To appear: Supercomputing '94, November 14-18, 1994, Washington, D.C.

A Library-based Approach to Portable, Parallel, Object-Oriented Programming: Interface, Implementation, and Application

Steven Parkes
parkes@crhc.uiuc.edu

John A. Chandy
jchandy@crhc.uiuc.edu

Prithviraj Banerjee
banerjee@crhc.uiuc.edu

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

Abstract

The use of parallel platforms, despite increasing availability, remains largely restricted to well-structured, numeric applications. We address the issue of facilitating the use of parallel platforms on unstructured problems through object-oriented design techniques and the actor model of concurrent computation. We present a multi-level approach to expressing parallelism for unstructured applications: a high-level interface based on the actor model of concurrent object-oriented programming and a low-level interface which provides an object-oriented interface to system services across a wide range of parallel architectures. The high- and low-level interfaces are implemented as part of the ProperCAD II C++ class library which supports shared-memory, message-passing, and and hybrid architectures. We demonstrate our approach through a detailed examination of the parallelization process for an existing unstructured serial application, a state-of-the-art VLSI computer-aided design application. We compare and contrast the library-based actor approach to other methods for expressing parallelism in C++ on a number of applications and kernels.

1. Introduction

The goal of utilizing multiple processors to solve difficult problems has, to date, been largely unattainable for all but a set of restricted—though significant—problems, namely numeric and transaction-processing problems. While substantial problems exist in other fields, the techniques used in scientific and database applications have not proven similarly effective on unstructured problems. The lack of structure in these problems deemphasizes floating point vector operations and emphasizes pointer-based data structures and operations comprising a mixture of integer and floating point arithmetic. Of particular interest is the issue of composability: the ability to combine modules representing solutions to sub-problems without deleterious inter-module interaction.

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

In this work, we present a new interface and implementation for expressing parallelism, the most significant characteristics of which are

- Statically-typed high-level interface based on Actors
- Class library-based interface and open implementation
- Multiple levels of abstraction
- Incremental parallelization

Many of the choices made in the development of these characteristics involved consideration of the needs and preferences of the target user community, a study of which is presented by Pancake and Bergmark [1]. Although their work focused on scientific applications, with a few exceptions their observations and conclusions are equally applicable to other problem domains. We borrow from their discussion as we consider the characteristics of our work.

Actor parallelism: Concurrency in an actor program is implicit in the semantics of the model. Interprocess communication is expressed via *continuation execution,* an extension of the member function execution mechanism of serial object-oriented languages. The use of explicit continuations makes the interface statically type-safe.

Library-based interface: Pancake and Bergmark classify interfaces as *concurrent languages, language extensions,* and *runtime libraries*. The interfaces in our approach are manifest as C++ class libraries. A new concurrent language was rejected because it would require the sacrifice of a large investment in existing C and C++ code. An extension to C++ was considered but rejected in favor of the class library approach because it was felt this would maximally conserve our investment in tools, code, and user experience without seriously limiting expressibility.

Multi-level abstraction: For a model of concurrency to be portable, the infrastructure—the interface, compiler, and run-time support—must insulate the programmer from details of the underlying hardware. Yet such insulation is not without cost; it may constrain the expressibility of the interface and/or the efficiency of implementation. As a result, most languages and libraries do not implement a single model of parallelism but instead provide primitives at various levels of abstraction. Unfortunately, this can lead to an interface which is confusing and difficult to manage [1]. As a result, we have developed two models. The high-level model is capable of expressing actor operations which are easily implemented on any machine, though at costs which will vary from architecture to architecture. The high-level interface is implemented via a well-defined low-level model which captures architectural details such as address space distribution. Use of the high-level interface does not preclude direct use of the low-level interface.

Incremental parallelization: To make feasible the generation of parallel applications from existing serial applications, not only must the cost of the parallelization process be minimized, but the parallel application must be considered a variant of the serial application. In our approach, we have targeted incremental parallelism in two ways. First, the library approach implies that a serial program with no calls to the library is a degenerate library client; incrementally greater degrees of parallelism are expressed by adding additional library primitives. Second, we implement parallelism via derivation; in a real sense, the parallel application is *derived* from the serial application. By applying parallelism via derivation, it is possible to share both source and object code with only limited perturbation of the serial code.

The ProperCAD project [2], of which this work is a part, has as its goal a suite of tools for solving the most computationally intensive problems in VLSI CAD. The ProperCAD II library provides the infrastructure that enables these tools to run unchanged across a range of parallel platforms.

2. An unstructured problem: test pattern generation

To motivate the need for parallelization techniques for unstructured applications, we consider those characteristics which make an application unstructured and introduce *test generation,* an unstructured application drawn from the area of VLSI CAD. We define unstructured problems as those not naturally expressible as operations on arrays or through the iterative solution of a number of similar problems. The area of computer-aided design for VLSI circuits has many such applications.

2.1. Test generation: a search problem

Most VLSI CAD problems can be viewed as either search or optimization problems. We use as an example *test pattern generation,* a search problem. Test patterns are stimulus sets that when applied to fabricated devices detect the presence of manufacturing defects. The output patterns produced by a device in response to a set of test patterns are compared against expected results and the device is rejected if the patterns do not match.

