Distributed Object Oriented Data Structures and Algorithms for VLSI CAD

John A. Chandy[1], Steven Parkes[2], and Prithviraj Banerjee[1]

[1] Coordinated Science Laboratory, University of Illinois, Urbana, IL 61801, USA
[2] Sierra Vista Research, 236 N Santa Cruz Avenue, Los Gatos, CA 95030, USA

Abstract. ProperCAD II is a C++ object oriented library supporting actor based parallel program design. The library easily allows the design of data structures with parallel semantics for use in irregular applications. Inheritance mechanisms allow creation of the distributed data structures from standard C++ objects. This paper discusses the use of such distributed data structures in the context of a particular VLSI CAD application, standard cell placement. The library and associated runtime system currently run on a wide range of platforms.

1 Introduction

The use of parallel platforms, despite increasing availability, remains largely restricted to well-structured numeric codes. Irregular applications in terms of data access patterns as well as control flow are difficult to effectively and efficiently parallelize. The use of object-oriented design techniques and the actor model of computation can address the use of parallel platforms for unstructured problems. ProperCAD II is an object oriented library supporting the design of actor based parallel programs [1, 2]. The library easily allows the design of data structures with parallel semantics for use in irregular applications. Because the foundation is based on C++, inheritance mechanisms allow creation of the distributed data structures from standard C++ objects.

The domain of VLSI CAD provides a rich class of irregular problems. The computational intensity of VLSI CAD tools makes parallel processing an attractive solution [3]. However, most applications in this area are characterized by complex inter-related data structures as well as irregular access patterns across these objects. These properties make VLSI CAD applications particularly difficult to efficiently parallelize. The use of the ProperCAD II library as well as C++ design techniques help alleviate this problem. The approach has been used on several VLSI CAD problems including test generation [4], fault simulation [5], and VHDL simulation [6]. In this paper, we demonstrate the use of these techniques in a specific VLSI CAD problem, standard cell placement.

2 ProperCAD II

The major goal of the ProperCAD project [7] is to develop portable parallel algorithms for VLSI CAD applications on a range of parallel machines including shared memory multiprocessors such as the Sun SparcServer 1000E and the SGI Challenge, distributed memory multicomputers such as the Intel Paragon, IBM SP-2, and Thinking Machines CM-5, and networks of workstations. ProperCAD II provides C++ library-based machine independent runtime support in an object–oriented manner [1, 2] (Figure 1).

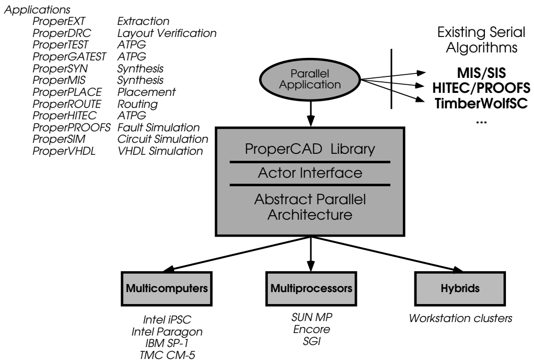
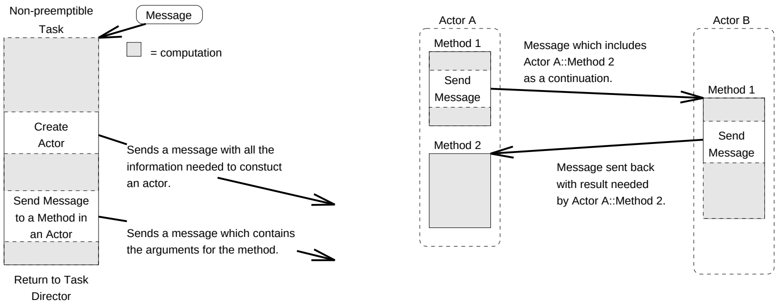


Fig. 1. An overview of the ProperCAD project.

