architectures and others, fine-grain architectures being those having upwards of a thousand processors, network latency on the order of a few microseconds, and per-node memory sizes on the order a few kilobytes per MIP of processing power[12].

Coarse-grain applications are ones for which the ratio of computation to communication is very high. Rather than trying to define an exact level of performance to differentiate coarse-grain applications, it is helpful to consider protection boundaries. If processes must validate all communication with other processes for complete safety, we would consider such communication coarse-grain. The motivation for such differentiation is that the overhead implied by such checking usually places a lower bound on the cost of communication, thus limiting performance as communication requirements increase.

Medium-grain architectures cover the wide but ill-defined range between fine- and coarse-grain architectures. Current machines from Thinking Machines, Intel, and Cray fit within this class, as do clusters of workstations incorporating the most recent networking technologies. For the purpose of this work, we consider clusters of workstations connected via an Ethernet LAN to be medium-grain machines. Depending on the speed of routers and bridges, even workstations connected to multiple Ethernet segments could provide enough performance for some applications.

Medium-grain Parallel Applications

While it is difficult to quantify the characteristics which make an application medium-grain, the library demonstrates this leaning in the interface it provides to the programmer. The library provides a set of parallel constructs which can be used to express parallelism while not precluding the use of native constructs provided in the language. In contrast, fine-grain parallel languages such as the original actor languages differ in that all constructs are implicitly parallel[13]. Though it is possible to express programs in the library using exclusively parallel interfaces, the performance of such an application on the targeted architectures is not likely to be high.

Even more difficult to distinguish is medium-grain parallelism from coarse-grain parallelism. The most significant difference between the medium-grain target here is the integration with the target language. Medium-grain implementations try to reduce the conceptual barrier between parallel and serial constructs. Additionally, on coarse-grain systems parallel constructs such as message passing typically cross hardware protection boundaries and incur significantly more overhead than the equivalent medium-grain operation. Just as an application aimed at fine-grain parallelism is likely to perform poorly on a medium-grain system, an application written for medium-grain parallelism is likely to suffer on an architecture where communication is burdened with large overheads.

2.1.2. Constraints

Several significant constraints were placed on the design space that was explored in development of the library. Most constraints were a result of the targeted application communities and practical usability.

C++

The overriding constraint was that the implementation be in either C or C++, with a noted preference for C++. The choice of C++ was made for two reasons. First, C++ has a level of expressiveness not found in other object-oriented languages. Unlike most object-oriented languages, C++ has the ability to express very low level constructs. This ability is critical to CAD applications which manipulate huge datasets and where even small overheads in data structures or functions can have dramatic impacts on application performance. C++ is also available on virtually all machines, parallel machines being notorious for lacking all but the most basic programming support. In view of this last observation, no extensions to either C++ or the programming environment were considered, such extensions being likely to prohibit the use of existing compilers and debuggers.

Portability

The wide variety of extant medium-grain parallel architectures makes the issue of portability extremely important; an implementation that is difficult to port would have little chance of success. Therefore, a major goal of the library design process was to try to isolate any machine-dependencies from both the application developer and, to the greatest extent feasible, the components of the library itself.

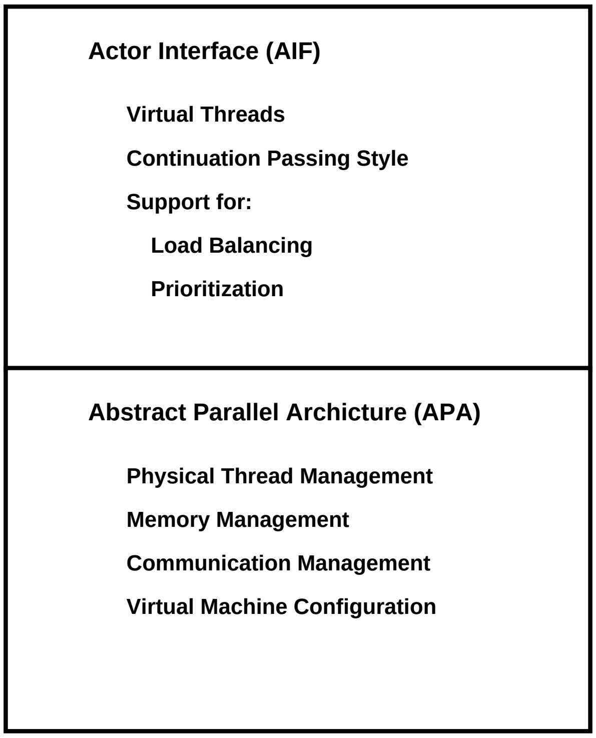

Figure 2.1 The ProperCAD II Library

2.2. Library Components

To simultaneously maximize the portability and efficiency of the library, the library was broken into two distinct interfaces (see Figure 2.1.) The upper level of the library, the Actor Interface (AIF), provides a high-level programmer interface, one that helps the application developer limit complexity, via abstraction and encapsulation, without limiting parallelism. The Actor Interface is based closely on the Actor paradigm and supports several advanced features such as aggregates (collections of actors) and meta-programmability.

The lower of the two interfaces, the Abstract Parallel Architecture (APA), provides a set of objects and interfaces which can be used to describe and utilize resources needed by virtually any programming model, across a wide variety of architectures. Such resources include processor threads, synchronization primitives, and memory management functions.

2.3. Related work

There are many approaches to concurrent object-oriented programming, covering wide ranges of both techniques and targeted platforms and applications. Since the work presented in the sequel is based on the Actor model targeted to medium-grain parallel architectures, previous work in these areas is considered in detail. Alternate approaches are considered more briefly. Previous work is divided into language-based work and library-based work, with languages further broken into actor and non-actor models.

2.3.1. Actor Languages

The Actor model for concurrent computing was first proposed by Hewitt[14] and subsequently refined and codified by Agha[13, 15]. In its original form, it is a message-driven model targeted towards fine-grain, massively parallel platforms.

A number of pure actor languages have been implemented since the proposal of the original model. Following the original model, most Actor languages are functional, i.e., side-effect free, and untyped. The majority of implementations are targeted to fine-grain concurrency on massively-parallel MIMD architectures. Few of the original actor languages were actually implemented on distributed machines. The earliest languages were Act[13], Act1[16], Act2[17], Act3[13], Sal[13], and ABCL/1[18]. A more recent contribution, with an emphasis on reflection, is MERING IV[19]. Of particular note is the HAL language of Houck and Agha[20] which was implemented on top of the Charm programming system[21]. HAL is a truly distributed implementation and is notable for having formed the basis for further exploration into extensions of the Actor model.

Aggregates

Chien[12] recognized the need for a multi-access interface to augment the serial interface of Actors. He proposed aggregates as collections of actors which present a unified, yet multi-access, interface to client code. Key to aggregates is support for efficient intra-aggregate addressing. Chien developed an actor language, Concurrent Aggregates (CA), which in addition to the features of actors, aggregates, and intra-aggregate addressing, provides support for first-class continuations and messages.

The Concert system of Chien, Karamcheti and Plevyak[22, 23], is a compiler and run-time support system for CA on stock hardware, i.e., such contemporary parallel machines as Thinking Machines' CM-5 and Intel's Paragon. The emphasis of the Concert system is on extensive compiler analysis and optimization.

ActorSpaces

Recently, Agha and Callsen[24] have proposed ActorSpaces as a technique for extending the actor model to open systems. ActorSpaces offer an alternate approach to collections of actors than that of aggregates. Rather than the fixed-size, homogeneous collections of actors in CA, ActorSpaces represent variable-sized heterogeneous collections of actors. The emphasis in ActorSpaces is on providing an open interface, via which an actor can communicate with other actors without knowing *a priori* of the existence of the other actors, something not possible in the basic actor model.

Charm

Charm[25, 21, 26] was developed specifically to address the need for a portable, parallel interface and implementation for parallel programming across a wide variety of both shared memory and message passing architectures. Charm differs from other actor languages in that it is targeted to medium-grain architectures rather than the fine-grain architectures. Among the most significant and unique features of Charm are:

Imperative-Language Interface Charm represents an extension of the C programming language with extensions to support actor- and aggregate-like characteristics. With a few exceptions, most of the the imperative constructs of C are retained. Recently, with the introduction of Charm++, the C-language model of Charm has been extended to a C++-based interface.

Meta-programming features Charm provides developer-visible models for the operation of the underlying run-time system in the areas of load-balancing and prioritized message delivery. Strategies in both areas can be selected from a set of alternatives at link-time. Of particular note is work with lexicographically-ordered message priorities applied to search problems[27].

Library Types Charm provides a library of object types with parallel semantics such as read-only variables, distributed computation types such as accumulators, and distributed mappings.

The first phase of the ProperCAD project, to which this work is a successor, was based upon Charm and many of the issues addressed in this work reflect the experience gained in using Charm.

2.3.2. Non-actor Language-based Approaches

Significant work has been done in concurrent object-oriented computing not based upon the actor model. We consider in particular C- and C++-oriented solutions.

Gannon and Lee[28] developed pC++, an extension of C++ with support for distributed data structures. The flavor of pC++ is much that of High-Performance FORTRAN (HPF). pC++ provides support for distributed collections of arbitrary types, both array-based and tree-based, with full support for the C++ mechanisms of derivation and dynamic-binding.

Compositional C++, or CC++, proposed by Chandy and Kesselman[29] takes an alternate approach to parallelism; where in pC++ processor control is implicit in the parallel data structures, in CC++ parallelism is achieved though imperative constructs which cause particular code fragments to be executed concurrently on different processing threads. CC++ also provides a number of synchronization primitives necessary for a thread-oriented programming interface.

The Experimental Systems Kernel, or ES-Kit, of Leddy and Smith[30] is implemented via modifications to an existing C++ compiler and as such tries to stay true to the spirit and syntax of C++. In ES-Kit, pointers are extended to address a global namespace and remote execution is represented by the execution of a method call though a pointer to a non-local address. Object distribution is either automatic or under program control via the C++ placement syntax[31]. The fundamental parallelism constructs in ES-Kit are remote procedure calls (RPCs) and futures[32].

The Amber system[33], derived from Presto[34], is an extension of C++ targeted specifically toward a workstation cluster running the Topaz operating system[35]. In Amber, the approach is to explicitly locate a shared datum on a particular node and then to cluster Topaz threads on that node. When access is made to a remote node, the run-time system traps to the Amber kernel and the thread of control is transferred to the processor on which the data value resides.

COOL, developed by Chandra, Gupta, and Hennessy[36], is also based on thread-explicit extensions to C++, in this case targeted towards shared memory architectures in general and the Dash architecture in particular. COOL provides a full-range of classical synchronization constructs.

Linda, developed by Carriero and Gelernter[37], represents a relatively radical approach to concurrent processing. In Linda, shared data are represented by a shared *tuple-space* to which all functions have access. Elements in the tuple space are key-value pairs which are accessed via pattern matching on the key. This formulation has particular benefit in logic-programming and artificial-intelligence and was one of the motivations for the development of the ActorSpaces extension to the actor model.

POOL-T, developed by America[38], is based upon message passing but uses synchronous message passing rather than the asynchronous message passing of actors. POOL-T is targeted particularly towards large system development on medium-parallelism architectures and was developed from the beginning with an interest in applying formal methods for proving program correctness.

2.3.3. Library-based Approaches

One of the most attractive aspects of library-based approaches is that they are at least to a first approximation compatible with existing compilers and other development tools. Again, we consider C and C++ libraries.

ACT++ is a C++ library-based implementation of the actor paradigm developed by Kafura and Lee[39]. ACT++ implements an Actor base class which supports the actor model primitives, *new*, *send*, and *become*. Additionally, ACT++ supports an RPC-style of actor method invocation[13]. ACT++ supports the use of normal (non-actor) C++ objects, but only as private, non-shared acquaintances of an actor. ACT++ is targeted toward medium-grain architectures but implementation details on parallel architectures have not yet been reported.

The Paragon project of Chase, Cheung, Reeves and Smith[40] is implemented via a C++ library for support of distributed data structures. Support is provided for distributed arrays, both through partitioning and replication.

PVM, developed by Sunderam et al.[4] is a widely used library for parallel programming on local area network-connected processors and, recently, a number of parallel architectures. The library provides C and FORTRAN interfaces for synchronous and asynchronous message passing and provides support for heterogeneity of processing elements, both in data representation and processing power. Included tools provide particularly strong support for easy machine configuration, debugging, and performance analysis.

p4, developed by Butler and Lusk at Argonne National Laboratory[41], provides a similar message passing interface to that of PVM and also adds an interface for shared memory. In addition to supporting network-connected processors, p4 provides support for a wide range of parallel platforms.

3 Actor Interface

The actor model is the high-level and preferred interface for application development. In this chapter we consider the objects in the ProperCAD II library that support an actor model. We begin with an examination of the original actor model.

3.1. Actor Model

The fundamental object in an Actor paradigm is the actor. An actor is an object which communicates with other actors by sending messages. Message delivery in an Actor system is *reliable, unordered,* and *fair.* Reliability guarantees that messages that are sent will be delivered. Delivery, however, is unordered; two messages sent from one actor to another are not guaranteed to be received in the order sent. Fairness guarantees that no actor can delay indefinitely the processing of a message by flooding a receiver with messages.

