Center for Reliable and High Performance Computing, University of Illinois, Urbana, IL. Technical Report CRHC-93-04

ProperCAD: A Portable Object-oriented Parallel Environment for VLSI CAD*

Balkrishna Ramkumar
Dept. of Electrical & Computer Engineering
University of Iowa
Iowa City, Iowa 52242

Prithviraj Banerjee
Center for Reliable & High-Perf. Computing
University of Illinois
Urbana, Illinois 61801

Abstract

Most parallel algorithms for VLSI CAD proposed to date have one important drawback: they work efficiently only on machines that they were designed for. As a result, algorithms designed to date are dependent on the architecture for which they are developed and do not port easily to other parallel architectures.

This paper describes a new project under way to address this problem. We are developing a **Por**table **o**bject-oriented **p**arallel **e**nvi**r**onment for **CAD** algorithms (**ProperCAD**). The objectives of this research are two-fold. (1) To develop new parallel algorithms that run in a portable object-oriented environment. We accomplish this in two stages. First, we are developing CAD algorithms using a general purpose platform for portable parallel programming called CHARM [6, 12] developed at the University of Illinois. Second, we are concurrently developing a $C++$ environment that is truly object-oriented and specialized for CAD applications. (2) To design the parallel algorithms around a good sequential algorithm with a well-defined parallel-sequential interface. This will permit the parallel algorithm to benefit from future developments in sequential algorithms.

We describe one CAD application that has been implemented as part of the ProperCAD project: flat VLSI circuit extraction. The algorithm, its implementation, and its performance on a range of parallel machines are discussed in detail. It currently runs on an Encore Multimax, a Sequent Symmetry, Intel iPSC/2 and i860 hypercubes, a NCUBE 2 hypercube, and a network of Sun Sparc workstations. We also provide performance data for other applications that have been developed: namely test pattern generation for sequential circuits, parallel logic synthesis and standard cell placement.

1 Introduction

In view of the increasing complexity of VLSI circuits of the future, the requirements on VLSI CAD tools will continuously increase. Parallel processing for CAD applications is becoming gradually recognized as a popular vehicle to support the increasing computing requirements of future CAD tools. Recent research on parallel CAD applications have been reported for a wide variety of

*This research was supported in part by the National Aeronautics and Space Administration under grant NAG 1-613, and in part by the Semiconductor Research Corporation under grant SRC 91-DP-109.

applications such as placement [2, 13, 19, 20], floor planning [11], circuit extraction [3, 4, 14, 25], test generation and fault simulation [17], etc. Parallel processing for VLSI CAD has become a reality in industry as well. Hardware vendors such as Solbourne have already announced products with multiple CPUs in a single workstation. Software CAD vendors such as Mentor have announced products such as CHECKMATE, a parallel design rule checker using multiprocessing, to accelerate a single job. A major limitation with almost all such previous work is that the parallel algorithms have been targeted to run on specific machines like an Intel iPSC/2 hypercube or an Encore shared memory multiprocessor. Such work, although interesting, is not usable by the rest of the VLSI CAD community since the algorithms are not portable to other machines.

A second serious problem also presents itself in the design of parallel algorithms. The software development cycle for parallel algorithms is considerably longer than for sequential algorithms. This has two important implications. The first is that they are considerably more costly to develop than sequential algorithms. This is only exacerbated by the lack of portability across parallel machines. The second implication is a more pragmatic one. Given the fast pace of progress in the development and improvement of sequential algorithms for CAD applications, for a given application, sequential algorithms frequently outperform parallel algorithms due to the longer development time of the latter. For example, this is evident in parallel test pattern generation. The latest version of HITEC [16], a uniprocessor test pattern generation program for sequential circuits is already comparable in performance and is slightly better in quality of results than a recent parallel algorithm for test pattern generation [17].

A related issue in the development of parallel algorithms is that certain approaches are inherently parallelizable and others are extremely hard to parallelize. More often than not, the tradeoff between these two approaches is in the quality of results. Cell placement is a good example. The quadrisection algorithm [24] is easily parallelizable and is significantly faster than algorithms based on simulated annealing. However, it cannot produce results comparable to TimberWolf [22], a sequential program that uses simulated annealing (and a host of related tricks) to do cell placement. An interesting possibility would be to use a hybrid of these two (and possibly other) techniques, where, for example, quadrisection could be used for decomposition of the layout area into regions, and TimberWolf would be used for placement in a given region. However, to experiment with such techniques, it should not be necessary to rewrite the software entirely. Any attempt to rewrite TimberWolf [22] will not only be extremely time consuming, it is also unlikely to be comparable in performance. However, if it is possible to decouple the parallel and sequential algorithms and provide a well defined interface between the two, it may be practical to experiment with hybrid schemes such as these.