We illustrate with an example. Test generation for a particular fault on a simple chip might reduce to finding a set of assignments to inputs a through h such that a Boolean function $t = a(b + c(def + g) + h)d + be$ is 1. The simplest way to find such an input is to search for one. We start by assigning an input, for example a, to a value, say 1. The result of doing this is a new expression, $b + c(def + g) + h)d$, so another input is chosen and the process repeated. If assigning an input results in $t = 1$, then that and the preceding assignments is a test. However, a result $t = 0$ indicates only that the particular set of assignments chosen is not a test; it does not indicate that there is no test. Each earlier assignment must be retried with the opposite value. This search process is performed once for each potential fault in a circuit.

Generating a set of tests that ensure that no device containing a defect is erroneously accepted is extremely difficult for contemporary chips containing millions of transistors. The problem of finding a set of inputs that yield a 1 is called *satisfiability*, an NP-complete problem. This class of problems has the characteristic that the number of possible solutions grows exponentially with the number of inputs. Because naive enumeration is impossible, *heuristics* are used to try to 'guess well.'

2.2. Parallel test

Given a set of heuristics, the process of parallelizing search is straightforward; multiple potential solutions are evaluated in parallel. Where no solution exists, near perfect speedup is observed. However, when a solution does exist, the effectiveness of the parallelization effort depends directly on how much extra search is done beyond that performed by a serial search. In particular, if the first candidate examined by the serial search is nearly always correct, little speedup will be observed.

Two approaches to parallel test generation are *fault-parallelism* and *decision-parallelism* (see Figure 1). In fault-parallelism, a number of the many faults in a design are targeted in parallel. While this generally generates sufficient parallelism, it does not reduce the time required to find a test for an individual fault and can lead to decreased result quality. As a result, decision-parallelism can be used, in which case the search tree for a particular fault is searched in parallel. In the figure, the test generator makes two types of decisions, both of which may be parallelized.

2.3. `HITEC`: a test generator for sequential circuits

`HITEC` [3] is a sequential test generation which exhibits performance and quality among the best known in the field. Table 1 gives a brief description of the major classes in `HITEC`. The original package comprises 25,000 lines of

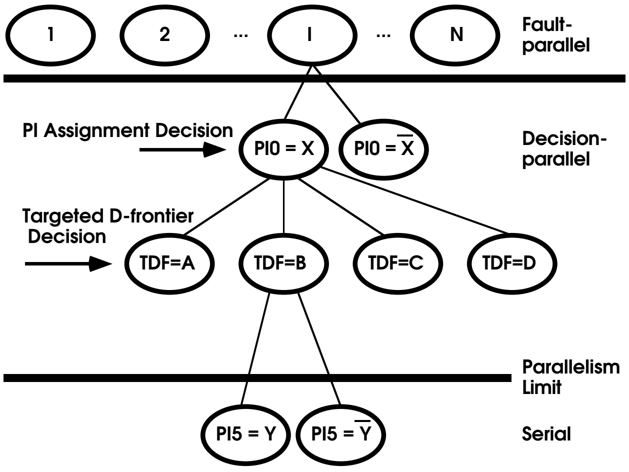

Figure 1 Parallelism in Test Generation

Table 1 Classes in `HITEC`

Class	Description
`Circuit`	circuit connectivity
`Window`	circuit state during search
`Fault`	fault status database
`VectorStates`	database of test vectors

C++.

To implement parallel search in `HITEC`, we create multiple search processes—in `HITEC`, `Window` objects—either from the fault to be tested (fault parallel) or from the fault, the current state of the heuristics, and the set of previously assigned inputs (decision parallel). We must also distribute the data represented by the other classes, e.g., the fault database. Although conceptually straightforward, application of parallel search techniques to test generation on a real algorithm holds considerable implementation complexity. We will see in the sequel how the actor interface is used to express parallelism in `HITEC`.

3. The actor interface

The Actor Interface (AIF) provides a high-level interface for expressing parallelism. The purpose of a high-level interface is to insulate the application—and by extension, the developer—from the characteristics of the underlying hardware. The AIF is based closely on the Actor model [4] and supports such extensions as aggregates and meta-programmability. In this section we consider the basic actor interface; in Section 4 we consider implementation and interface features which facilitate less abstract operations.

3.1. The actor model and continuation-passing style

The fundamental object in the Actor model [4] is the actor, an object which communicates with other actors via messages. All actions an actor takes are in response to messages. There is a close relationship between sending messages in the Actor model and calling remote procedures in the RPC model. Because messages automatically invoke a member function when received, the actor *send* operation more closely resembles a remote procedure call than it does a *send* operation in a send-receive programming model.

The actor model lacks explicit sequencing primitives. Synchronization is implicit and derives from the single-threaded nature of individual actors. The return executed at the completion of an actor method is an implicit wait; the actor automatically becoming available for any pending method invocations. Since an actor can not suspend execution implicitly in the middle of a computation, *continuation-passing style* (CPS) [5] is used to express control and data dependencies. Figure 2 contrasts the RPC and Actor/CPS models.