All actions an actor performs are in response to messages; i.e., the model is message-driven. When a message is received, the receiving actor can perform a number of actions in response. The possible actions are use the `send` primitive to send a message to an existing actor; use the `new` primitive to create a new actor; and use the `become` primitive to change its behavior.

send An actor can send a message to any actor for which it knows the receiving actor's name.[1] Actors know their own name and can determine the name of any actor they create. Actor names can be communicated as part of the contents of a message. Otherwise, an actor cannot determine the name of an arbitrary actor and thus cannot send such an actor a message. This property is key to the formalism of the actor model but is also important in that it facilitates encapsulation in software design. The `send` operation completes when the message has been passed to the run-time system; delivery is asynchronous.

new An actor can create new actors using the `new` primitive. Creating a new actor is similar to sending a message; the actual birth of the actor is asynchronous with respect to the execution of the creating actor.

become An actor can change its behavior via the `become` primitive. The utility of the `become` operation is unclear in languages such as C++ which have side-effects. Since it has not yet been shown to be useful in targeted algorithms, we omit further discussion.

[1] Agha[13] refers to these names as *mail-addresses.*

C++ Object	AIF Object	Comments
object	`Actor`	An actor has parallel semantics
none	`Aggregate`	C++ has no built-in collection types
object pointer	`ActorName`	`ActorNames` are pointers to `Actors` or `Aggregates`
none	`AggregateName`	`AggregateNames` are pointers to `Aggregates`
member function	`ActorMethod`	
constructor	`NewActorMethod`	
none	`Continuations`	A `Continuation` is a bound member function pointer
value	`Message`	Used as argument to continuations

Table 3.1 Comparison of C++ and AIF objects

Of particular note is the lack of explicit sequencing primitives in the actor model. Instead, synchronization is implicit and derives from the message reception serialization property of the model; while an actor is processing a message, it may not receive another. Because there are no explicit synchronization primitives, an actor processing a message cannot block or suspend itself in anticipation of a particular event. When the processing for a particular message is complete, the actor returns control to the run-time library and implicitly becomes available for any messages pending for it. This limited synchronization control is key to the concurrency of actors; concurrency in parallel applications is often lost because the interface to a message passing system requires an ordering on message sends and receives. The lack of blocking primitives is also beneficial in terms of portability; since an actor cannot suspend execution implicitly in the middle of a computation, all the context needed to restart the computation upon message reception must be explicitly moved off the run-time stack. While this puts more burden on the developer, the elimination of the need for the run-time system to deal with stack and register manipulation has obvious benefits in terms of portability.

3.2. Library Interface

The actor interface (AIF) provides a set of classes that represent the concepts of the actor model concretely. Table 3.1 shows the correlation between native C++ concepts and the actor equivalents. The AIF does not restrict the use of the native C++ objects but it likewise provides no parallel interpretation.

3.2.1. `Actors`

Actor instances in the actor interface are similar in most respects to class instances in C++. In the ProperCAD II library, all actor types are derived from a common class, `Actor`, provided in the library. The `Actor` interface is very simple[2]:

```
class Actor
{
  METACLASS( Actor )
  protected:
    Actor();
    virtual ~Actor() = 0;
```

[2] Since the private interface of a class is not exported to derived classes, it is elided in most examples. In the case of the `Actor` class, it is also empty. The `METACLASS` declaration at the beginning of a class is a macro used to implement the run-time type information needed by the library.

```
class JacobiBlock : public Actor
{
  METACLASS( JacobiBlock )
  public:
    class New : protected NewActorMethod<JacobiBlockData> {
      METACLASS( JacobiBlock::New )
    };
    JacobiBlock( Message<JacobiBlockData>& );
    virtual ~JacobiBlock();
  protected:

    void performIteration();

    void sendAll();
    void sendBoundary( const ActorName<JacobiBlock>&,
                       int xOffset, int yOffset, int stride );

    class boundary : public ActorMethod<JacobiBoundary> {
      METACLASS( JacobiBlock::boundary )
    };
    void boundary( const JacobiBoundary& );

  private:
    Message<JacobiBlockData> message;
    int iteration;
    int boundariesReceived;

    Matrix& alternateMatrix;
    Matrix* pCurrentMatrix;
    Matrix* pNextMatrix;
};
```

Figure 3.1 C++ code for the `JacobiBlock` actor class definition.

```
};
```

The `Actor` class is an abstract base class and provides only the minimal abstract interface for `Actor` types. Aside from the necessary base class, there are few restrictions on the structure of actor classes. Like all other C++ classes, they may have public, protected, and private members and may be derived from other types, including other Actor types.

As an example, Figure 3.1 shows the definition of a `JacobiBlock` actor type. Instances of this class use the Gauss-Jacobi algorithm, an iterative algorithm for solving partial differential equations, to calculate a block of a larger Jacobi computation (see Figure 3.2.) Each instance exchanges boundary values with neighboring objects at the end of each iteration. The `JacobiBlock` actor class uses normal member functions and data to perform the desired computation. Only the `Actor` base and the `ActorMethods` differentiate it from other C++ classes.

3.2.2. ActorNames

`ActorNames` play the role of pointers and references for `Actor` instances. `ActorNames` are valid in the global namespace of a running program, independent of the number, type, and interconnection of physical threads executing the application. This contrasts with native C++ pointers and references which may or may not be valid, depending on the physical characteristics of the

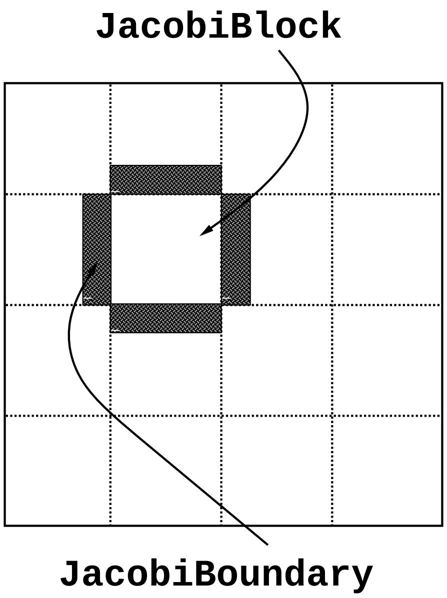

Figure 3.2 Sub-block of Jacobi Computation

underlying machine. To support this functionality, in addition to the address of an actor instance, `ActorNames` encode information which is needed to send a message to an actor irrespective of where the actor exists. Actor names can also be used to send messages to actors which have not yet been created; the underlying run-time system will queue messages until actor creation occurs. An example of this use appears in the next section.

Actor names are first class values; they may be created, copied, assigned, stored, and transmitted to other actors in the contents of a message. In most cases, actor names are created and initialized similar to the way in which C++ references are created.

3.2.3. `ActorMethods`

The message passing Actor paradigm is not the common view of computation used in the C++ community. While the calling of a member function and the sending of a message are treated as more or less synonymous in the larger object-oriented community, each language tends to emphasize one model over the other; the C++ community emphasizes the function-call paradigm.

To that end, in the ProperCAD II library the normal member interface for `Actor` classes has been extended with the concept of *actor methods* which, when executed via *continuations*, fill the role of asynchronous remote procedure calls. An actor method is a nested class, derived from the templated `ActorMethod` library class, which enables the creation of a continuation which will execute the member function of the same name. When such a continuation is executed, an asynchronous invocation of the C++ member is scheduled to occur.

3.2.4. Messages and Continuations

Actor methods are implemented using two abstractions: `Messages` and `Continuations`, both of which are provided by the library. The interaction between the classes comprising these abstractions results in a statically-typed interface to application code; the classes encapsulate all type information necessary to correctly transport the method operand to a distant processor and continuation execution is checked at compile-time for type correctness.

Messages are first-class values that are passed as arguments to actor method calls. Continuations fill the role of bound method pointers; when called—using function-call syntax—execution of the appropriate member function is scheduled. Actual execution occurs asynchronously with respect to the calling actor but is serialized with respect to the target actor.

Referring back to the `JacobiBlock` example, a `JacobiBlock` actor instance creates and sends `JacobiBoundary` instances to neighbors to exchange boundary data:

```
void
JacobiBlock::sendBoundary( const ActorName<JacobiBlock>& neighbor,
                           int xOffset, int yOffset, int stride )
{
    boundary::Continuation aContinuation ( neighbor );
    Message<JacobiBoundary> aBoundary
                    ( xOffset, yOffset, stride, pCurrentMatrix );
    aContinuation( aBoundary );
}
```

Since continuation types play the role of bound function pointers, they require an `ActorName` of the correct type when they are constructed; the member function is encoded in the name of the continuation type. In the example above, `continuation` is a continuation which, when 'called', will schedule an invocation of the `boundary` member of the `JacobiBlock` instance referred to by `neighbor`. Invocation of continuations naturally uses function-call syntax, as seen in the example above.

Continuation calls differ from C++ member function calls in a few key ways, reflecting their parallel nature:

Continuation execution occurs asynchronously When an actor method is invoked via a continuation, a request (message) is passed to the run-time system and dispatched to the location of the target actor. Continuation execution is the sole construct in the AIF for expressing parallelism and subsumes task-parallelism primitives such as `fork` and `join`.

Continuation calls take a single argument This requirement is an artifact of the library implementation and could be hidden in a compiler implementation via a compile-time transformation. In cases where multiple arguments would be used by a C++ member function, it is sufficient to wrap the argument vector in a minimal C structure. The type of the single argument is specified by the template argument of the `ActorMethod` base class. For example, the `JacobiBlock::boundary` method is derived from `ActorMethod<JacobiBoundary>`. `JacobiBoundary` is an ordinary C++ class containing the boundary values exchanged between `JacobiBlock` actors.

Continuation calls do not return a value The lack of return value is a result of the asynchronous nature of continuation execution. This restriction is unavoidable in a strict actor paradigm but may be loosened by using a compiler which performs continuation-lifting to implement what is essentially a remote-procedure call (RPC).

Beyond these characteristics, `ActorMethod` execution is virtually synonymous with member function execution. When an `ActorMethod` invocation is scheduled to run on a particular processor, the member function of the same name is invoked by the run-time system with the datum passed to the continuation call as the function parameter.

The implementation of the `ActorMethods`, `Continuations`, and `Messages` is handled completely within the library; the developer sees only the type-safe interface. Discussion of implementation details would be lengthy and would add little to this discussion.

3.2.5. NewActorMethods

`NewActorMethods` implement the Actor `new` primitive; for actor types, they fill the role of a C++ *new-expression*[31]. Creation of actors typically differs from creation of common C++ objects in that actors are usually distributed over available processors to maximize resource utilization.

Declaration and usage of `NewActorMethods` is similar in most respects to that of `ActorMethods`. `NewActorMethods` differ from `ActorMethods` in that there is no target actor provided when a new actor continuation is created. Instead, a set of options which will control the way that the actor is eventually created may be specified.

The constructor for `NewActorMethod` continuations takes two optional parameters, the location (processing element) where the actor will be created and the name of the new actor. While the meaning of the location parameter is intuitive, the use of the optional name parameter is less so and most easily explained via an example. In the case where a location is not given, the run-time system selects a location to maximize load balance. The only effect of not providing a name is the inability of the creator to communicate with the new actor.

In the Jacobi application, a `UserInterface` actor reads in all necessary information, decomposes the problem into pieces, and then creates one `JacobiBlock` actor to solve each subproblem. Each `JacobiBlock` actor needs to be able to communicate with up to four neighbors in order to exchange boundary values and to do this, it needs to know their names. As mentioned previously, an actor which creates another can determine the name of the new actor, so the `UserInterface` object could be expected to know the names of all the instances it has created and therefore it could send those names to any object it subsequently creates. However, in the Jacobi example, this is not enough; `JacobiBlock::boundary` actors need to know the names of all neighbors which is not possible when actor creation is performed sequentially.

To alleviate this problem, the ProperCAD II library provides an *actor name server* which allows the asynchronous allocation and binding of actor names. Taking advantage of this, the `UserInterface` actor preallocates all the names it will need and then binds these names to individual actors as they are created. The code, slightly simplified for clarity, is:

```
ActorName<JacobiBlock>** names =
    new ActorName<JacobiBlock>[ nJacobi ][ nJacobi ];
for( i = 0; i < nJacobi ; i++ ) {
    for( j = 0; j < nJacobi ; j++ ) {
        // actor name server operates behind the scenes to allocate
        // names for the newName call
        names[ i ][ j ] = ActorName<JacobiBlock>::newName();
    }
```

```
}
for( i = 0; i < nJacobi; i++ ) {
    for( int j = 0; j < nJacobi; j++ ) {
        JacobiBlock::New create ( names[ i ][ j ] );
        create( names[ i-1 ][  j  ],      // up
                names[ i+1 ][  j  ],      // down
                names[  i  ][ j-1 ],      // left
                names[  i  ][ j+1 ],      // right
                ... );
    }
}
```

In this case, the continuation is created with an optional parameter, a name that will be bound to the actor when it is created. Since creation and name binding are no longer causally related, construction is simple. The library provides the necessary communication to spread actor name information among involved processors and uses an as-needed protocol to minimize communication. The actor name server is also responsible for tracking name usage, assuring that continuations called on unbound names are queued until the actor is created. The name server itself is implemented as an aggregate.