2.1 Actor basics

The ProperCAD II library expresses parallelism through a statically-typed high level C++ interface based on actors. The interface is class library-based and allows multiple levels of abstraction as well as incremental parallelization. Through the use of a fundamental object called an actor [8], the library provides mechanisms necessary for achieving concurrency. An actor object consists of a thread of control that communicates with other actors by sending messages, and all actor actions are in response to these messages. Specific actor methods are invoked to process each type of message.



(a) Actor Behaviors

(b) Continuation Passing Style of execution.

Fig. 2. Actor model with continuation passing.

Figure 2(a) shows the three basic actions that an actor method actor can take: create new actors, send messages to actors, and perform computations that change its state. When a method creates an actor, a message is sent to the run time system with all the information needed to construct an actor. When a method sends a message, a message containing the arguments and the identity of the method to be invoked is sent to the run-time system for later execution. Both actor creation and message sends are non-blocking calls. The model only specifies that the actor be created or the task be run sometime in the future. Once a task starts, it runs to completion and cannot be preempted.

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 cannot suspend execution implicitly in the middle of a computation, *continuation-passing style* (CPS) [9] is used to express control and data dependencies. Figure 2(b) shows an example of continuation passing style. The actor model is a message-driven model in which the method name is in the message and the method is the code invoked upon message reception.

2.2 ProperCAD II interface

Applications created with ProperCAD II use five basic classes provided by the library: `Actor`, `ActorName`, `ActorMethod`, `Continuation`, and `Aggregate`.

All actor types are derived from the library-supplied class, `Actor`. Adding the `Actor` base to a class in a sequential object-oriented program enables the creation of actor methods and continuations as described below. These features allow the expression of parallelism. For example, a user class may be created as follows.

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

`ActorNames` serve the role of pointers and references for instances of actor classes. Because normal pointers are not valid across processor boundaries, actor names provide the mechanism for access of actors in a global namespace.

```
Foo* actorPtr = ...;
ActorName<Foo> fooName = actorPtr;
```

`ActorMethods` are member functions which may be invoked asynchronously and remotely. They are executed via `Continuations`, the concurrent equivalent of member function pointers. The definition and use of these constructs is shown below.

```
class Foo : public Actor {
  void bar( barArgs& );
  class bar : public ActorMethod<barArgs> {};
}

barArgs &args;
Foo::bar::Continuation cont ( fooName );
cont( args );
```

The `Foo` actor has a method `bar` which takes `barArgs` as an argument. This is easily designated as an actor method by creating a new nested class `bar` derived from a templated `ActorMethod`. In order to invoke the `bar` method asynchronously, we simply create a `Continuation` `cont` bound to a particular `ActorName`. We can now treat `cont` as member function pointer and execute it directly thereby causing an implicit message send. Since `Continuations` are first-class, `cont` may also be passed to another method.

Individual actors express neither internal parallelism nor data distribution. Collection types, based on aggregates, allow both concurrency within the object as well as data distribution. An aggregate is simply a collection of actors which share a common name [10]. An example of an aggregate would be a distributed array where different elements are stored on different actors. The use of aggregate representations removes the serialization step that would be required because of a gateway actor. The `Aggregate` interface is similar to that of the `Actor` class, and the creation and use of names and actor methods is accomplished similarly.

```
class FooAggr : public Aggregate { ... };
```

Aggregates provide the necessary mechanisms for distributed data structures. Because of the standard C++ interface, access to these distributed data structures is efficient. The benefit of aggregates is apparent in the following section in the presentation of an example application built using the ProperCAD II library.

3 Example: Standard Cell Placement

The VLSI cell placement problem involves placing a set of cells on a VLSI layout, given a netlist which provides the connectivity between each cell and a library containing layout information for each type of cell. This layout information includes the width and height of the cell and the relative position of each pin. The primary goal of cell placement is to determine the best locations of each cell so as to minimize the total area of the layout and the length of the nets connecting the cells together. Standard cell layouts are organized into equal height rows, and the desired placement should have equal length rows, as shown in Figure 3.