It would be presumptuous to assume that it will be trivial to interface the parallel algorithm with the sequential algorithm as described above. For this to be practical, it is imperative that sequential algorithms be written in a modular fashion. Fortunately, object-oriented programming in C++ (or even disciplined C programming) goes a long way in realizing this requirement. Many CAD vendors are already rewriting many of their well-established CAD applications using such disciplined, modular programming methods due to the benefits offered in program design and maintenance.

The most important questions that need to be addressed in the development of parallel algorithms are therefore: "How can we design parallel algorithms that are truly portable across parallel machines?", "How can we exploit good sequential algorithms in the design of parallel algorithms", and "How can parallel algorithms keep pace with future developments in sequential algorithms?" These are the main objectives of a new project to be discussed in this paper.

CHARM [6, 12] is a run-time support system for portable parallel programming developed at the University of Illinois. It currently runs on a wide range of parallel machines including shared memory machines, message passing multiprocessors and a network of workstations. We are using CHARM to build a prototype of a **Por**table **o**bject-oriented **p**arallel **e**nvi**r**onment for **CAD** applications (**ProperCAD**). Since inception, the ProperCAD project (see Figure 1) is designed to be completed in two phases. In the first phase, we are designing portable parallel algorithms for a large set of CAD applications using CHARM. To date, algorithms for flat extraction, test generation for sequential circuits [18] and combinational logic synthesis [5] and standard cell placement have been designed and implemented. New algorithms for global routing, fault simulation and behavioral simulation are currently under development.

The second phase of the project is expected to take a couple of years. It will involve the design and implementation of a run-time support system for portable parallel programming in C++. This system, although inspired by CHARM, will be tailored specifically for CAD applications. This will make the programming environment truly object-oriented and will support features like *inheritance* and *classes*. The ProperCAD applications will then be rewritten and ported onto the new C++ platform. The new platform will make it possible to adapt and integrate the parallel applications with software developed by companies like Cadence and Mentor, which are increasingly using C++ as a standard for their software development. Recall that reuse of sequential code is one of the primary objectives of the ProperCAD project. In the second phase of this project, we will also develop a library for the rapid prototyping and development of additional parallel CAD applications. The library will essentially be a parallel data manager that supports data distribution abstractions and primitives designed for an integrated parallel CAD environment. The library can be viewed as as being analagous to the Oct tools [9] distributed by the University of California at Berkeley for uniprocessor CAD applications.

The CHARM system was chosen as the platform for two significant reasons. The first is that it is a working prototype of a run time support system that offers true portability of parallel applications across MIMD machines. Second, although not truly object-oriented, it supports an object-oriented style of programming. This will make porting of the CAD applications to C++ much easier. We discribe the CHARM system briefly in Section 2.

In Section 3, we discuss how flat circuit extraction is expressed as an example of the use of the programming paradigm supported by the ProperCAD environment. The algorithm for circuit extraction presented in this paper has three significant contributions: **(1)** It is portable across MIMD architectures. **(2)** It is built around an existing sequential circuit extractor using a well-defined interface. This enables it to benefit from future improvements in the sequential algorithms for circuit extraction. **(3)** Unlike previous approaches to parallel circuit extraction, it uses an asynchronous coarse-grained data-flow model of execution. This is instrumental in rendering the parallel algorithm scalable on all the target machines. Contributions (2) and (3) together also permit good load balancing and high processor utilization.