3.2.6. Aggregates

Though the actor paradigm is in some sense complete, it is not necessarily the most convenient way of describing parallel problems. There are many higher-level constructs which are not easily mappable to a single actor. Moreover, since continuation execution is synchronized, routing all requests through a single actor serves to reduce concurrency rather than increase it.

To alleviate this synchronization, aggregates were introduced [12]. An aggregate is a collection of one or more actors which share a common name. Messages sent to an aggregate name are directed by the run-time system to a member actor or *representative*. In contrast to actors, which always exist on exactly one processor, aggregates may have representatives distributed across multiple processors. This makes it possible for other actors and ordinary C++ objects to directly execute member functions on aggregate instances, in those cases where a local representative exists; such access is facilitated by the run-time library. Application aggregate types are derived from the library class `Aggregate`.

An example of the use of aggregates in representing distributed data structures is shown in Figure 3.3. The abstraction in the figure is a distributed array where distinct ranges of elements are located on different processors. The figure shows two possible implementations; an actor implementation and an aggregate implementation. In the actor implementation, a gateway actor is required (since client code only knows one name) which must redirect messages to the appropriate sub-range actor. This is inefficient in that it requires an extra indirection and because the redirection process is serialized. Alternately, in the aggregate implementation, since all representatives in the aggregate share the same name, there is no need to send all requests to a single actor. Instead, a message can be sent to any of the representatives where it may be either serviced immediately or forwarded to another representative to be serviced. The ProperCAD II interface allows the aggregate designer to chose among a number of standard representative selection mechanisms or to provide a custom selection mechanism. In the figure, serialization of access to the array would only happen if two clients needed to access an element stored in the same representative.

The constructor for a `NewActorMethod` continuation for aggregates takes an additional optional parameter beyond those described for actors. The additional parameter is a *distribution*

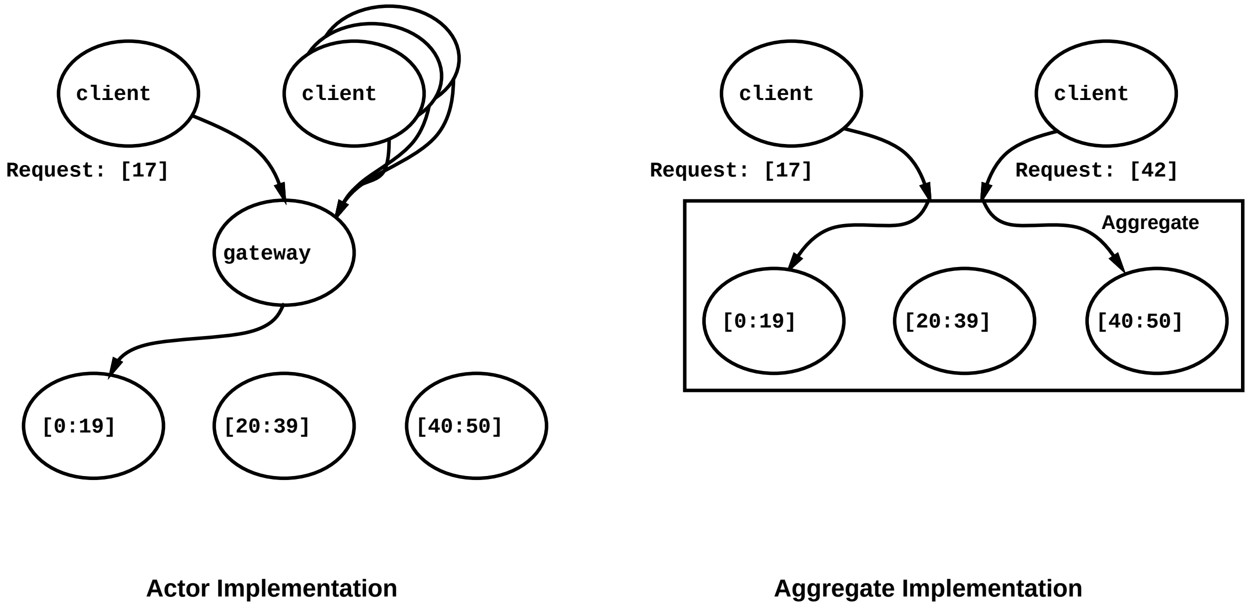

Figure 3.3 Actor and Aggregate Implementation of a Distributed Array

object which specifies the number and location of the representatives comprising the aggregate.

3.2.7. `Distributions`

Distribution of aggregate representatives is specified via a distribution instance, an object which specifies the number and location of the representatives of an aggregate. The interface of the abstract `Distribution` class is general enough to represent both enumerated and algorithmically-computed distributions. The library provides distribution classes which facilitate the most commonly used sharing abstractions found in medium-grain applications.

3.2.8. `AggregateNames`

The library provides `AggregateNames` which provide extra support for intra-aggregate addressing. The library supports intra-aggregate communication by allowing aggregates to send messages to representatives by index. Aggregate communication is also extended to support broadcast continuations which, when called, send identical messages to all actors within the aggregate. A significant feature of `AggregateNames` is their interchangeability with `ActorNames`. Although intra-aggregate addressing functionality is only provided via the static `AggregateName` interface, an `AggregateName` can be turned into an `ActorName` without loss of information. Thus client code is usually independent of whether the object being communicated with is an actor or an aggregate; in all cases characteristics of continuations, such as broadcasts, are preserved.

3.3. Meta-programming Interface

In many cases, the actor model is sufficient for describing problems. However, certain characteristics of the actor model, such as unordered message delivery, occasionally have a significant impact on algorithm performance. In such cases, it is desirable to have an application-visible interface which facilitates programming the underlying actor run-time system. Such programming of the library itself is called meta-programming.

3.3.1. System Aggregates

The run-time support of the actor interface is meta-circularly implemented via a number of library-supplied aggregates, brief descriptions of which follow. Since the system aggregates are themselves simply aggregate classes, they can be used as base classes for application-specific types, in which case the application-type may modify the semantics provided by the library. Additionally, the system aggregates have member function interfaces which may be used to affect run-time characteristics without the need to resort to derivation.

`Director` The main controller of each physical thread in an actor program is a representative of the `Director` aggregate. The `Director loop()` method is invoked on each physical thread as it is created. The `Director` representative is responsible for sending and receiving messages and for delivering messages to individual actors.

`NameServer` The `NameServer` aggregate is responsible for coordinating actor and aggregate names within the system. This coordination includes allocating names which are unique across all processors, routing actor method calls to the appropriate processor for execution, and maintaining binding information as new actors and aggregates are created.

`LoadBalancer` Load balancing functionality is also implemented as an aggregate and the `Director` aggregate uses services of the `LoadBalancer` aggregate to effectively utilize processor resources. Currently the library implements only a random placement technique for load balancing; the addition of more sophisticated techniques is planned. Load balancing is of special interest because experience in the first phase of the ProperCAD project showed that problem-specific load balancing was often required resulting in significant duplication of effort[10]. It is hoped that an aggregate-based approach will allow more of the load balancing functionality to be placed in the library, reducing developer effort.

`QuiescenceDetector` The `Director` aggregate coordinates its action with a quiescence detection aggregate to determine when the system is idle; i.e., there are no pending actor method calls. The current algorithm follows that of [42] but the notification interface has been generalized. Any actor can pass a `Continuation<Void>` instance to the quiescence aggregate, which will in turn invoke the continuation when quiescence is detected.

3.3.2. Task Queues and Priorities

The `Director` maintains all ready-to-run actor method invocations on a set of task queues, prioritizable by a library-defined, user-extensible priority system. Several factors influenced our decision to add priorities to the ProperCAD II library, even though a pure Actor paradigm has no equivalent abstraction. Since the `Director` aggregate uses the same message passing mechanisms for intra-aggregate communication as do the user-level actors and aggregates, a method is required to ensure that certain time-critical `Director` information is processed expeditiously. Once this conclusion was reached, it was relatively easy to expand the abstraction to a user-level service. Several CAD algorithms have been shown to be more easily expressible when some sort of ordering is applied [9, 11, 43]. Though priorities have been used in Charm [21], the priorities implemented in the ProperCAD II library differ in that they are *hierarchical* and *extensible*. The developer may add new priority types to the library as needed without in anyway affecting the prioritization of other classes of messages. This is required in order to support the composition property of object-oriented systems.

4 Abstract Parallel Architecture

4.1. Overview

The Abstract Parallel Architecture (APA) is a model of a parallel computer and a set of objects that represent a reification, or concrete, program-visible implementation, of the model. The APA provides abstractions for thread, memory, and communication management and has been architected to provide a standard interface to common resources across all platforms with no significant overhead. To that end, the interface has been factored in such a way that code is shared to the maximum extent possible within the zero-overhead requirement. This factoring facilitates porting by minimizing the amount of code that must be changed to support new operating system interfaces. The APA is self-sufficient and has been designed to be usable apart from the actor interface.

The APA can be divided along logical and physical lines. The physical classes are primarily involved with implementation aspects of the library and include the `Machine` and `Network` classes. The logical classes represent implementation-independent resources such as threads and datagrams. The application designer usually deals solely with the logical classes without concern for the type of machine represented by a particular thread. Similarly an individual responsible for porting the library to a new machine architecture or network protocol is usually concerned solely with the physical classes. An area of future extension is a third APA dimension, performance, which provides a mechanisms for describing the relative processor speed and network bandwidths.

The APA is broken into four interrelated subsystems. Three of these comprise the logical interface; the fourth is the physical interface. The three logical interfaces are thread management, address space management, and communication management.

4.2. Thread Management

The APA manages a set of physical threads, represented by instances of the `Thread` class. `Thread` objects are collected into sets characterized by the means by which the processors they represent are interconnected. These sets are represented by the library classes `Process`, `ProcessGroup`, and `Cluster`. The relationship between the classes is depicted in Figure 4.1.

`Threads` The fundamental unit of computation is the `Thread`, defined as an allocated processing element. A thread may or may not represent individual processors; under operating systems with

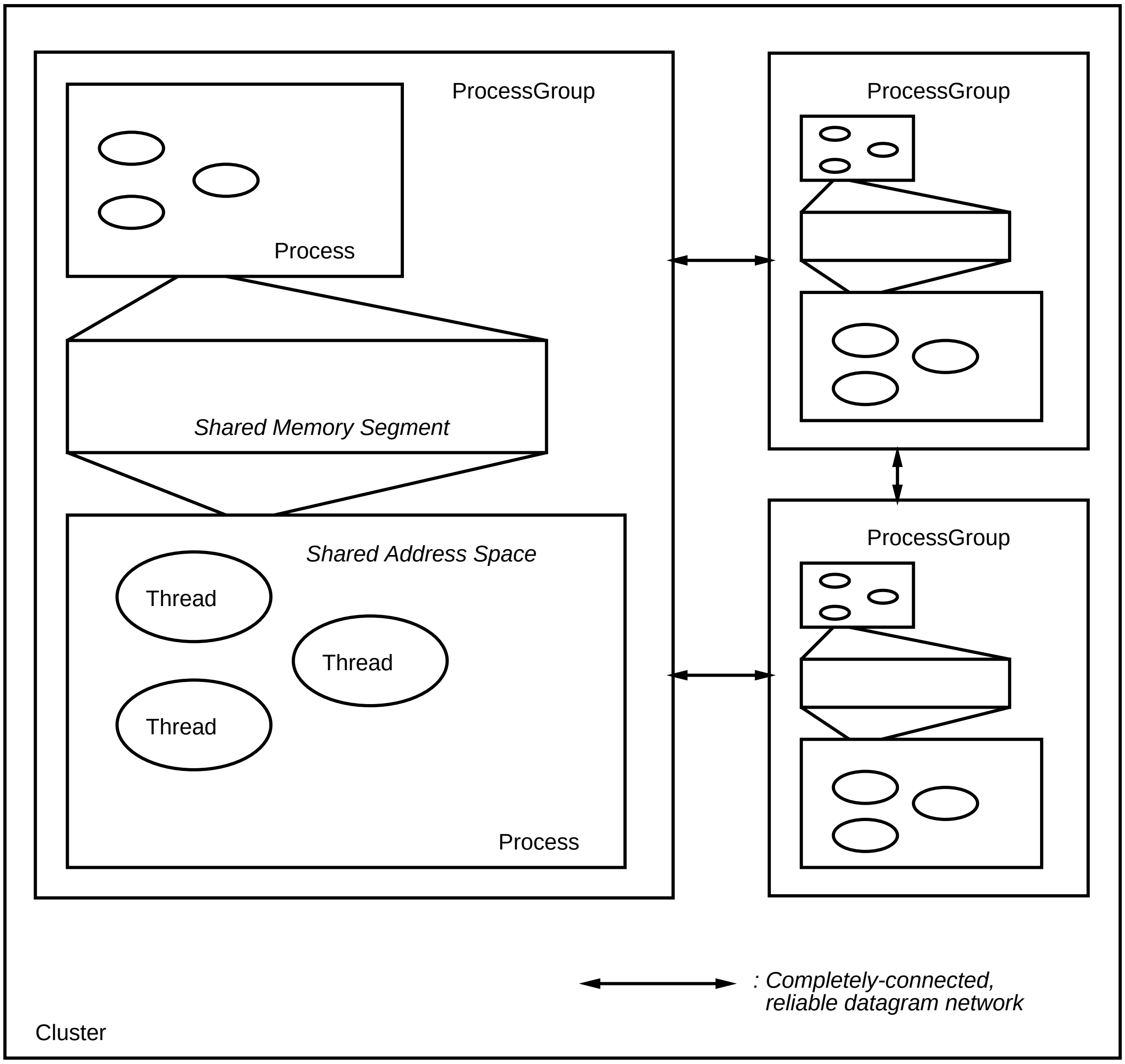

Figure 4.1 APA Thread Management Classes

Machine	APA Configuration
Figure 4.1 architecture	3/2/3
16 node INTEL iPSC	16/1/1
4-processor Sun 4/690MP	1/4/1
Two 4-processor Sun 4/690MPs connected by Ethernet)	2/4/1
INTEL 64 fat node Paragon	64/1/4

Table 4.1 APA triples (`ProcessGroups/Processes/Threads`) for various machines.

multiprocessing support, a thread represents an operating-system scheduled entity.[1]

`Processes` `Threads` that share a complete address space are collected into a container called a `Process`. Since all `Threads` in a `Process` share the same address space, all pointers are valid across `Thread` boundaries within a `Process`. This model is applicable, for example, to Mach threads.