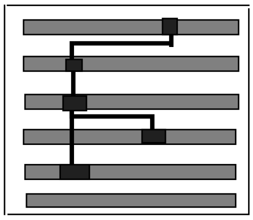


Fig. 3. Standard cell placement.

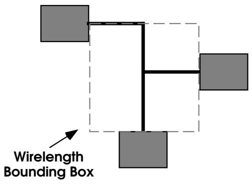


Fig. 4. Wirelength cost function.

3.1 Serial Algorithm

We have developed the parallel algorithm based on the serial `TimberWolfSC` program. `TimberWolfSC`'s core algorithm, simulated annealing, is a suitable approach to problems like VLSI cell placement since they lack good heuristic algorithms. Briefly, simulated annealing is an iterative improvement strategy that starts with a system in a disordered state, and through perturbations of the state, brings the system gradually to a low energy, and thus optimal, state. One of the unique features of simulated annealing is that, unlike greedy algorithms, perturbations that increase the energy of the system are sometimes accepted with a probability related to the temperature of the system [11]. In the context of cell placement, and `TimberWolfSC` in particular, perturbations are simply moves of the cells to different locations on the layout, and the energy is an approximated layout cost function. The major component of the cost function in `TimberWolfSC` is the sum of all nets bounding box perimeters as an estimate of net wirelength (Figure 4).

In order to understand the use of aggregates in parallelization, some further explanation of the data structures used is necessary. The circuit information is described primarily with the use of three arrays: the list of cells, the list of nets, and finally an array describing row information. Each cell data structure contains positional information as well as a linked list of pins that belong to the cell. Likewise, each net data structure has bounding box information as well as a linked list of pins that belong to the net. The pin data structures are shared by both the cell and net linked lists.

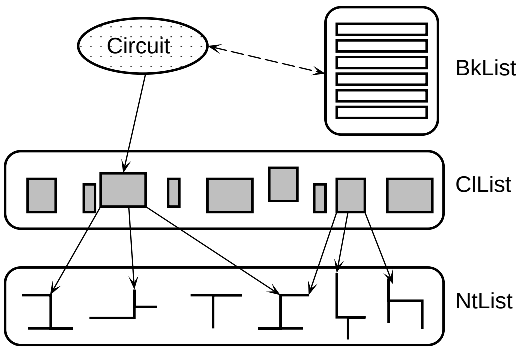


Fig. 5. Relationships between objects.

In an object-oriented framework, these data structures are C++ objects. The cell and net structures are thus placed in `Cell` and `Net` C++ classes respectively. Of particular interest are the cell and net arrays, which now become the `ClList` and `NtList` C++ objects. A `BkList` object is used to keep track of row information as well as the bins used for overlap penalty calculation. All the objects are integrated together in an object called `Circuit` which manages the core annealing algorithm. In addition, the `Circuit` object also contains many of the global parameters and flags that are relevant during annealing. The relationships between these objects is shown graphically in Figure 5.

The core algorithm is shown in the `Circuit::anneal()` code fragment in Figure 6. The `Circuit` object makes requests to the `ClList` object through the `pickACell()` method to pick a `Cell` to move. A new position for the cell is chosen by making a request to the `BkList` data structure. The `Circuit` object then asks the `ClList` to evaluate the delta cost of the move of the chosen `Cell`. If the move is accepted, then the data structures are updated through `updateCell()` which updates each net attached to `Cell`, through the `updatePin()` method.

```
class Circuit {
  ClList &carray;
  NtList &netarray;
  BkList &barray;
  // ...
};

Circuit::anneal()
{
  while ( terminationNotReached() )
    tryCellMove();
  finishUp();
}

Circuit::tryCellMove()
{
  Cell &cell = carray.pickACell();
  newLocation = barray.newLocation();
  deltaCost = carray.evaluateMove( cell, newLocation );
  if ( acceptMove( deltaCost ) )
    carray.updateCell( cell, newLocation );
}

ClList::updateCell( Cell &cell, Position &newLocation )
{
  for ( pinIterator pin(cell); pin.end(); pin++ ) {
    pin->updateLocation( newLocation );
    netarray.updatePin( *pin );
  }
}
```