In the RPC model, a function `f` invokes a function `g` which executes on a remote thread. When the call to `g` occurs, the run-time system on the calling thread marshals any arguments, sends a message to the run-time system on the remote thread, and blocks waiting for a response. On the remote thread, the run-time system receives the message, unmarshals arguments, and invokes `g`. When `g` executes a `return`, the run-time on the remote thread marshals the return value and sends a message to the local thread which unmarshals the return value and continues `f`. RPC represents *distributed* programming, a single thread of control in multiple address spaces; the RPC model does not express parallelism.

The Actor/CPS model extends the RPC model to support object-oriented programming and parallelism. Functions are invoked with respect to an actor. The call of an actor method is non-blocking and consequently actor methods do not return values. In the figure, the `f` member of actor `a` calls the `g` member of `b` but does not wait. Instead, a *continuation* is passed as an argument to `g`. The continuation that `f` passes, `a.f'`, specifies that the object `a` and the method `f'` will process the value returned by `g`. In this case, `f'` will be that code in the RPC `f` which is dependent on the return value of `g`. `g` treats the continuation `c` as

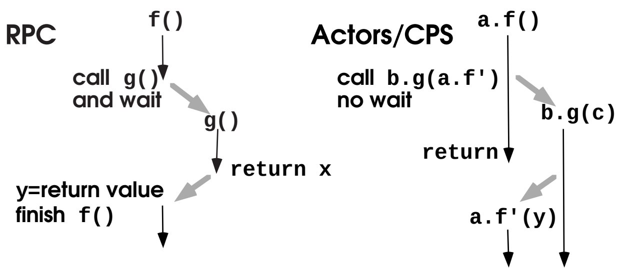

Figure 2 RPC and Actors/CPS

a function pointer; once the return value x is computed, g calls c with that value. Continuation variables are the parallel, object-oriented extrapolation of serial, procedural function pointers.

3.2. Basic interface

The library classes representing the fundamental abstractions in the actor model, are the `Actor` base class, `ActorNames`, `ActorMethods`, and `Continuations`.

`Actors`: Actor types are derived from the library-supplied class `Actor`. For example, to create a parallel test generator, we could modify `HITEC` to make the `Window` class an actor class:

```
class Window : public Actor { ... };
```

This, however, requires an incompatible modification of the serial application. Instead, we choose to define a new type which is both an actor type and a `Window`:

```
class TestGenerator :
  public Actor, public Window { ... };
```

Adding the `Actor` base to a class enables the creation of actor methods, actor names, and continuations, described below.

Member function calls and access to data members expressed via C++ pointers occur for actors as they do for any C++ object; parallelism and synchronization are expressed only through the use of continuations. This approach—the addition of mechanisms for expressing parallelism without changing the meaning of native C++ constructs—is taken throughout the interface. Rather than preclude constructs because they are *potentially* unsafe, we choose a design style where parallel constructs are added in those situations where semantics in a parallel environment are undefined. In this respect, we have tried to stay true to the 'spirit of C++' [6].

`ActorNames`: Actor names serve the role of pointers and references for instances of actor classes. Unlike C++ pointers, actor names are valid in the global namespace of a running program, independent of the number, type, and interconnection of threads executing the application.

Actor names, provided as a library-supplied template class, are *first-class* values, user-visible objects which possess a rich interface which facilitates parallel programming on contemporary architectures. For example, as one might expect, actor names may be created from a pointer to an actor:

```
TestGenerator* p = ...;
ActorName<TestGenerator> name = p;
```

`p` points to an actor which exists within the same address space as the object executing the code fragment; the pointer value is not necessarily valid on all threads. In contrast, `name` is valid on any thread. More interestingly, the inverse operation is also defined:

```
ActorName<TestGenerator> name = ...;
TestGenerator* p = name;
```

In this case, we start with an actor name and try to create a local pointer. After execution of the fragment, `p` will point to the actor represented by `name` if that actor is in the address space of the executing thread; otherwise it will be null. Other features of `ActorNames` are considered in the discussion of aggregates (Section 3.3) and the Gauss-Jacobi example (Section 7.1).

`ActorMethods` and `Continuations`: `ActorMethods` are member functions which may be invoked asynchronously and remotely. `ActorMethods` are executed via `Continuations`, the concurrent object-oriented extension of member function pointers. An actor method is a member function of an actor class that is, by definition, callable through a continuation:

```
class Vectors :
  public Actor, public VectorStates {
  ...
  void test( const VectorList& );
  class test
    : public ActorMethod<VectorList>
    { ActorMethodOf(VectorList); };
  ...
};
```

The `test` member function of the vector database class `Vectors`, derived from the `HITEC` `VectorStates` class, is called by `TestGenerator` actors when a test sequence is found. The declaration of the nested class `test` enables the creation of continuations that take a value of type `VectorList` as a parameter. The templated base class `ActorMethod` and the macro `ActorMethodOf` are provided by the library; together they define nested `Continuation` classes.

A continuation is created by specifying an actor method `Continuation` class and an actor name:

```
Vectors::test::Continuation
                    cont ( vectorDB );
```

This statement declares a continuation `cont` which when executed will call the member function `test` for the actor identified by the name `vectorDB`. Continuations are called using function-call syntax:

```
cont( list );
```

Here, the continuation `cont` is called with the vector list `list` as an argument. Execution via continuations is asynchronous with respect to the caller; actual execution of the member function occurs at some unspecified time in the future.