`ProcessGroups` `Threads` which share at least a portion of their address space are collected into a container called a `ProcessGroup`. Since all `Threads` in a `ProcessGroup` do not necessarily share identical address spaces, pointers may not be valid across `Process` boundaries; addresses are guaranteed to be valid if and only if they point into a shared memory segment. This model is applicable, for example, to Unix `fork`-based multiprocessing systems.

`Clusters` All threads in a program are collected into a `Cluster` object. To the user of the APA, it appears that `ProcessGroups` are interconnected by a completely-connected, reliable, unordered, datagram network. All `Threads` in the APA can communicate in some way; either through shared memory if they are in the same `ProcessGroup`, or via a reliable, datagram network if they are not.

The utility of this architecture can be demonstrated by considering the mapping of contemporary architectures to the model. Table 4.1 shows the number of each type of thread management object for a number of common parallel architectures. The first number in each triple represents the number of `ProcessGroups` per `Cluster`, the second the number of `Processes` per `ProcessGroup`, and the last the number of `Threads` per `Process`; the total number of threads can be calculated by forming the product of the triple elements. Of particular interest are the last two rows in the table, an IP-connected pair of Sun multiprocessors and an INTEL Paragon with *fat nodes*.[2] In contrast to most previous work, the APA can represent machines which are hybrids of message passing and shared memory architectures. Though not shown in the table, the APA is not restricted to architectures that can be expressed by triples of the form above (e.g. uniform numbers of `Processes` per `ProcessGroup`.)

In terms of managing these resources, the APA provides standard interfaces for several important functions. During initialization and deinitialization, the APA provides for the spawning

[1] Having more threads than processors provides no performance improvement but is useful when debugging a program on a uniprocessor workstation.

[2] The Paragon is a mesh-connected multicomputer; when configured with fat nodes each node contains four processors connected by shared memory and running OSF/1.

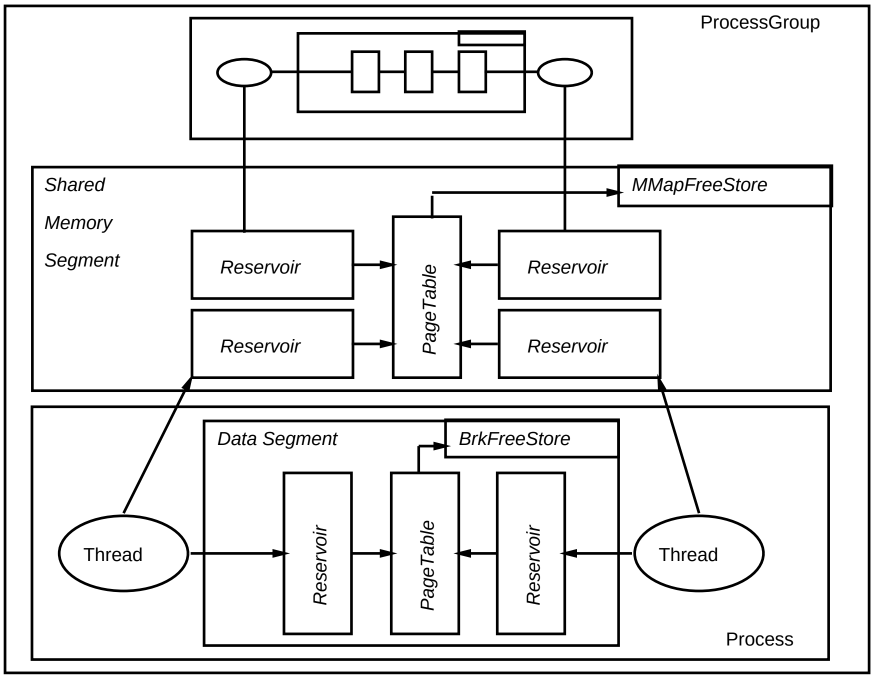

Figure 4.2 APA Free Store Management Classes

and reaping of objects at every level; e.g., for a `Process` which spawns and reaps a number of `Threads`. The APA also provides a static 'this' function for each type, e.g., `Thread::thisThread()`, which returns the object relevant to the executing thread of control.

4.3. Address Space Management

Free store (`malloc`) management is an area of common difficultly and little standardization. Most machines which supply a shared memory interface provide either little support for allocation of memory within shared memory segments or provide it in a very machine-dependent manner. `malloc` implementations are also notorious for having widely varying performance characteristics. The APA free store management classes were designed to address both issues. Figure 4.2 depicts graphically the relationship between the component classes, described below.

`FreeStores` The operating system interface to the free store hierarchy is contained completely within the `FreeStore` class. A `FreeStore` object manages a contiguous segment of memory in units of pages. `FreeStores` allocate, on demand, new pages of memory at the end of the current segment and can return pages at the end of the segment to the underlying source. `FreeStores` neither handle non-contiguous segments nor do they maintain lists of free pages.

`FreeStore` is an abstract class; concrete classes derived from `FreeStore` provide services for a particular operating system interface. The APA currently supports free stores for memory allocated using the UNIX `brk()` and `sbrk()` system calls (`BrkFreeStore`), for memory allocated using the UNIX `mmap()` system call (`MMapFreeStore`), and for memory allocated using the UNIX `brk()` call and shared using the Encore Multimax `share()` call (`ShareFreeStore`).

`PageTables` Since `FreeStores` do not support arbitrary deallocation or provide a thread-safe interface, another object is needed to manage lists of pages and which can be safely used in a multi-threaded application. The `PageTable` object provides this functionality. One `PageTable` is allocated for each `FreeStore` and provides an interface for allocating and deallocating arbitrary numbers of pages. It also supports concurrency on those architectures where it is available. A `PageTable` keeps an array of page descriptors which encode information on the state of each page, whether it is free or allocated and, if allocated, how it was allocated. This last feature is used by `Reservoirs`, described next. Although the `PageTable` has a few critical regions to prevent chaotic behavior, it has been optimized for performance. Since most accesses to the `PageTable` do not modify the state of the table, the `PageTable` object differentiates between read access and write access and allows concurrent access to multiple readers.

`Reservoirs` Since `PageTables` support only allocations in multiples of page sizes, an interface is needed to efficiently handle large numbers of small and medium size objects. The `Reservoir` fits this need. A `Reservoir` contains a number of `Pools` of equal-sized objects. Since most of the time there are unused objects in a pool, allocation for most cases is very fast.

Like `PageTables`, `Reservoirs` must be thread-safe. Because an object may be allocated by one thread, passed to another thread, and then freed by that thread, write access to the `Reservoir` must be synchronized. However, a thread will usually make heavy use of allocation and deallocation primitives so it is not desirable to require synchronization on every allocation and deallocation. Instead, we note that if we provide each `Thread` with its own `Reservoir`, all allocations will be to its private `Reservoir` and the vast majority of deallocations will be to its own `Reservoir`. This is the approach taken; synchronization is only needed in a few special cases of deallocation.

Figure 4.2 shows what the snapshot of the free store system might look like during the execution of an application on a machine with an APA triple of 1/2/2; i.e., two `Processes` communicating via a shared memory segment, each containing two `Threads` running in the same address space. Each `Thread` has two reservoirs, one which exists in the private address space of its `Process` and one in the shared segment, owned by the `ProcessGroup`. Each `Process` has one `PageTable` to manage its private address space. Each `ProcessGroup` has a `PageTable` to manage the shared segment. Each `PageTable` has a `FreeStore` object which provides the sources of pages.

The ProperCAD II library implements a number of data structures — variable-sized arrays, linked-lists, etc. — which understand the APA memory management system and arrange to create individual collection elements in the same free store as the collection object.

4.4. Communication Management

The final portion of the APA provides the interface necessary to support interprocess communication. The two abstractions provided are `Semaphores` and `Networks`.

Semaphore The Semaphore class provides a uniform interface to spin-lock style semaphores. ReadWriteSemaphores provide separate read and write locks; multiple read locks are allowed at one time while a write lock ensures exclusive access. In addition to the normal P and V operations, Semaphores provide a state-based interface based on Guards. Instances of a Guard class guarantee exclusive access for exactly their lifetime. For example, typically a PageTable member function locks the object for the duration of a member call. Rather than execute separate lock and unlock functions, a guard object is used. PageTable has several functions of the type:

```
PageTable::function(...) const
{
    ReadWriteSemaphore::ReadGuard guard ( accessSemaphore );

    // Since guard is live during the call to unsafeFunction, the state of
    // the PageTable will not be changed (or examined) by other threads
    // for the duration of the call.

    return unsafeFunction(...);
}
```

The guard object locks accessSemaphore in its constructor and unlocks it in its destructor. This eliminates the chance that a semaphore will be left locked.

For performance reasons, the necessary indivisible read-modify-write operation for Semaphores is implemented in assembly code on those machines for which it is necessary. Moreover, the Semaphore class is *not* an abstract class; a single implementation is chosen when the library is compiled. In this case, the overhead of dynamic binding is of particular concern.

Network The Network class provides an abstract interface supporting reliable, unordered, datagram communication among ProcessGroups. The interface to Networks is very simple, consisting of send, broadcast, receive, and isReady members. Each concrete, physical class is responsible for implementing these functions along with any necessary local data.

The Network class provides support for marshaling and unmarshaling; the process of turning an object into a datagram and back into an object. The information required to perform this function is provided by the library for simple and collection classes and by the developer for complex types. Automating this process is a subject of future research.

4.5. Physical Interface

The Machine and Network classes provide abstract interfaces to underlying hardware and have interfaces sufficiently flexible and recursive to describe almost any interconnection of machines. The primary responsibility of the Machine class is the management of machine configuration. The Network classes work together to provide a reliable, unordered, complete, datagram interconnection.

Current machine classes are the Machine abstract base class, IPSCMachine for INTEL iPSC and Paragon computers, UnixMultiprocessedMachine for Unix machines based on multiple processes sharing a segment of memory, i.e., the Sun multiprocessors, and UnixMultithreadedMachine for Unix machines supporting multiple threads within an address space, i.e., the Encore Multimax and Mach-based operating systems. A special class, CompositeMachine does not reflect any particular physical machine but instead a composite machine made up of separate physical machines joined by a network. Because all machine classes have the same interface, client code is not dependent upon the particular machine architecture.

Current network classes are the `Network` abstract base class and the `IPSCNetwork` for INTEL iPSC and Paragon computers. The `Network` class currently provides both a logical and physical interface. `Networks` may also be hierarchical, as would be required for two Ethernet-connected Paragon machines.

The development of the physical interface is still underway so a detailed interface description is pending.

5 Performance

An implementation of the Gauss-Jacobi iteration for solving partial differential equations was created to explore the performance of the library. Four implementations were created: serial, a straightforward serial implementation, shared-memory, a version applicable to shared memory machines, message-passing, a version applicable to message passing machines, and actor, an actor implementation based on the run-time library. All implementations except for actor were implemented in C using the most efficient method for the target architecture. The actor implementation is based on the actor model and thus represents a message passing style even on shared memory machines. Since the library is written in C++, it includes some inefficiencies reflecting the relative lack of maturity of C++ compilers. The same actor code is used on all architectures.

Figures 5.1 and 5.2 show the performance of 100 iterations of the Jacobi computation across a range of problem sizes on a message passing machine, the Intel iPSC/2, and a shared memory machine, the Encore Multimax. The data presented represent speedups over the serial algorithm, serial. The results show that the performance of the actor application lags the custom versions, especially for small problem sizes but approaches the optimum for larger problem sizes.[1] For sufficiently large grain-sizes, the actor application demonstrates little performance penalty and near optimal speedup for all cases. The bulk of the difference between the custom C and actor C++ implementation is due to overhead in the non-parallel elements of the C++ library; the library is currently undergoing optimization of these elements.

[1] A small super-linear speedup is seen on the uniprocessor versions of the shared memory and message passing codes; this is tentatively attributed to small changes in memory allocation.