Fig. 6. Core code for serial algorithm

3.2 Parallel algorithm

Parallelism through inheritance. In order to remain compatible with the arrays in the sequential code, we need arrays with distributed semantics. This is accomplished easily with the use of aggregates. For example, we can simply create a new distributed class, `CellAggr`, that is derived from the `ClList` class as well as the aggregate class, as below.

```
class CellAggr : public ClList, public Aggregate {
  Cell &pickACell();
  // ...
};
```

By doing so, the `CellAggr` class has representative actors on each processor that are responsible for the cells allocated to that particular processor. Thus, a request made to access a particular cell can be made to the local representative actor which will then forward it on to the actor on the thread where the cell is actually present. A similar transformation is done for the net array. In total, there are aggregates for the cells (`CellAggr`), nets (`NetAggr`), and rows (`BlockAggr`), as well as the `CktAggr` that is derived from `Circuit`. The dashed line in Figure 7 indicates the separation between two threads. Notice that the individual cells and nets are not replicated.

How does this data distribution affect the code? The main operations performed by the `Circuit::anneal()` method are the selection of a cell, evaluating the move, deciding whether to accept, and updating the cell. The evaluation and decision phases are independent of location of the cell. Picking a cell should be made only from the local pool of cells. To accomplish this, we make `ClList::pickACell()` a virtual base function, and rewrite the implementation as shown in Figure 8.

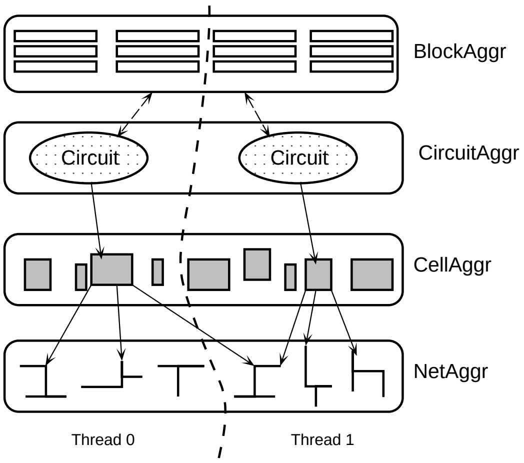


Fig. 7. Relationships between distributed objects.

Likewise, although updating cells is a local operation, the updating of nets can not be performed locally since the adjacent nets are distributed. Therefore, `updatePin()` is made virtual and rewritten as in Figure 8. We use the continuation invocation mechanism to invoke a remote `NetAggr::updatePin()`. The invocation of the remote method is asynchronous, so the update is in effect a "lazy" update.

The presence of the updates causes an interesting problem. Since actor methods are run to completion without preemption, if `Circuit::anneal()` is left as it is, the while loop will run to completion before any update messages can be serviced by the run time system. In order to allow these messages to be received, the while loop must be transformed into message driven code. As seen in Figure 9, through the use of virtual functions again, the `anneal()` method is modified to try only only one move, and then re-enable itself by executing an asynchronous continuation to itself. Similar transformations have been discussed in [12].

```
Cell &CellAggr::pickACell()
{
  Cell cell = ClList::pickACell();
  while ( !isLocalCell( cell ) )
    cell = ClList::pickACell();
}

class NetAggr : public NtList, public Aggregate {
  void updatePin( Pin& );
  class updatePin : public ActorMethod<Pin> {};
};

void NetAggr::updatePin( Pin &p, Position &location )
{
  if ( isLocalNet( p ) {
    NtList::updatePin( p, location );
  } else {
    updatePin::Continuation cont ( ownerOf( p ) );
    cont( p, location );
  }
}
```