First-class continuations are necessary to write type-safe modular libraries. Code which calls a continuation is dependent only on the type of argument expected by the continuation; it is independent of the actor type and individual method which was used to create the continuation. In a statically-typed actor model lacking first-class continuations, client code is dependent on the type of the actor to which the result is to be returned. This precludes type-safe class libraries and/or separate compilation.

3.3. Collection interface

Individual actors express neither internal parallelism nor data distribution. Collection types, based on aggregates with explicit distributions, facilitate both object-internal parallelism and data distribution.

`Aggregates` and `AggregateNames`: An aggregate is a collection of actors which share a common name [7]. When a continuation created from an aggregate name is executed, the appropriate actor method is executed by one or more representative actors.

An example of the use of aggregates is a distributed array where non-overlapping ranges of elements are stored in individual actors. A distributed array can be represented as a set of actors or as an aggregate. In the actor implementation, a *gateway* actor redirects requests to the appropriate sub-range actor. In this case, an extra indirection is required and the indirection process is serialized. In the aggregate implementation, there is no need to send all requests to a single actor; all representatives of the aggregate share a common name and a request to that name may be sent directly to the appropriate representative. Serialization occurs only when two clients attempt to access simultaneously an element stored in the same representative.

Aggregate types are derived from the library class `Aggregate`. For example, the `HITEC` fault database class `Fault` becomes an aggregate in the parallel application:

```
class FaultDataBase :
  public Aggregate, public Fault
  { ... };
```

`Continuations` are created from `AggregateNames` just as they are for `ActorNames`; when such a continuation is called, a representative is selected by the run time system. The representative selection mechanism may be controlled by the application and supports broadcasting.

The first-class nature of names has a significant impact on the conceptualization and implementation of aggregates; `AggregateNames` may be coerced into `ActorNames` without loss of representative-selection flexibility. Such conversion enables the writing of libraries for which client code is independent of the choice of actor or aggregate implementation. Because the unicast/broadcast nature of a continuation is inherited from the name used to create the continuation, the unicast/broadcast choice can be made by the creator of the name. By contrast, in the original aggregate model the unicast/broadcast choice must be made at the point where the continuation is called [7]. The ability to encapsulate this information in continuation objects which are then passed to independent packages has considerable potential in the design of reusable library modules.

`Distributions`: Distribution of aggregate representatives is specified via an optional argument supplied when creating aggregate instances. Distributions specify the number and location of representatives of an aggregate. The interface of the abstract `Distribution` library class is general enough to express both enumerated and algorithmically-computed distributions. The library provides distribution classes which facilitate the most commonly used sharing abstractions found in medium-grain applications.

3.4. Advanced interface

The run-time support of the actor interface is metacircularly implemented via a number of library-supplied aggregates. Because the implementation is open, any of the system aggregate types may be used as a base class in an application, effectively customizing the run-time system in an application-specific manner. Although a detailed discussion of the advanced aspects is beyond the scope of this paper, we summarize the most significant features below.

`Director`: The main controller of each physical thread in an actor program is a representative of the `Director` aggregate. `Director` representatives are responsible for sending and receiving messages and for delivering messages to individual actors. The `Director` maintains all ready-to-run continuation invocations on a set of task queues, prioritizable via a application-extensible priority system.

`NameServer`: The `NameServer` aggregate is responsible for the allocation, distribution, and resolution of actor and aggregate names. `NameServer` representatives allocate names which are unique across all processors, route actor method calls to the appropriate processor for execution, and maintain binding information as new actors and aggregates are created.

The actor interface provides first-class message types; although messages are generally handled completely within the library, for reasons of efficiency, flexibility, and resource management, messages may be manipulated explicitly. The most common use of first-class messages is the minimization of the copying of data passed as parameters to continuations.

4. AIF implementation

The set of target architectures for the AIF comprises existing platforms supporting medium-grain parallelism. The wide variety of extant medium-grain parallel architectures makes the issue of portability vital; an implementation that runs on a limited number of platforms or is difficult to port would be of limited interest and have a short lifespan.

The implementation of the actor interface is defined in terms of a low-level interface, the Abstract Parallel Architecture (APA), which provides an interface and implementation which can be used to describe and utilize resources needed by any parallel application across a variety of architectures. The APA is used by the AIF but may also be used by applications directly. The AIF has been carefully integrated with the APA so that common types of architectural tuning and incremental parallelization can be expressed in a systematic way, reducing the need for ad hoc combinations of AIF and APA references.

4.1. Abstract parallel architecture

The Abstract Parallel Architecture (APA) comprises a model of a parallel computer and a set of objects that represent a reification of that model. The APA provides abstractions for thread, memory, and communication management and has been designed to provide a standard interface across all platforms with no extraneous overhead. The APA is self-sufficient and may be used apart from the actor interface.