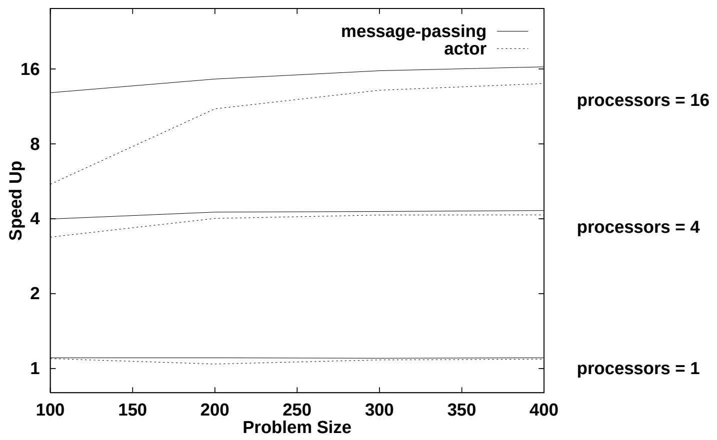

Figure 5.1 Speedup over optimized C code on iPSC/2

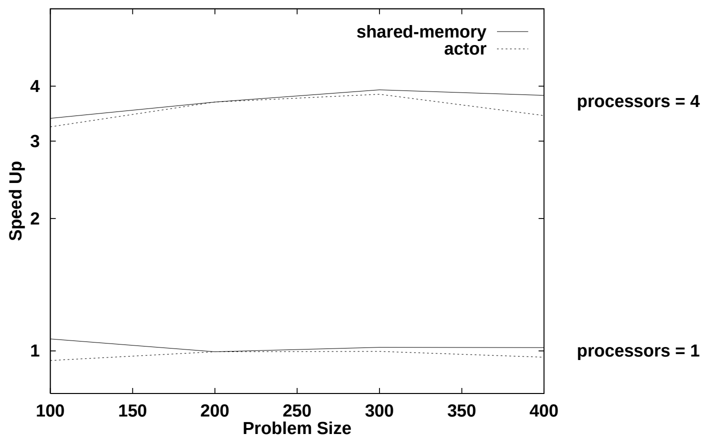

Figure 5.2 Speedup over optimized C code on Encore Multimax

Part II

Applications

6 Automatic Test Pattern Generation

Automatic test pattern generation (ATPG) involves the generation of a set of inputs to an integrated circuit that, when applied to a fabricated device, will indicate if a defect occurred during manufacturing. ATPG is an NP-complete search problem and represents a significant portion of the design effort for new devices. The `HITEC/PROOFS` package for sequential test generation[44] exhibits performance and quality among the best known and thus was chosen as the basis for the development of a parallel test generator.

6.1. Related Work

Most popular serial algorithms for test generation are based on the PODEM algorithm[45]. Several approaches have been proposed for parallelizing test pattern generation, most of which involve modification of the PODEM algorithm.

Fault-parallel *Fault-parallelism* refers to the generation of test patterns for a given fault set in parallel. In this method, the fault set is divided equally among available processors, each processor generating tests for its fault set independently. Such a scheme has been proposed by Chandra and Patel [46], Patil and Banerjee[47, 48], Patil, Banerjee and Patel[49], and Agrawal *et al*[50]. The main advantage of fault-parallelism is that communication overhead is very low and, as such, it is possible to achieve linear speedups when the number of processors is very small compared to the number of faults. The main disadvantage is that faults which are hard to detect by the serial algorithm (i.e., faults which require a large number of backtracks) remain undetected even on the parallel implementation. Additionally, since most test generation systems now use fault simulation to capitalize on the serendipitous detection of multiple faults by the patterns generated for a single fault, when fault-parallelism is used, speedups can fall due to one processor expending useless effort to detect a fault which will be serendipitously detected by the vectors generated for another.

Decision-parallel *Decision-parallelism* refers to the evaluation of the functions associated with several decision alternatives in parallel. In this respect, techniques to parallelize ATPG borrow from the parallelization of pure depth-first-search, examples of which appear in [51, 52, 53, 54]. This technique was proposed in [55], but the search space allocation strategy did not utilize heuristics to increase the probability of searching in a solution area. A parallel branch and bound algorithm was proposed by Patil and Banerjee [56, 57] that is based on searching different portions of the search space concurrently. A similar parallel algorithm for combinational test generation, suitable for execution on a network of workstations, was proposed by Arvindam *et* *al*[58]. Recently, a parallel algorithm for ATPG on sequential circuits has been proposed by Patil, Banerjee and Patel [49]. The parallel algorithm, which is suitable for execution on shared memory multiprocessors, uses a variation of decision-parallel functional decomposition. Ramkumar and Banerjee[43] used Charm[21] to create a parallel version of [49] which used both fault-parallelism and decision-parallelism. The work presented here adopts many of the techniques of their work.

Heuristic-parallel Almost all test generation algorithms use a set of heuristics to guide test generation. Experiments reported in [59, 60] suggested that there is no clear cut advantage of using one heuristic over the other. One can therefore exploit *heuristic parallelism* by letting each processor use a different heuristic to guide the search for the same fault. Chandra and Patel[46] reported results on a parallel algorithm for test generation of combinational circuits using heuristic decomposition. The main disadvantage of this method is that the parallelism is limited by the number of heuristics available for search, which is usually no more than five or six. Also, when using different heuristics there is no guarantee that search spaces are disjoint, which may lead to redundant work. Finally, no improvement is possible if a fault remains undetectable for all the heuristics. The main advantage of this method is that communication overheads are low.

Partition-parallel Another approach to parallel test generation is based on circuit decomposition or *partition-parallelism*. In all the other parallel approaches, each processor has a copy of the entire circuit. For extremely large circuits, the memory of each processor may not be able to store the entire circuit. In a circuit decomposition approach, each processor keeps a partition of the circuit and performs backtracing operations on its own subcircuit to satisfy various test generation objectives[61]. It has proven to be extremely difficult to achieve efficient speedups using this approach.

6.2. ProperHITEC

The `HITEC/PROOFS` package for sequential test generation and fault simulation exhibit performance and quality among the best known in the field[44]. Following the premises of the ProperCAD project, it was chosen as the basis for the development of an efficient and effective parallel test generator. `HITEC` is a descendent of and an extension to the PODEM approach. The extensions, roughly in decreasing order of significance, are: Targeted D-Frontier, enhanced state justification, fault simulation, and variable time frame processing.

The overall organization of the original `HITEC/PROOFS` package is shown in Figure 6.1. In the original, `HITEC` and `PROOFS` run as separate programs, communicating via a TCP/IP socket.[1] The original applications are written in C++ using what might be called an 'object-based' design style. In particular, the object-oriented concept of encapsulation is only weakly represented and the use of dynamic-binding is omitted. Table 6.1 provides a brief description of the major classes in `HITEC`.

The approaches used to parallelize `HITEC` are fault-parallelism and decision-parallelism (see Figure 6.2). Implementation of fault-parallelism is relatively straightforward, since test generation for different faults is by-and-large independent. The use of decision parallelism was deemed

[1]Even though the two programs run in parallel, the synchronous nature of the communication eliminates any possibility of parallelism.

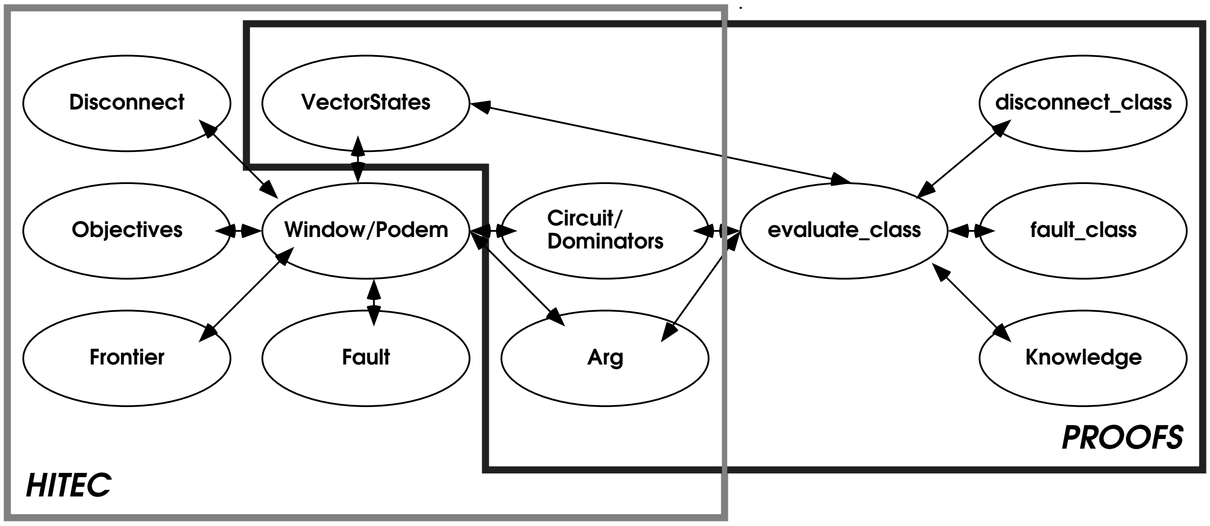

Figure 6.1 HITEC/PROOFS Organization

Class	Description
Circuit	circuit connectivity, observability, and controllability
Dominators	precalculated dominator information for all nodes
Arg	arguments to HITEC/PROOFS
Disconnect	object used to perform fault injection
Window	time window representing circuit state for all active time frames
Fault	fault status database
VectorStates	data base of test vectors and resulting state from PROOFS
Objectives	list of justification objectives
Frontier	list of D-Frontier nodes
Podem	wrapper of core podem() and state_justify() routines

Table 6.1 Classes in HITEC

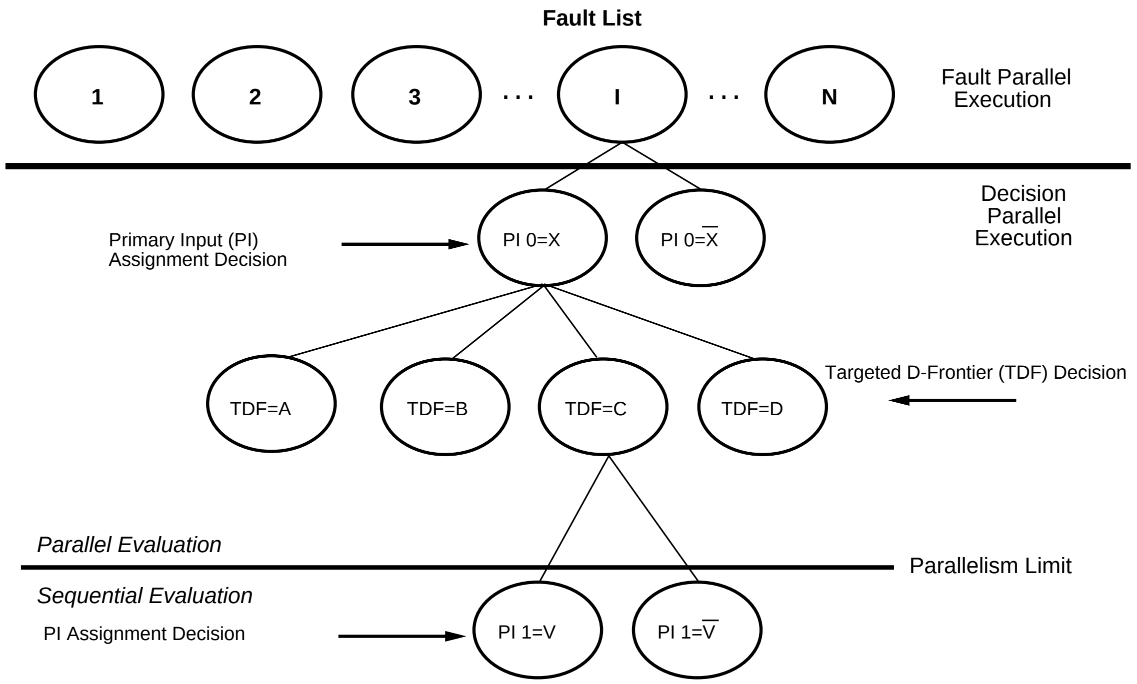

Figure 6.2 Parallelism in `ProperHITEC`

necessary to achieve high efficiency on many processors while maintaining quality of results. Quality is higher for decision-parallel execution because the execution trace more closely matches the serial version when decision and fault-parallelism are combined. In strictly fault-parallel execution, all but one processor are working on a different fault than the serial algorithm; if additional faults are covered by patterns generated by a previous fault, all work done in generating tests for those faults is wasted. Moreover, since test generators generally spend most of their time generating tests for a relatively small number of hard faults, even with fault-parallelism, execution time is bounded on the low-end by the time required to test the hardest fault. Decision-parallelism explores different areas of the search space in parallel and thus for cases where a large portion of the search space must be explored, can provide significant speedup. Note that by casting the ATPG search in such a general framework, it is easy to enable purely fault-parallel, purely assignment-parallel, or assignment and D-Frontier-parallel execution (`HITEC`), allowing extensive experimentation.

Only one significant change was required to `HITEC` heuristics to implement decision-parallelism. The change stems from the ability to pass information up the search tree in serial depth-first search. When a backtrack occurs in a depth-first search, information gained in detecting the backtrack can be passed up the search tree to influence the backtracking process itself. `HITEC` utilizes such information when a state-justification failure occurs. When such a condition occurs, a `backtrack-must-change-state` flag is set indicating that any decision alternatives that do not change the state-justification target should be backtracked immediately. When a decision alternative results in a different state for justification, normal search resumes.