Fig. 8. Code for `NetAggr` and `CellAggr`

```
class CktAggr : public Circuit, public Aggregate {
  void anneal();
  class anneal : public ActorMethodVoid {};
};

void CktAggr::anneal()
{
  tryCellMove();
  if ( terminationNotReached() ) {
    CktAggr::anneal::Continuation cont( *this );
    cont();
  } else {
    finishUp();
  }
}
```

Fig. 9. Code for `CktAggr`

Data distribution. The circuit is read in on a single thread, and as each cell and net is read in, the associated data structures are distributed to the other threads. The process of determining which cells are assigned to which processor is done using a prepartitioning phase. There are two primary concerns in our partitioning. First, the load balance must be maintained, i.e. each partition should have roughly the same number of cells. Secondly, the number of nets cut should be minimized to decrease the interaction between partitions. Ratio cut partitioning methods have long been used in the CAD community because of their effectiveness at reducing the cut size. However, these methods are inappropriate for our use because they do not provide well balanced partitions. We have instead used a partitioning algorithm based on the Sanchis modification of the Fiduccia-Mattheyses algorithm [13].

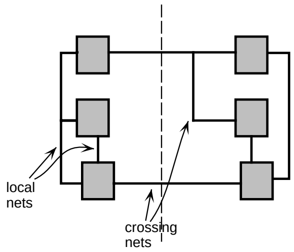


Fig. 10. Crossing nets.

When a request is made to evaluate the cost of a cell move, the connected `Net` objects may be located on another thread. One option would be for the `NetAggr` to send a message to the appropriate representative to request a calculation of the cost. The overhead involved in the communication makes this prohibitive. Therefore, copies of these nets that span multiple threads are replicated locally. An example of these "crossing" nets is shown in Figure 10. Consistency is maintained through the `updatePin()` method.

Dynamic redistribution. The final element of our parallel algorithm is the dynamic redistribution. As the annealing schedule proceeds, the initial partition becomes more and more irrelevant since it corresponds very little to the geographic partitioning of the rows. While the initial partitioning does reduce the amount of communication in terms of pin updates, the fact that the partition is spread across many rows affects the row penalty calculations as well as cell mobility. Therefore, it is reasonable to repartition the cells so that the partition reflects the geographical row-based partition. The repartitioning is started only after the cells have settled within some proximity of their final locations.

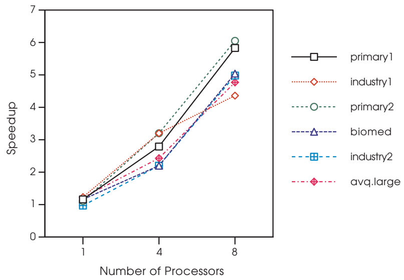
In the sample code in Figure 11, `CktAggr::anneal()` has been modified to execute a continuation to start up the repartition process. The details of the repartitioning are not shown, but is achieved through two phases of message sends which synchronize all the representatives of the aggregates. Notice that the `anneal()` re-enabling continuation is not executed. After the repartitioning has been completed, the `anneal()` method can then be re-enabled, and the asynchronous behavior starts again.