A unique characteristic of the APA is integrated support for parallel architectures which combine message-passing and shared-memory features, i.e., where some but not all threads share some part of an address space. This eliminates the need for separate shared-memory and message-passing implementations. It also facilitates specializing communication for different configurations; when running on a workstation cluster which includes multiprocessors, communication occurs via shared-memory within a multiprocessor and via message-passing otherwise. The APA currently supports Sun and Encore multiprocessors and the Intel iPSC/2, iPSC/860, and Paragon multicomputers. The APA also supports IP-connected clusters of machines, where individual machines may be either uniprocessors or multiprocessors.

Thread management: The APA thread subsystem manages a set of processing elements, represented by instances of the `Thread` class. Thread objects are collected into sets characterized by the means by which they communicate. These sets are represented by the library classes `Process`, `ProcessGroup`, and `Cluster`. The relationships between the sets are depicted in Figure 3.

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

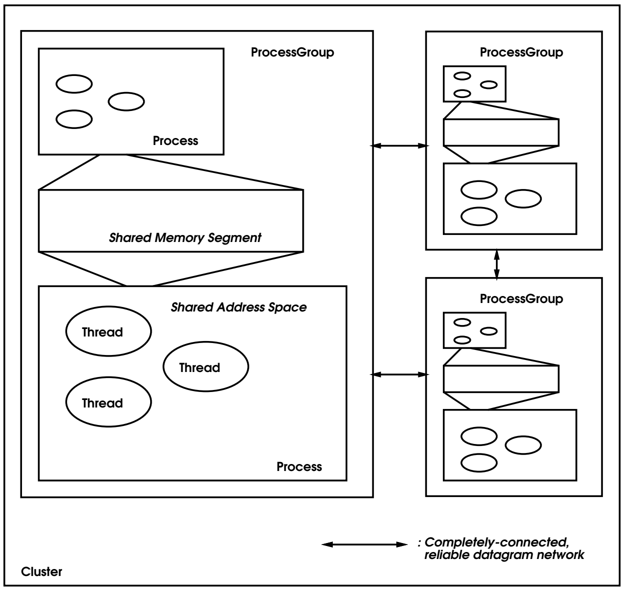

Figure 3 APA Thread Management Classes

`ProcessGroups:` `Threads` which share some portion of their address space are collected into a container called a `ProcessGroup`. Since `Threads` in a `Process-Group` may not have 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 systems using `mmap()` or System V IPC.

`Cluster:` All threads in a program are collected into a `Cluster` object. `ProcessGroups` are interconnected by a completely-connected, reliable, unordered, datagram network.

The expressibility of the APA is demonstrated by considering the mapping of contemporary architectures to the model. Table 2 shows the number of instances of each thread management class for a number of 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 is the product of the triple elements. Of particular interest are the last two rows in the table, an IP-connected

Table 2 APA triples for various machines

Machine	APA Configuration
Figure 3 architecture	3/2/3
16 node Intel iPSC	16/1/1
4-processor Sun 4/600MP	1/4/1
Two 4-processor Sun 4/600MPs connected by Ethernet	2/4/1
Intel 64 fat node Paragon	64/1/4

pair of Sun multiprocessors and an Intel Paragon with *fat nodes.* In contrast to most previous work, the APA represents machines which are hybrids of message-passing and shared-memory architectures.

Resource management: Free store (`malloc`) management is a common area of difficulty and little standardization. The APA provides a set of classes which implement free store management on top of the primitive operating system calls, e.g., `brk()` and `mmap()`. In addition to providing `malloc()` functionality for both private and shared memory segments, the free store classes provide multiple interface levels, i.e., very fast operations for memory private to a thread and slower operations (i.e., requiring critical sections) for memory which may be shared among multiple threads.

Communication management: The final portion of the APA provides the interface necessary to support interprocess communication. Abstractions are provided for `Semaphores`, `Networks`, and `Datagrams`. For network protocols where delivery is unreliable, e.g., UDP/IP, the APA implements reliability.

4.2. AIF/APA interaction

Many applications require or benefit from some amount of architecture-specific customization. While the complete APA interface is available to actor applications, it is desirable to provide high-level support for the most common architecture-specific optimizations.

The most common type of architecture-specific tuning is the distribution of data, represented by aggregate distributions. The AIF provides standard distributions based on the APA model: `PerThread`, `PerProcess`, and `PerProcessGroup`. Aggregates created with these distributions have one representative for each of the related APA instances. `PerThread` distributions can be used to represent data private to individual threads; member functions and data may be accessed by other actors allocated on the same thread without the possibility of a race. `PerProcessGroup` distributions can be used to represent read-only or, in concert with `Semaphores`, write-rarely data.

5. An unstructured parallel application: parallel test generation

To evaluate the expressiveness the AIF and efficiency of the implementation, we developed `ProperHITEC`, a parallel version of `HITEC`.

5.1. Implementation

Figure 4 shows the `ProperHITEC` classes which have a parallel semantics, along with the `HITEC` and AIF objects from which they are derived. We give a brief description of the parallel types below; further details may be found in [8].