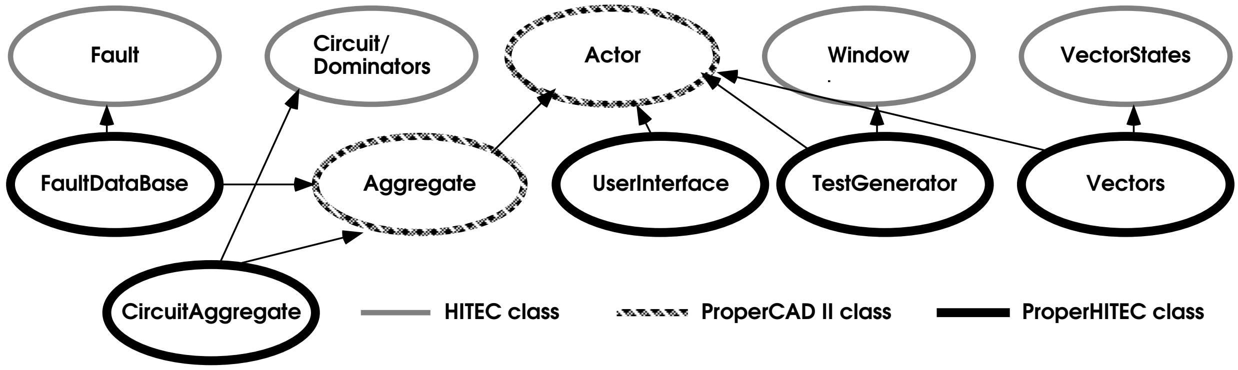

Figure 6.3 `ProperHITEC` Organization

When decision-parallelism is used, a breadth-first-like search is employed. Since decision alternatives no longer execute strictly in sequence, there is no way to pass the `backtrack-must-change-state` flag among them. Fortunately, this information is closely related to the failed states array already maintained by the fault database. In fact, it is easy to show that for those cases where the `backtrack-must-change-state` flag is set, the backtracked state must be in the failed states array. Checking the failed states array is actually more accurate, since a backtrack that does change the state may simply change it to another failed state, a condition which the original `HITEC` algorithm does not detect. Since the amount of processing required to check the failed states array is more than that required to simply to test a flag, the choice of which technique to use is dependent upon the relative amount of time required to check the array versus that required to perform the useless work of exploring non-solution alternatives. Informal testing of the original `HITEC` using both heuristics showed the cost of useless searching to be much greater than the cost of checking the failed states array and thus `HITEC` and `ProperHITEC` have been modified to use the better heuristic.

`ProperHITEC` uses lexicographically-ordered bit-string priorities to guide the execution of `ProperHITEC` as closely as possible to the order used in the sequential algorithm. This technique is essentially the same as described in [43]. We note that in the ProperCAD II library, the use of bit-string priorities for the test generator objects has no effect on the priority system used by the other objects in `ProperHITEC` or in the library in general.

Figure 6.3 shows the most significant objects in the `ProperHITEC` implementation along with the `HITEC` and ProperCAD II objects from which they are derived.

`TestGenerator` The `TestGenerator` object is an actor which represents a 'test generator machine.' The implementation is relatively straightforward; `TestGenerator` instances can be created for a specified fault to implement fault-parallelism or can be cloned from an existing instance and an alternate assignment to implement decision parallelism. To implement an efficient algorithm, fault-parallelism is bounded by a user-specified limit, past which decisions are made in a depth-first manner.

Each `TestGenerator` new actor continuation is assigned a priority which the underlying run-time system uses to execute the `TestGenerator` objects in an order as close as possible to the serial algorithm. When run on a uniprocessor, the `TestGenerator` objects are executed in exactly the same order as the serial algorithm; on one processor, `HITEC` and `ProperHITEC` produce virtually identical results.

CircuitAggregate The circuit is implemented as an aggregate with one representative on each processor. A limited form of sharing for shared memory machines (supported completely by the library) was considered but discarded because the `HITEC/PROOFS` implication procedures are based on fault-injection through circuit modification. Since only one `TestGenerator` object is active at a time on a processor, it is granted write access to the local `CircuitAggregate` representative. When the `TestGenerator` instance completes its work, it restores the circuit to its original state.

FaultDataBase The fault database is implemented as an aggregate with a representative on each processor. Each representative stores the most recent state of the test generation process for each fault and provides the same interface to the test generator objects as the serial `Fault` object. When an update is received from a `TestGenerator` object or from the vector/states database, in the form of a change of status, notification of resource usage, or detection of a new failed state, the `FaultDataBase` representative records the information locally and if this results in a change of the local state, broadcasts that information to all other representatives. The broadcast operation is implemented completely via ProperCAD II intra-aggregate addressing.

Vector The `Vectors` object is essentially the same as the serial object but uses `ActorMethods` to record new vectors and to send results to the `FaultDataBase`. Currently the fault simulator is the `FaultSimulator` object used in `HITEC` using the actor methods provided by the `Vectors` actor. This implies that fault simulation occurs sequentially on a single processor. This was done both for reasons of expediency and because test generators have been observed to spend little time in fault simulation relative to time spent in test generation. Parallel fault simulation in the ProperCAD II library is a topic of future interest.

UserInterface The `UserInterface` object is used to interact with the user during the running of the test application. It creates the system objects and then creates test generator objects for each undetected fault in the circuit. If the progressive time limit feature of HITEC is used, the process of creating test generator objects is iterated using progressively larger time limits.

6.3. Performance

`ProperHITEC` runs on all platforms supported by the library. Results are presented on the Sun, iPSC/860, and Encore; the memory on the nodes of the INTEL iPSC/2 is too small to run all but the smallest circuits.

Figures 6.2, 6.3, and 6.4 show the results of `ProperHITEC` for a number of circuits drawn from the ISCAS-89 benchmark set. All times, T, are reported in seconds and represent the elapsed wall clock time. Fault efficiency, E, is computed as efficiency $= \frac{(\#\text{faults}-\#\text{aborted})}{\#\text{faults}}$. V is the number of

Circuit/ Seconds Per Fault	HITEC			ProperHITEC Processors 1			2			3			4		
	T	E	V	T	E	V	T	E	V	T	E	V	T	E	V
s344/20	369.4	95.9	121	374.3	95.9	121	251.9	96.2	110	160.0	96.5	130	156.2	96.2	112
s820/20	435.9	99.3	956	396.8	99.3	956	225.4	99.3	1010	196.3	99.1	1059	140.3	99.1	1013
s953/20	125.8	100	20	134.2	100	20	71.37	100	12	64.17	100	16	47.24	100	12
s1238/2	13.13	100	386	21.64	100	386	15.15	100	390	13.31	100	405	16.18	100	385
s1494/20	722.0	99.1	1058	663.4	98.9	1058	434.1	98.9	1123	350.5	98.9	1153	240.1	99.1	1093

Table 6.2 Results on the Sun 4/670MP

Circuit/ Seconds Per Fault	HITEC			ProperHITEC Processors 1			2			4			8		
	T	E	V	T	E	V	T	E	V	T	E	V	T	E	V
s344/20	481.4	94.2	89	485.8	94.2	89	215.8	96.8	105	194.5	96.8	112	142.1	96.5	102
s820/20	438.3	99.3	959	440.8	99.3	959	270.4	99.2	958	158.0	99.3	951	108.0	98.9	1034
s953/20	140.2	100	14	147.7	100	14	89.13	100	14	49.00	100	24	28.66	100	14
s1238/1	14.15	100	374	23.29	100	374	14.69	100	383	12.16	100	369	11.12	100	402
s1494/20	819.8	99.0	1079	821.3	98.7	1079	503.3	99.1	1168	310.1	98.6	1113	192.2	98.8	1151

Table 6.3 Results on the INTEL iPSC/860

test vectors generated. Higher fault efficiencies and lower numbers of generated vectors represent better solutions.

For each circuit, the results of the sequential HITEC algorithm and the ProperHITEC algorithms on various machine configurations are reported. The HITEC numbers presented are for the version of HITEC that shares code with ProperHITEC. Although the current version of HITEC takes more advantage of dynamic memory allocation, the amount of time spent doing memory management has been carefully analyzed and shown to be less than a fraction of a percent.

The results show that ProperHITEC achieves consistent speedup with only marginal effect on quality across a range of moderately difficult test problems. The major effect on quality is the addition of a small amount of noise in the results; while ProperHITEC does not always achieve results identical to HITEC, the number of cases where it does worse are on par with number of cases where it does better.

The results show that ProperHITEC does not achieve particularly good results on the 'easy' benchmarks, those for which 100% efficiency is achived by the serial algorithm in a few tens of seconds on current microprocessors. The reason for this is known; if the very first attempt at generating a test virtually always succeeds, the decision parallelism in ProperHITEC is not useful. We are considering techniques for improving the response of ProperHITEC for these cases.

Circuit/ Seconds Per Fault	HITEC			ProperHITEC Processors 1			2			4			8		
	T	E	V	T	E	V	T	E	V	T	E	V	T	E	V
s344/20	484.2	93.9	105	493.2	93.9	105	274.3	95.6	95	167.0	95.6	85	131.2	94.7	108
s820/20	1200	97.8	891	1206	98.1	891	761.8	97.6	1008	418.9	97.2	959	255.6	96.9	955
s953/100	572.4	100	20	597.1	100	20	343.3	100	18	252.1	100	14	166.7	100	10
s1238/10	65.41	100	386	97.49	100	386	60.53	100	382	53.83	100	382	55.35	100	406
s1494/10	2615	87.0	492	2920	84.2	402	1654	85.5	460	997.0	83.9	497	540.2	85.5	510

Table 6.4 Results on the Encore Multimax

Machine	Seconds Per Fault	# of Processors	T	E
Sun MP	1	1	707.0	82.2
	2	2	230.7	97.1
	3	3	222.0	97.6
	4	4	185.4	98.1
iPSC/860	2	1	777.4	79.5
	3	2	335.0	94.4
	8	4	212.7	97.8
	20	8	192.2	98.8
Multimax	2	1	2069	42.0
	4	2	1465	67.1
	8	4	1047	77.4
	20	8	540.2	85.5

Table 6.5 Results on Increased Efficiency on s1494

6.4. Fault Efficiency Results

In addition to providing faster turnaround, parallel processing can be used to achieve higher fault efficiency in a fixed amount of time. Table 6.5 shows the results of running test generation on s1494 where the time-per-fault limit was scaled with number of processors. Consistent improvement is observed for all platforms. Of note is the fact that even though the time limit per fault was raised in tandem with the number of processors, run times still decreased on parallel runs. This is due to the fact that once a fault is detected, raising the per-fault time limit does not increase run time.

7 Standard Cell Placement

When the logic design of a VLSI circuit has been completed, automatic placement is performed to choose locations for each cell such that the area of the layout is minimized. With chips approaching tens of millions of gates, the complexity of this problem is enormous, and moreover, the most popular technique for placement, simulated annealing, is inherently expensive. Using the ProperCAD II library, we have constructed `ProperPLACE`, an efficient parallel placement implementation based on an existing simulated annealing placement tool, `TimberWolf` 6.0 [62].

7.1. Simulated Annealing for Placement

Simulated annealing is a powerful technique that has been employed to solve combinatorial problems using a probabilistic hill-climbing algorithm with the added ability to escape from local minima in the search space. Given a problem and an associated cost function, the objective is to find a near-optimal solution by generating possible solutions and selectively accepting or rejecting them. If a candidate solution reduces the cost function, it is always accepted. Unlike greedy methods, however, simulated annealing may also allow solutions that increase the cost. These "uphill" solutions are accepted with a probability dependent on a parameter called the temperature. In any simulated annealing algorithm, four important criteria are the choice of the initial temperature, the equilibrium detection condition at a particular temperature, the rate of decrease of the temperature, and the frozen or termination condition. This set of criteria is referred to as the *annealing schedule.* Most implementations of simulated annealing use a fixed sequence of empirically-derived temperatures [63, 62].

Simulated annealing has been successfully applied to the standard cell placement problem in `TimberWolf` [62]. The cost function used in the original version of the `TimberWolf` algorithm (version 3.2) for standard cell placement consists of three parts:

- Estimate of the wire length of all nets as the half perimeter of the bounding box which is the smallest aligned rectangle containing all terminals in a net.
- Overshoot or undershoot of each row length over the desired row length
- Area overlap between cells in the same row

Two types of moves are used to generate new configurations. Either a cell is chosen randomly and displaced to a random location on the chip or two cells are selected randomly and exchanged. The ratios of displacement to exchange moves is set to about 5:1. A temperature dependent range limiter is used to limit the distance over which a cell can move. Initially, the span of the range limiter is set such that a cell can move anywhere on the chip. Subsequently, the span decreases logarithmically with temperature. The annealing schedule is fixed in that at each temperature a fixed number of moves per cell is attempted. The initial temperature is set to a very high fixed temperature, and the final temperature is a very low fixed temperature. The cooling schedule is computed as $T_{i+1} = \alpha(T) \cdot T_i$, where α varies between 0.8 to 0.95.