Experimental Results. We ran the parallel implementation on a shared memory multiprocessor Sun SparcServer 1000E as well as the Intel Paragon, a distributed memory machine. The speedups and wirelengths for a set of benchmark circuits are shown in Figures 12 and 13. The speedup graphs show an average speedup of 5 on 8 processors on the SparcServer and an average speedup of over 11 on 32 processors on the Paragon. The tables indicate the wirelength cost of the resultant placement relative to the cost from sequential `TimberWolfSC`. There is moderate (7-10%) degradation of the quality of the placement. The dashed lines on the Paragon results indicate that the circuit could not be run because of excessive memory requirements. These larger circuits (biomed, industry2, avq.large) show the advantage of using a distributed circuit approach to exploit the memory resources on multiple processors. For these larger circuits, the speedups are de-

```
void CktAggr::anneal()
{
  tryCellMove();
  if ( timeToRepartition() {
    CellAggr::repartition::Continuation cont( *this );
    cont();
  } else if ( terminationNotReached() ) {
    CktAggr::anneal::Continuation cont( *this );
    cont();
  } else {
    finishUp();
  }
}

void CellAggr::repartition()
{
  // perform repartitioning steps
  // ...
  CktAggr::anneal::Continuation cont( cktName );
  cont();
}
```

Fig. 11. Code for repartitioning



(a) Speedups

Circuit	1 proc	4 procs	8 procs
primary1	1.02	1.01	1.03
industry1	1.01	1.06	1.08
primary2	1.01	1.02	1.03
biomed	0.92	1.08	1.10
industry2	0.95	1.13	1.11
avq.large	1.00	1.05	1.09
Average	0.98	1.06	1.07

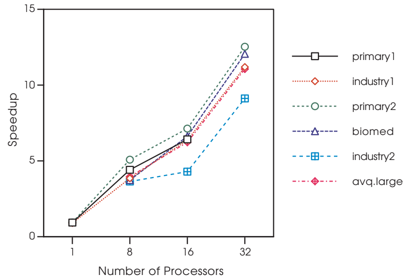
(b) Wirelengths

Fig. 12. Results on a Sun SparcServer 1000E (Shared Memory Multiprocessor)

rived from uniprocessor times extrapolated from a SUN4/690MP, a machine with comparable uniprocessor performance.

4 Related work

Several other researchers have produced work in environments to support irregular applications in object oriented environments, such as Charm++ [14], pC++ [15], CC++ [16]. Charm++ provides similar run time support for message driven applications to ProperCAD II. The primary differences are Charm++'s lack of support for static message typing, as represented by first-class continuations, and composability. pC++ is a language extension of C++ with support for data parallel semantics in much the same manner as HPF [17]. Since it presents a data parallel view of the world, it is difficult to express irregular problems such as VLSI CAD in this framework. CC++ achieves concurrency



(a) Speedups

Circuit	1 proc	8 procs	16 procs	32 procs
primary1	1.01	1.00	1.08	[a]
industry1	0.95	1.03	1.13	1.12
primary2	1.00	1.04	1.13	1.09
biomed	-	1.03	1.06	1.07
industry2	-	1.06	1.08	1.13
avq.large	-	1.05	1.05	1.08
Average	0.99	1.04	1.09	1.10

[a] Circuit was not large enough to be effectively partitioned over 32 processors

(b) Wirelengths

Fig. 13. Results on an Intel Paragon (Distributed Memory Multicomputer)

through parallel constructs which direct particular code fragments to be performed on different processing threads. This task parallelism approach can mimic many of the features in actor model. However, the ProperCAD II library does provide extra meta programmability features that allow the program designer to change the behavior of the run time system such as queuing policies, memory usage, load balancing, etc.

Other work to support irregular applications but not targeted toward object-oriented environments include Multipol [18] and PARTI/CHAOS [19]. Multipol provides a library of distributed data structures for use with a message driven run time system, based on atomic threads, the functional equivalent of actor methods. PARTI/CHAOS offers irregular run-time support for iterative irregular computation where the communication pattern is unchanged and predictable, but not resolvable at compile time. Neither Multipol or PARTI/CHAOS allow parallelism via derivation as available in ProperCAD II.

5 Conclusions

In this paper, we have presented a methodology for creating parallel programs and distributed data structures through simple inheritance mechanisms present in C++ and the use of the ProperCAD II library. The approach has been used on a wide variety of applications. The distributed data structures presented in this paper for standard cell placement are effectively being reused through object-oriented methodologies for global routing and timing driven placement.