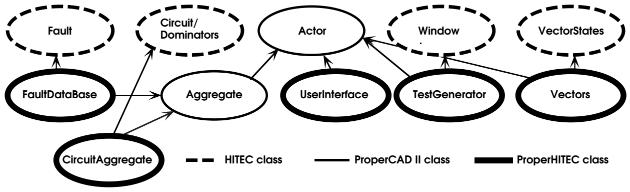

Figure 4 `ProperHITEC` Organization

`TestGenerator`: `TestGenerator` instances are created from a specified fault to implement fault-parallelism and cloned from an existing instance and an assignment alternative to implement decision-parallelism. Each `TestGenerator` constructor continuation is assigned a priority; when run on a uniprocessor, `TestGenerator` objects are created in the same order as the serial algorithm.

`CircuitAggregate`: The circuit is implemented as an aggregate with a `PerThread` distribution. Read-only sharing via `PerProcessGroup` distribution was implemented but removed because the method of parallel fault-injection resulted in excessive overhead in the core `HITEC` algorithm.

`FaultDataBase`: The fault database is implemented as an aggregate with a `PerThread` distribution. Each representative stores the most recent status of the test generation process for each fault.

`Vectors`: 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`.

`UserInterface`: The `UserInterface` object fills the role of the function `main` in a serial program.

Tables 3 and 4 show the results of `ProperHITEC` for a number of standard benchmarks. The column labeled `HITEC` gives the results of the serial application. T is elapsed wall clock time in seconds. Fault efficiency, E, is a measure of the quality of results; higher efficiencies, up to 100%, represent higher quality solutions. Results for the parallel application, `ProperHITEC`, is reported for several configurations.

`ProperHITEC` achieves consistent speedup with only a marginal effect on quality. The only cases for which

Table 3 `ProperHITEC` on Sun 4/670MP

Circuit	HITEC		ProperHITEC Processors					
			1		2		4	
	T	E	T	E	T	E	T	E
s344	369.4	95.9	374.3	95.9	251.9	96.2	156.2	96.2
s820	435.9	99.3	396.8	99.3	225.4	99.3	140.3	99.1
s1238	13.13	100	21.64	100	15.15	100	16.18	100
s1494	722.0	99.1	663.4	98.9	434.1	98.9	240.1	99.1

Table 4 `ProperHITEC` on the Intel iPSC/860

Circuit	`HITEC`		`ProperHITEC` Processors					
			1		4		8	
	T	E	T	E	T	E	T	E
s344	481.4	94.2	485.8	94.2	194.5	96.8	142.1	96.5
s820	438.3	99.3	440.8	99.3	158.0	99.3	108.0	98.9
s1238	14.15	100	23.29	100	12.16	100	11.12	100
s1494	819.8	99.0	821.3	98.7	310.1	98.6	192.2	98.8

the parallel algorithm does not achieve acceptable results are the benchmarks for which 100% efficiency is achieved within a few seconds. This is due to algorithmic issues and is not related to overhead within the run-time library.

5.2. Experiences and observations

The most interesting characteristic of `ProperHITEC` is the use of derivation to extend the serial application to a parallel version. The only fundamental change made to the serial code was the modification of several functions to use dynamic binding. Because dynamic dispatch involves an additional level of indirection, the impact on performance was studied but was found to have no measurable impact on performance.

6. Alternate approaches

A significant amount of related work has been performed in concurrent object-oriented computing; we concentrate on approaches related to actors and C++.

A number of pure actor languages based on the Actor model have been implemented [9]. The majority of these approaches have targeted fine-grain concurrency on massively-parallel MIMD architectures. Chien developed an actor language, Concurrent Aggregates (CA), which in addition to the features of actors and aggregates provides support for first-class continuations and messages [7, 10]. Charm and Charm++ [11, 12] were developed to address the need for a portable, parallel language realizable across a wide variety of shared-memory and message-passing architectures. pC++ [13] is an extension of C++ with support for distributed data structures, in much the same manner as High-Performance FORTRAN. CC++ [14] extends C++ with support for task parallelism. We consider how the test generation problem might be expressed on some of these systems.

6.1. Data parallelism

In data parallelism, a data structure, usually an array, is distributed across processors in a regular manner. This type of expression is natural in scientific computing where many algorithms are described as operations on arrays. This type of parallelism is supported by pC++, an extension of C++ [13].

It is difficult to hypothesize how one would implement test generation on top of a data parallel model. The majority of the work in test generation is incorporated in individual test generation tasks, not in a distributed data structure. Moreover, the distributed data structures in `ProperHITEC`, e.g., the fault database, are replicated on each thread; data parallelism usually does not apply to duplicated distribution.

The difficulty of expressing test generation in terms of data parallelism reinforces the need to choose the correct tool for the correct problem; in Section 7.1 we report an implementation of the Jacobi computation using the actor model. It is clear that while the actor interface can be used to efficiently implement the Jacobi computation, data parallelism and pC++ are the more intuitive ways of expressing the problem.

6.2. Task parallelism

Task parallelism is usually interpreted to mean either a series of expressions to be evaluated in parallel or a loop for which individual iterations are to be performed in parallel. This type of parallelism is supported by another extension of C++, CC++ [14].