Other implementations of simulated annealing for cell placement have been reported. For example, in a later version of `TimberWolf` (version 4.2), several enhancements have been made to improve the run-time performance. The cost function is the same as the earlier version, but the move generation and cost computation have been changed. Each row is divided into non-overlapping bins, use of which makes the computation of the row overlap and row overshoot functions very efficient. For move generation, a cell is selected randomly and a random location is selected as a destination. If the destination is vacant, a displacement is performed, otherwise an exchange is performed. The temperature profile is, again, a set of fixed temperatures but over a smaller range. More recent versions of `TimberWolf` (version 6.0) use the notion of early rejection of bad moves. The basic idea being that before initiating the costly evaluation of the wire length portion of the cost function, the row and cell overlaps are determined. On the basis of those costs, if it is determined that the move is not going to be accepted, the wire length evaluation is bypassed.

7.2. Parallel Placement

Because of the inherent computational costs associated with simulated annealing, several researchers have investigated the parallelization of these placement algorithms. There have been two major approaches that researchers have taken to apply parallelism to simulated annealing for placement.

7.2.1. Parallel evaluation within each move

In this approach, each individual move is evaluated faster by breaking up the task of evaluating a move into subtasks and allocating various subtasks to different processors. This type of parallelism is also called *move decomposition* or *move acceleration.* The work of a move consists of selecting a feasible move, evaluating the cost changes, deciding to accept or reject, and perhaps updating a global database. One can then delegate individual portions of work such as wire length evaluation and updating to different processors [64]. Such approaches to parallelization are limited to shared memory algorithms and have limited scope for large scale parallelism.

7.2.2. Parallel evaluation of multiple moves

In this methodology, each processor evaluates a different move (cell displacement or cell exchange) in parallel. Unfortunately, there is a problem with using parallelism to propose and evaluate several moves simultaneously since global information about the state is required in order to evaluate the cost function. The cost function calculations may be incorrect due to interacting moves (see Figure 7.1.) If two processors are simultaneously considering moves involving different cells, one processor may move a cell after the second processor reads the state of the layout. The second processor now has inaccurate information about the positions of the cells and will use this out-of-date information to compute the cost of the proposed move. The resulting evaluation

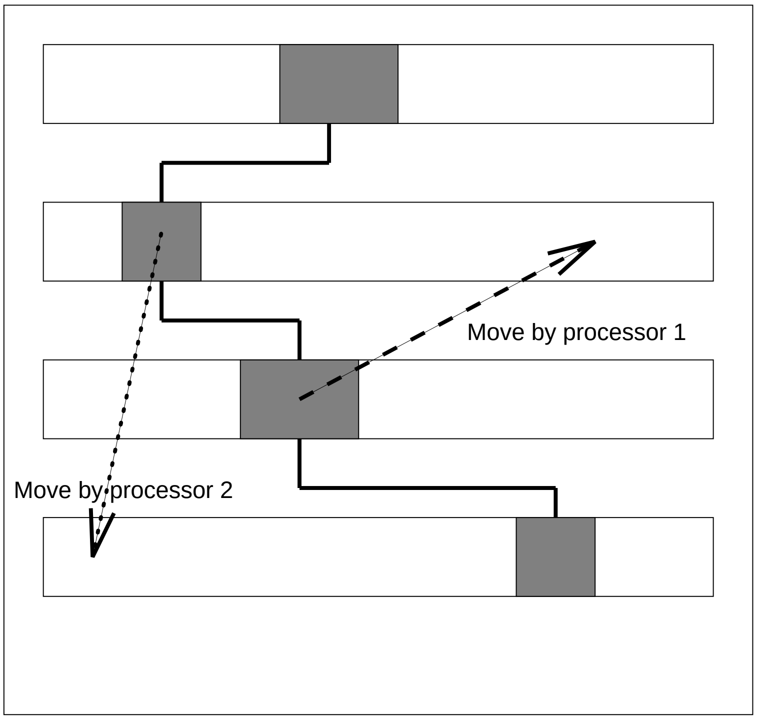

Figure 7.1 Illustration of parallel move interaction

will be wrong and may lead to a wrong decision in accepting or rejecting a move. There are two possibilities for handling these interactions in parallel moves.

Parallel evaluation of multiple moves and acceptance of moves that do not interact In this approach, a set of moves can be identified either statically or dynamically such that they do not interact and apply parallelism to those moves. The advantage of this method is that the convergence characteristics of the parallel algorithm are identical to those of the serial algorithm. However, since many possibly good moves are left unused, the majority of the computation is wasted, and hence, the effective parallelism is limited. In addition, the identification of moves that do not interact is itself an NP-complete problem [65].

Parallel evaluation and acceptance of multiple interacting moves The alternative is to allow multiple moves to be evaluated for acceptance based on inaccurate information; some errors in the computation of cost functions are permitted [66, 67]. Here, there can be problems with move interactions, affecting the convergence characteristics of the parallel algorithm. However, this approach allows for maximal parallelism. In such schemes, a large number of parallel moves are evaluated and accepted on the basis of past state information. The new cell positions are then updated after every set of parallel moves.

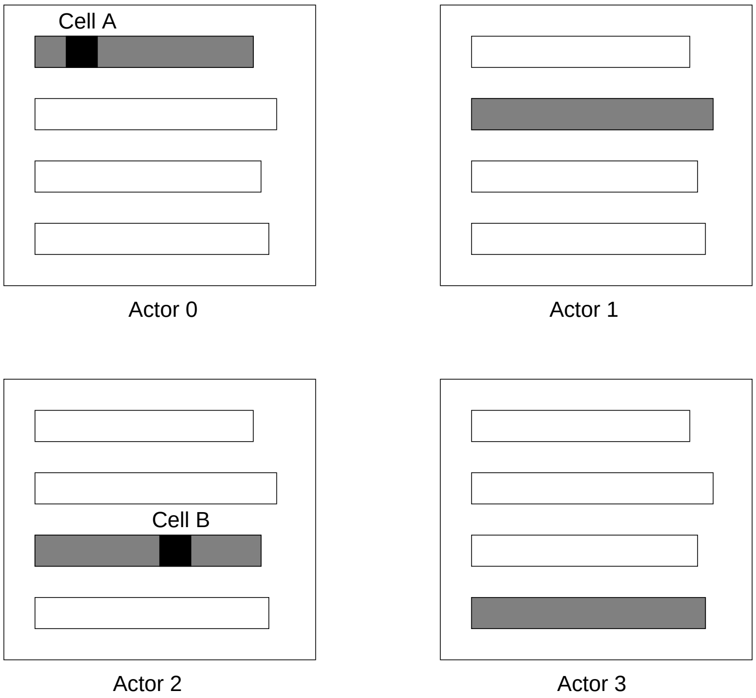

Figure 7.2 Row ownership in ProperPLACE. Example shows one row per partition.

7.3. ProperPLACE

ProperPLACE is based on an existing uniprocessor simulated annealing placement tool, TimberWolf[1]. This application is parallelized using the parallel interacting moves strategy described above.

In order to allow for these parallel moves, the circuit must be divided among the available threads. Ownership of specific cells is determined for each layout row by partitioning consecutive standard cell rows as shown in Figure 7.2. An Anneal actor is then created to process each partition or processor grouping, and concurrency is achieved as each actor independently proposes moves for evaluation. Typically, for maximum performance, one actor is created for each of the processors available in the target machine.

Each actor is responsible for moving cells that it owns, by completing the sequential simulated annealing schedule on its own partition. In ProperPLACE three different types of moves are possible (see also Figure 7.3):

M1 Intra-actor cell displacement
M2 Intra-actor cell exchange
M3 Inter-actor cell displacement

If a cell moves to a region owned by another actor (M2 or M3), the ownership of the cell goes to the new actor. To evaluate the cost function accurately, each processor must have complete cell location information in its local database. Therefore, after an actor accepts a move, it has

[1] The work described here is based on an earlier version of ProperPLACE [11] which was implemented using the Charm language, but has since been modified to take advantage of the ProperCAD II library.

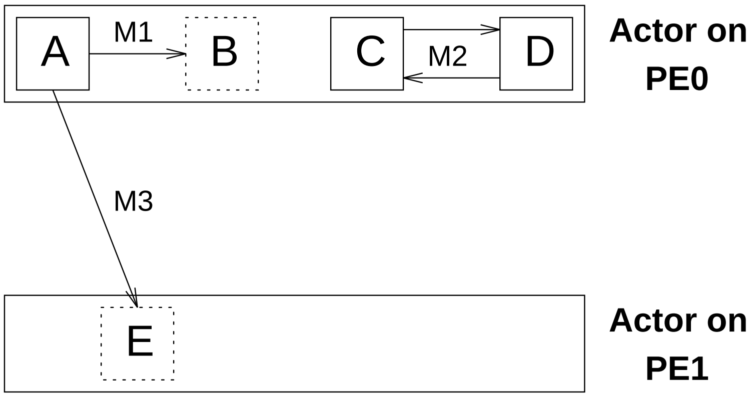

Figure 7.3 Moves in `ProperPLACE`

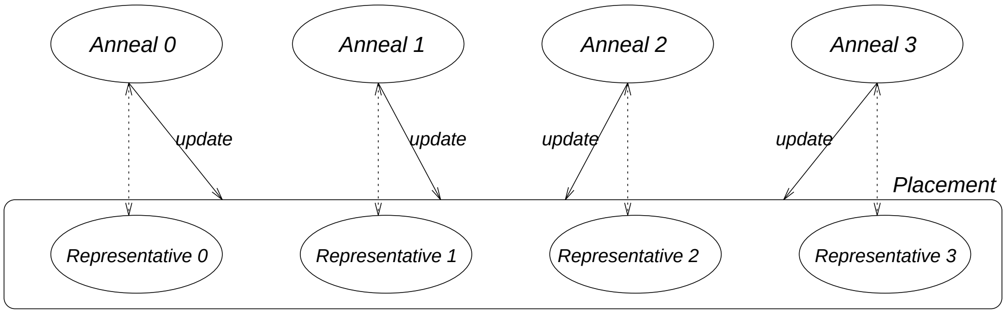

Figure 7.4 `Anneal` and `Placement` relationship in `ProperPLACE`

to propagate the accepted move to other actors so that each actor uses more or less the same information about cell locations.

This task of maintaining a coherent state of the current placement is the responsibility of the `Placement` aggregate which has a representative associated with each actor, as illustrated in Figure 7.4. Cell database management is the responsibility of the `Placement` aggregate. The relationships between the `Anneal` actors and the `Placement` aggregate are shown in the figure. The solid lines show the update messages that an actor will send to the aggregate every time it makes an accepted move. The dashed lines indicate a relationship between an `Anneal` actor and the corresponding `Placement` representative on its thread. It is through this latter association that an actor is able to access the cell location information. This ability to allow an actor access to internal representatives of an aggregate is a powerful feature of the ProperCAD II library.

The difficulty with updating after every accepted move is that the system will get flooded with update messages as more and more moves are accepted. To reduce this traffic a threshold, U, is introduced to limit the number of moves accepted before an update message is sent. This threshold may directly affect the quality of the solution. As U is raised, the cell position database becomes more and more inaccurate thus causing errors. However, previous researchers [65, 66] have shown that simulated annealing is tolerant of some error in the cost function calculations.

Another user defined threshold, L, is used to limit the number of moves that an `Anneal` actor may perform in succession without interruption. Since actors are not preemptive, this threshold

Circuit	Original W	TimberWolf		ProperPLACE Processors							
				1		2		3		4	
		T	W	T	W	T	W	T	W	T	W
s298	126984	192.7	31832	183.6	29909	113.2	31364	99.90	36357	87.25	49873
s420	274056	315.0	45744	269.4	44330	191.3	48206	153.5	61794	143.2	79596
primary1	1202241	1403	285000	1281	265318	934.5	272686	823.8	281475	796.1	285963
struct	2668124	4011	182930	3540	173730	2588	174548	2469	183693	1529	221819
primary2	7292946	4784	1687487	4191	1410450	2825	1461104	2638	1615180	3265	1614893

Table 7.1 Results on the Sun 4/670MP

Circuit	Original W	TimberWolf		ProperPLACE Processors							
				1		2		4		8	
		T	W	T	W	T	W	T	W	T	W
s298	126984	190.0	31832	205.1	29558	113.2	31978	96.90	42868	82.87	74612
s420	274056	288.0	45744	318.4	37253	203.9	41356	149.3	48271	124.9	94823
primary1	1202241	—	285000	1442	271062	937.9	286624	676.5	291960	568.9	434095
struct	2668124	—	182930	3795	173730	2554	180038	1938	177153	1928	318648
primary2	7292946	—	1687487	4347	1410450	2906	1449492	2200	1708271	1968	2009394

Table 7.2 Results on the INTEL iPSC/860

determines the granularity of the `Anneal` actor workload. After L moves have been evaluated, the actor gives up control of the thread, to allow the `Placement` aggregate to gain control to process update messages. To ensure that the actor regains control, it must send a message to itself, before giving up control.

7.4. Performance

The ProperCAD II implementation of `ProperPLACE` has been run on all platforms supported by the library. Results are shown in Tables 7.1, 7.2, and 7.3 for a sampling of circuits from a standard benchmark set. All times, reported in seconds, represent the elapsed wall clock time. The wire-length measure, W, is a cost factor generated by `TimberWolf` that estimates the total wire length of the layout.