Acknowledgements

Our sincere thanks to Dr. Carl Sechen for providing us with the `TimberWolfSC` 6.0 placement code. We are also grateful to the San Diego Supercomputing Center for providing us access to their Intel Paragon, and also to Intel Corporation for the donation of an Intel Paragon. This research was supported in part by the Semiconductor Research Corporation under contract 95-DP-109 and the Advanced Research Projects Agency under contract DAA-H04-94-G-0273 administered by the Army Research Office.

References

1. S. Parkes, J. A. Chandy, and P. Banerjee, "A library-based approach to portable, parallel, object-oriented programming: Interface, implementation, and application," in *Proceedings of Supercomputing '94*, (Washington, DC), pp. 69–78, Nov. 1994.
2. S. M. Parkes, "A class library approach to concurrent object-oriented programming with applications to VLSI CAD." Ph.D. Dissertation, University of Illinois at Urbana-Champaign, Sept. 1994. Tech. Rep. CRHC-94-20/UILU-ENG-94-2235.
3. P. Banerjee, *Parallel Algorithms for VLSI Computer Aided Design Applications*. Englewoods Cliffs, NJ: Prentice Hall, 1994.
4. S. Parkes, P. Banerjee, and J. H. Patel, "ProperHITEC: A portable, parallel, object–oriented approach to sequential test generation," in *Proceedings of the Design Automation Conference*, (San Diego, CA), pp. 717–721, June 1994.
5. S. Parkes, P. Banerjee, and J. Patel, "A parallel algorithm for fault simulation based on PROOFS," in *Proceedings of the International Conference on Computer Design*, (Austin, TX), Oct. 1995.
6. V. Krishnaswamy and P. Banerjee, "Actor based parallel VHDL simulation using Time Warp," in *Proceedings of the 1996 Workshop on Parallel and Distributed Simulation*, (Philadelphia, PA), May 1996.
7. B. Ramkumar and P. Banerjee, "ProperCAD: A portable object-oriented parallel environment for VLSI CAD," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 829–842, July 1994.
8. G. A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA: The MIT Press, 1986.
9. A. W. Appel, *Compiling with Continuations*. Cambridge, England: Cambridge University Press, 1992.
10. A. A. Chien, *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs*. Cambridge, MA: The MIT Press, 1993.
11. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, May 1983.
12. J. G. Holm, A. Lain, and P. Banerjee, "Compilation of scientific programs into multithreaded and message driven computation," in *Proceedings of the Scalable High Performance Computing Conference*, (Knoxville, TN), pp. 518–525, May 1994.
13. L. A. Sanchis, "Multiple-way network partitioning," *IEEE Trans. Computers*, vol. 38, pp. 62–81, 1989.
14. L. V. Kalé and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Proceedings of OOPSLA '93*, Sept. 1993.
15. D. Gannon and J. K. Lee, "Object-oriented parallelism: pC++ ideas and experiments," *Proc. Japan Society for Parallel Processing*, pp. 315–339, 1993.
16. K. M. Chandy and C. Kesselman, "Compositional C++: Compositional parallel programming," in *Proceedings of Workshop on Compilers and Languages for Parallel Computing*, pp. 79–93, 1992.
17. High Performance Fortran Forum, *High Performance Fortran Language Specification, version 1.1.* Houston, TX, 1994.
18. C.-P. Wen, S. Chakrabarti, E. Deprit, A. Krishnamurthy, and K. Yelick, "Runtime support for portable distributed data structures," in *Workshop on Languages, Compilers and Runtime Systems for Scalable Computers*, May 1995.
19. R. Ponnusamy, J. Saltz, and A. Choudhary, "Runtime-compilation techniques for data partitioning and communication schedule reuse," in *Proceedings of Supercomputing '93*, (Portland, OR), pp. 361–370, Nov. 1993.