To express test generation in terms of task parallelism the problem is broken up into similarly sized tasks. The approach taken in `ProperHITEC`, in which a large number of variable-weight tasks are created, poses two problems. The greatest difficulty is guiding the ordering of task execution; if ordering of execution of test generator objects is not preserved, processor efficiency and quality of results fall dramatically. Task parallelism generally does not provide a priority mechanism. Further, many task parallelism implementations work best where the degree of parallelism expressed is on the order of the number of threads in the machine. On such a system, a master-slave model would be needed to maintain an ordered list of test generation tasks. This light-weight scheduling is implicit in the actor interface.

Again, as is the case for data parallelism, there are problems for which task parallelism more closely matches the most intuitive solution. In particular, while features such as implicit barriers and futures can be expressed with continuations, such expression may be cumbersome. Applications which rely heavily on these operations could be tedious to translate to CPS.

6.3. Charm and Charm++

Charm and Charm++ are parallel programming environments, derived from C and C++ respectively, targeted at medium-grain architectures. The fundamental object of parallelism in Charm is the *chare,* an object with characteristics very similar to actors. Charm supports collection types via *branch-office chares* which are aggregate-like objects with implicit per-thread distribution.

The first phase of the ProperCAD project [2] used Charm as the parallel programming environment and many of the extensions to the actor model which exist in the AIF are drawn from or are extensions of features from Charm. The priority mechanism of Charm was determined in earlier work [2] to be crucial for efficient execution of CAD applications and was adopted and extended in the AIF. Features of the Actor Interface which differ most significantly from Charm are support for static typing, as represented by first-class continuations, and composability, as represented by per-class meta-programmability.

Because Charm does not have first-class continuations, two techniques are used to represent method calls. A user defined continuation pair, <EntryPointID,ChareID>, can be used. However, because the pair is not a system object, static type checking cannot be done to be ensure that the elements of the pair are consistent. Derivation has also been proposed to express a type-safe alternative [12]. A chare which will receive a message from another chare inherits from a receiver type defined by the sender. For example, the user interface in ProperHITEC which uses barriers to synchronize aggregate creation could be defined in Charm++ as

```
chare class UserInterface
: public BarrierReceiver { ...
     virtual void receiveBarrier();
... };
```

However, this precludes the use of more than one barrier function in a class. The UserInterface actor, which creates three barriers, would need to keep an extra state variable. By contrast, using first-class continuations a barrier actor is created with a continuation representing the method to be called in the computation.

Charm was one of the first actor models to implement advanced meta-programmability features and the Actor Interface incorporates extensions of many of the features in Charm. The most significant difference in the area of meta-programmability is the support for composability. In Charm, meta-programmability features, e.g. priority representations and queueing algorithms, are chosen at link time, implying that all classes which will be composed into an executable must use a uniform representation. In contrast, all meta-programability features in the Actor Interface are specified on a per-class basis such that the use of a particular representation by one class does not preclude use of other representations by other classes.

7. Experiences and observations

This section summarizes our experiences with other applications using the actor interface as well as some observations on the effectiveness and efficiency of the library in general.

7.1. Other applications

Gauss-Jacobi: An implementation of the Gauss-Jacobi iteration for partial differential equations was created to explore the performance of the library. Four implementations were created: serial, a serial implementation, shared-memory, a version for multiprocessors, message-passing, a version for multicomputers, and actor, an implementation based on the AIF. With the exception of actor, all implementations were implemented in C using the most efficient algorithm for the target architecture. Figure 5 shows the performance of 100 iterations of the Jacobi computation across a range of problem sizes on a multicomputer, the Intel iPSC/2, and a multiprocessor, the Encore Multimax. 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.

The most interesting aspect of the Jacobi computation is the use of first-class names. An actor type representing each sub-block of the computation, JacobiBlock, was created. Each actor of this type must communicate boundary values with neighboring actors after each iteration. To do this, each actor requires the name of those neighbors. Difficulties arise if names cannot be created separately from the creation of the individual actors. This linkage of name creation with actor creation in Charm results in the Jacobi

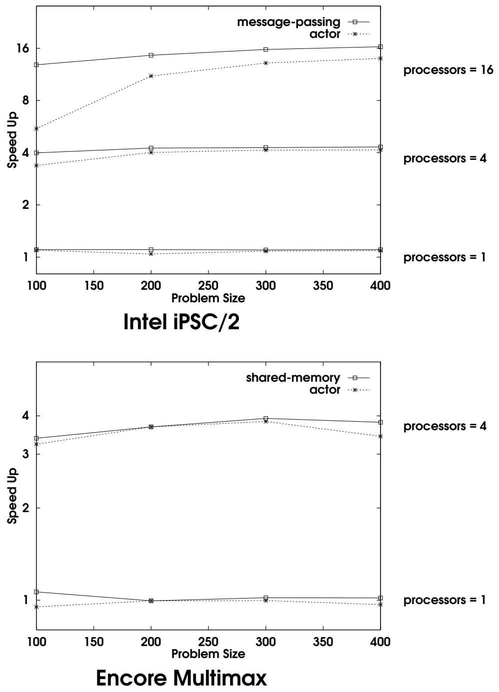

Figure 5 Gauss-Jacobi speedup over serial

iteration being expressed as a branch-office chare. Because names in the AIF may be created in a separate operation, there is no need to use aggregates to implement Jacobi.