The results show that `ProperPLACE` achieves reasonable speedup with moderate impact on resultant quality. As circuit size and number of processors grow, the speedups become smaller because the amount of inter-actor communication grows. Notice, also, on circuits such as `struct`, where the wire length has been reduced by over a factor of 10, the speedups are small. This behavior is caused by the poor initial layout quality, which results in a dramatic increase in the number of accepted moves. This causes an increase in the number of update messages, and thus a drop in speedup. Layout quality declines due to algorithmic issues that do not allow inter-actor cell exchanges. Algorithm modifications are a subject of future work.

Circuit	Original W	TimberWolf		ProperPLACE Processors							
				1		2		4		8	
		T	W	T	W	T	W	T	W	T	W
s298	126984	536.5	31832	561.2	29909	372.9	29668	286.8	46384	142.2	79585
s420	274056	776.7	45744	926.9	44330	586.4	46354	392.7	71068	169.3	87816
primary1	1202241	4887	285000	4571	265318	2983	268257	2182	295594	1927	439205
struct	2668124	12098	182930	12477	173730	9481	181214	6708	191249	6650	332345
primary2	7292946	15266	1687487	15427	1410450	10417	1435690	8295	1900559	6844	2892132

Table 7.3 Results on the Encore Multimax

8 Conclusions

We have presented a run-time library suitable for a strongly- and statically-typed language, C++, which implements the actor model of concurrent computation. We have shown how a library-based implementation can be used to incrementally modify an existing code with only incremental increase in development cost. We demonstrated, through a scientific kernel and two large industrial applications, the efficacy of this approach on contemporary architectures. On-going work comprises the addition of new applications, implementation on other parallel platforms including workstation clusters, and development of a compiler which generates code utilizing the services of the run-time library.

References

[1] Kendall Square Research, Waltham, MA, *KSR-1 Technical Summary*, 1992.

[2] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. L. Hennessy, M. Horowitz, and M. Lam, "The Stanford DASH," *IEEE Computer*, vol. 25, pp. 63–79, Mar. 1992.

[3] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Computers*, pp. 241–248, Sept. 1979.

[4] V. S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2, pp. 315–339, 1990.

[5] Parasoft Corporation, Pasadena, CA, *Express Reference Guide for FORTRAN Programmers*, 1992.

[6] A. W. Appel, *Compiling with Continuations*. Cambridge University Press, 1992.

[7] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: A mechanism for integrated communication and computation," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 256–266, May 1992.

[8] B. Ramkumar and P. Banerjee, "ProperCAD: A portable object-oriented parallel environment for VLSI CAD," Tech. Rep. CRHC-93-04/UILU-ENG-93-2205, Center for Reliable and High-performance Computing, University of Illinois, Jan. 1993. A shorter version of this report appears in the Proceedings of the International Conference on Computer Design, 1992.

[9] K. De, B. Ramkumar, and P. Banerjee, "ProperSYN: A portable parallel algorithm for logic synthesis," *Digest of Papers, International Conference on Computer-Aided Design*, pp. 412–416, Nov. 1992.

[10] B. Ramkumar and P. Banerjee, "ProperEXT: A portable parallel algorithm for VLSI circuit extraction," *Proceedings of the International Parallel Processing Symposium*, pp. 434–438, 1993.

[11] S. Kim, "Improved algorithms for cell placement and their parallel implementations." Ph.D. Dissertation, University of Illinois at Urbana-Champaign, July 1993. Tech. Rep. CRHC–93–18, UILU–ENG–93–2231.

[12] A. A. Chien, *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs*. The MIT Press, 1993.

[13] G. A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.

[14] C. Hewitt, "Viewing control structures as patterns of passing messages," *Journal of Artificial Intelligence*, vol. 8, pp. 323–364, June 1977.

[15] G. A. Agha, *Foundations of Object-Oriented Languages*, ch. The Structure and Semantics of Actor Languages, pp. 1–59. Springer-Verlag, 1991.

[16] H. Lieberman, *Object-Oriented Concurrent Programming*, ch. Concurrent Object-Oriented Programming in Act1, pp. 9–36. MIT Press, 1987.

[17] D. Theriault, "Issues in the design and implementation of Act2," Tech. Rep. 728, MIT Artificial Intelligence Laboratory, June 1983.

[18] E. Shibayama and A. Yonezawa, *Object-Oriented Concurrent Programming*, ch. Distributed Computing in ABCL/1, pp. 129–158. MIT Press, 1987.

[19] J. Ferber and P. Carle, "Actors and agents as reflective concurrent objects: A MERING IV perspective," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, pp. 1420–1436, Dec. 1991.

[20] C. Houck and G. Agha, "HAL: A high-level actor language and its distributed implementation," *Proceedings of the International Conference on Parallel Processing*, pp. 158–165, Aug. 1992.

[21] W. Fenton, B. Ramkumar, V. A. Saletore, A. B. Sinha, and L. V. Kalé, "Supporting machine independent programming on diverse parallel architecturs," *Proceedings of the International Conference on Parallel Processing*, Aug. 1991.

[22] A. A. Chien, V. Karamcheti, and J. Plevyak, "The Concert system — compiler and runtime support for efficient, fine-grained concurrent object-oriented programs," Tech. Rep. UIUCDCS-R-93-1815, Department of Computer Science, University of Illinois, June 1993.

[23] V. Karamcheti and A. Chien, "Concert — efficient runtime support for concurrent object-oriented programming languages on stock hardware," *Proceedings of Supercomputing '93*, pp. 33–36, 1993.

[24] G. Agha and C. Callsen, "ActorSpaces: A model for scalable heterogeneous computing," *Proceedings of the 26th Hawaii International Conference on System Sciences*, 1993.

[25] L. V. Kalé, "The Chare Kernel parallel progrmaming language and system," *Proceedings of the International Conference on Parallel Processing*, vol. II, Aug. 1990.

[26] A. Gursoy and L. V. Kalé, "Dagger: Combining the benefits of synchronous and asynchronous communication styles," Tech. Rep. 93-3, Department of Computer Science, University of Illinois, June 1993.

[27] L. V. Kalé, B. Ramkumar, V. Saletore, and A. Sinha, "Prioritization in parallel symbolic computing," *Library Notes of Computer Science*, 1993.

[28] D. Gannon and J. K. Lee, "Object-oriented parallelism: pC++ ideas and experiments," *Proceedings of the Japan Society for Parallel Processing*, pp. 13–23, 1993.

[29] K. M. Chandy and C. Kesselman, "Compositional C++: Compositional parallel programming," *Proceedings of the Fifth Workshop on Compilers and Languages for Parallel Computing*, pp. 79–93, 1992.

[30] W. J. Leddy and K. S. Smith, "The design of the experimental systems kernel," *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications*, pp. 10–17, Mar. 1989.

[31] B. Stroustrup, *The C++ Programming Language*. Addison Wesley, second ed., 1991.

[32] H. Baker and C. Hewitt, "The incremental garbage collection of objects," *Conference Record of the Conference on AI and Programming Languages*, pp. 55–59, Aug. 1977.

[33] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield, "The Amber system: Parallel programming on a network of multiprocessors," Tech. Rep. 89-04-01, Department of Computer Science and Engineering, University of Washington, Sept. 1989.

[34] B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Presto: A system for object-oriented parallel programming," *Software — Practice and Experience*, vol. 18, Aug. 1988.

[35] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, vol. 2, pp. 39–59, Feb. 1984.

[36] R. Chandra, A. Gupta, and J. L. Hennessy, "Integrating concurrency and data abstraction in a parallel programming language," Tech. Rep. CSL-TR-92-511, Computer Science Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Feb. 1992.

[37] N. Carriero and D. Gelernter, "Linda in context," *Communications of the ACM*, vol. 32, pp. 444–458, Apr. 1989.

[38] P. America, *Object-Oriented Concurrent Programming*, ch. POOL-T: A Parallel Object-Oriented Language, pp. 199–220. MIT Press, 1987.

[39] D. Kafura and K. H. Lee, "ACT++: Building a concurrent C++ with actors," *Journal of Object-Oriented Programming*, pp. 25–37, May/June 1990.

[40] C. M. Chase, A. L. Cheung, A. P. Reeves, and M. R. Smith, "Paragon: A parallel programming environment for scientific applications using communications structures," *Proceedings of the International Conference on Parallel Processing*, vol. II, pp. 211–218, 1991.

[41] R. Butler and E. Lusk, *User's Guide to the p4 Parallel Programming System*. Argonne National Laboratory, Argonne, IL, June 1992. Tech. Rep. ANL-92/17.

[42] B. Ramkumar, "Machine independent "AND" and "OR" parallel execution of logic programs." Ph.D. Dissertation, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Oct. 1990. Tech. Rep. UIUCDCS-R-90-1639.

[43] B. Ramkumar and P. Banerjee, "ProperTEST: A portable parallel test generator for sequential circuits," *Digest of Papers, International Conference on Computer-Aided Design*, pp. 220–223, Nov. 1992.

[44] T. Niermann and J. Patel, "HITEC: A test generation package for sequential circuits," *Proceedings of the European Design Automation Conference*, pp. 214–218, Feb. 1991.

[45] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Transactions on Computers*, vol. C-30, pp. 215–222, Mar. 1981.

[46] S. Chandra and J. H. Patel, "Test generation in a parallel processing environment," *Proc. Int. Conf. Comp. Design (ICCD-88)*, pp. 11–14, Oct. 1988.

[47] S. Patil and P. Banerjee, "Fault partitioning issues in an integrated parallel test generation fault simulation environment," *Proc. Int. Test Conf.*, pp. 718–727, Aug. 1989.

[48] S. Patil and P. Banerjee, "Performance trade-offs in a parallel test generation fault simulation environment," *IEEE Trans. Computer-Aided Design*, vol. 10, no. 12, pp. 1542–1558, Dec. 1991.

[49] S. Patil, P. Banerjee, and J. Patel, "Parallel test generation for sequential circuits on general purpose multiprocessors," *Proc. 28th Design Automation Conf. (DAC-91)*, Jun. 1991.

[50] P. Agrawal, V. D. Agrawal, and J. Villoldo, "Sequential circuit test generation on a distributed system," *Proceedings of the Design Automation Conference*, June 1993.

[51] G. J. Li and B. W. Wah, "Manip-2: A multicomputer architecture for evaluating logic programs," *Proc. Int. Conf. Parallel Processing*, pp. 123–130, Aug. 1985.

[52] B. W. Wah, G. J. Li, and C. F. Yu, "Multiprocessing of combinatorial search problems," *IEEE Computer*, vol. 18, no. 6, pp. 93–108, June 1985.

[53] V. N. Rao and V. Kumar, "Parallel depth first search, part I: Implementation," *International Journal of Parallel Programming*, vol. 16, no. 6, 1987.

[54] V. N. Rao and V. Kumar, "Parallel depth first search, part II: Analysis," *International Journal of Parallel Programming*, vol. 16, no. 6, 1987.

[55] A. Motohara, K. Nishimura, H. Fujiwara, and I. Shirakawa, "A Parallel Scheme for Test-Pattern Generation," in *Proceedings of the International Conference on Computer Aided Design*, pp. 156–159, Nov. 1986.

[56] S. Patil and P. Banerjee, "A parallel branch and bound approach to test generation," *Proc. 26th Design Automation Conf.*, pp. 339–345, Jun. 1989.

[57] S. Patil and P. Banerjee, "A parallel branch and bound approach to test generation," *IEEE Transactions on Computer Aided Design*, vol. 9, pp. 313–322, Mar. 1990.

[58] S. Arvindam, V. Kumar, V. N. Rao, and V. Singh, "Automatic test pattern generation on parallel processors," tech. rep., Computer Science Dept, Univ. of Minnesota, May 1990.

[59] S. T. Patel and J. H. Patel, "Effectiveness of heuristics measures for automatic test pattern generation," *Proc. 23rd Design Automation Conf.*, pp. 547–552, 1986.

[60] S. J. Chandra and J. H. Patel, "Experimental evaluation of testability measures for test generation," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 93–97, Jan. 1989.

[61] R. H. Bell, Jr., R. H. Klenke, J. H. Aylor, and R. D. Williams, "Results of a topologically partitioned parallel automatic test pattern generation system on a distributed-memory multiprocessor," *ASIC '92*, Sept. 1992.

[62] C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf placement and routing package," *IEEE Journal for Solid State Circuits*, vol. SC-20, pp. 510–522, April 1985.

[63] S. Kirkpatrick, C. Gelatt, and M. Vecci, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, May 1983.

[64] S. Kravitz and R. Rutenbar, "Multiprocessor-based placement by simulated annealing," in *Proceedings of the 23rd Design Automation Conference*, pp. 567–573, June 1986.

[65] P. Banerjee, M. H. Jones, and J. Sargent, "Parallel simulated annealing algorithms for standard cell placement on hypercube multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 91–106, Jan. 1990.

[66] R. Jayaraman and R. A. Rutenbar, "Floorplanning by annealing on a hypercube multiprocessor," in *Proceedings of the International Conference on Computer Aided Design*, pp. 346–349, Nov. 1987.

[67] J. S. Rose, W. M. Snelgrove, and Z. G. Vranesic, "Parallel cell placement algorithms with quality equivalent to simulated annealing," *IEEE Transactions on Computer-Aided Design*, vol. 7, pp. 387–396, Mar. 1988.