Cell placement: When the logic design of a VLSI circuit has been completed, placement is performed to choose locations for each component such that the area of the layout is minimized. We have implemented `ProperPLACE`, a parallel version of `TimberWolf` 6.0, a simulated annealing placement application [15]. Results are shown in Table 5 for a sampling of circuits from a standard benchmark set. Times, reported in seconds, represent elapsed wall clock time. The wire-length measure, W, is a cost factor generated by `TimberWolf`. `ProperPLACE` uses actors, aggregates, first-class names, first-class messages, the ability to co-locate actors, and coercion of actor names to actor pointers.

Logic synthesis: Logic synthesis is the process of optimizing a digital circuit to reduce area or delay. We have developed `ProperMIS`, a parallel implementation of the MIS synthesis application [16]. The heart of the optimization procedure is rectangle covering, a standard search problem. Results of `ProperMIS` are shown in Table 6 for standard benchmarks on a Sun 4/670MP multiprocessor. Lit, the number of literals in the circuit, is a measure of cost. `ProperMIS` uses actors, aggregates, first-class names, first-class messages, the ability to co-locate actors, and coercion of actor names to actor pointers.

Table 5 `ProperPLACE` on Sun 4/670MP, Intel iPSC/860

Circuit	Original W	`ProperPLACE` Sun 4/670MP Processors			
		1		4	
		T	W	T	W
fract	111779	433.1	21184	265.1	22245
primary1	1202241	1048	249982	607.2	274238
struct	2668124	2701	173730	1164	184613
primary2	7292946	3201	1410450	1945	1678079
		Intel iPSC/860 Processors			
		1		8	
fract	111779	527.3	21184	179.6	22971
primary1	1202241	1442	249982	475.3	296624
struct	2668124	3795	173730	1164	225248
primary2	7292946	4347	1410450	1352	1727777

Table 6 `ProperMIS` on Sun 4/670MP

Circuit Name	Original Lit	number of processors			
		1		4	
		T	Lit	T	Lit
C499	1032	77.96	678	37.27	684
alu2	730	166.82	510	52.37	573
duke2	1746	83.36	497	40.55	506
misex3	2692	157.67	427	74.81	503

7.2. Strengths

First-class names and continuations: The ability to create, operate on, and interchange actor and aggregate names has turned out to be vital to the construction of an interface which has a high degree of expressibility but which supports static type safety and separate compilation. We believe that the expressibility of names linked with statically-typed continuations will be key to implementing application libraries that can be effectively composed to create new applications.

Derivation-based parallelization: Our experience with parallelization through derivation leads us to believe that this will be key to parallelizing existing codes without doubling development and support costs.

7.3. Weaknesses

Remote procedure calls: The actor model provides no blocking operations; although specification and implementation of blocking semantics may be difficult, experience shows that users strongly desire blocking calls.

I/O interface: The actor interface provides no direct support for I/O; we are investigating the use of additional threads, i.e., more than the number of processors available, which would be used to execute only blocking I/O operations.

Performance interface: The logical model of the APA does not capture available processing power; a measure of available resources must be developed and incorporated into the scheduling mechanism.

8. Summary

We have presented a run-time library suitable for a statically-typed language which implements the actor model of concurrency. We have demonstrated how a library-based implementation can be used to incrementally parallelize an existing serial code with only incremental increase in development cost. We demonstrated, through a large unstructured application, the efficacy of this approach on contemporary architectures. We contrasted our approach with others and identified some of the most significant strengths and weaknesses.

Acknowledgements: The authors gratefully acknowledge the support of the Argonne National Laboratory, the San Diego Supercomputer Center, and the National Center for Supercomputer Applications.

References

[1] C. M. Pancake and D. Bergmark, “Do parallel languages respond to the needs of scientific programmers?,” *IEEE Computer*, vol. 23, pp. 13–23, Dec. 1990.

[2] B. Ramkumar and P. Banerjee, "ProperCAD: A portable object-oriented parallel environment for VLSI CAD," *IEEE Transactions on Computer Aided Design*, vol. 13, pp. 829–842, July 1994.

[3] 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.

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

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

[6] B. Stroustrup, *The Design and Evolution of C++*. Addison Wesley, 1994.

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

[8] S. Parkes, P. Banerjee, and J. Patel, "ProperHITEC: a portable, parallel, object-oriented approach to sequential test generation," in *Proceedings of the 31st Design Automation Conference*, pp. 717–721, June 1994.

[9] G. Agha, "An overview of actor languagues," *SIGPLAN Notices*, vol. 21, pp. 58–67, Oct. 1986.

[10] 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.

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

[12] L. V. Kalé and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Proceedings of the 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 91–108, Sept. 1993.

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

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

[15] S. Kim, J. A. Chandy, S. Parkes, B. Ramkumar, and P. Banerjee, "ProperPLACE: A portable parallel algorithm for cell placement," in *Proceedings of the International Parallel Processing Symposium*, pp. 932–941, Apr. 1994.

[16] K. De, J. A. Chandy, S. Parkes, and P. Banerjee, "Portable parallel algorithms for factoring and simplification in logic synthesis." Submitted to the 1995 International Parallel Processing Symposium.