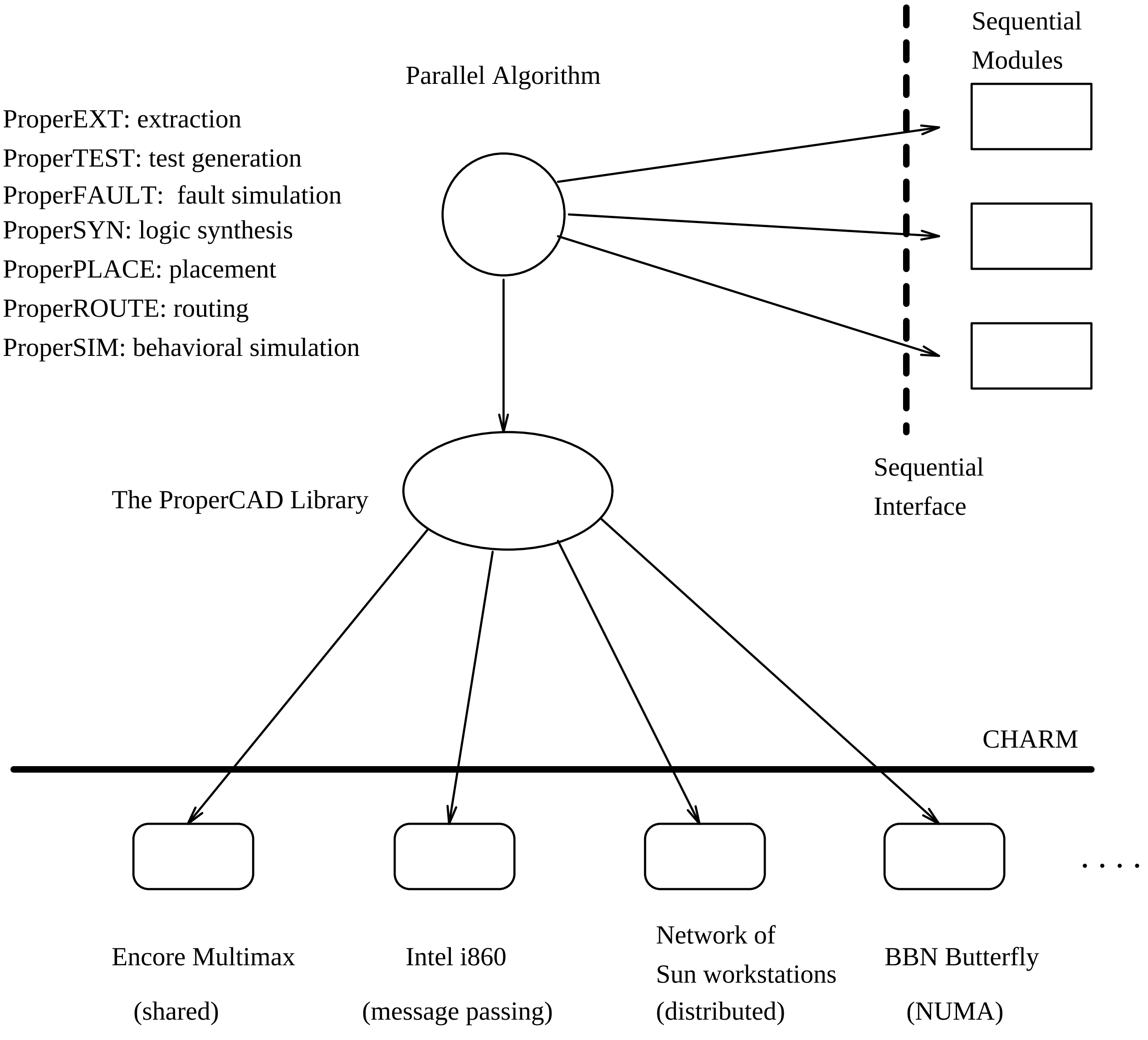


Figure 1: A high-level view of the ProperCAD project currently under development using the CHARM parallel programming environment. We list the CAD applications under development above.

2 The Parallel Programming Model

CHARM is a run time support system for *portable* parallel programming [6]. It abstracts away all machine dependent features away from an application program and provides a uniform set of primitives that can be used by the application to render their program machine independent. Features like dynamic process creation, mapping of processes to processors, dynamic load distribution and load balancing, scheduling, interprocess communication, are provided by the kernel. These are implemented in the most efficient manner possible on each of the machines that the kernel runs on. These features often complicate the user application considerably. CHARM helps the programmer separate these concerns.

The CHARM kernel supports a message-driven style of execution. Conceptually, it maintains a pool of messages that represent work. The application program can create processes dynamically by creating a message that represents a seed for a new dynamically created object.[1] Information can be exchanged between these objects also via messages. When a message is created it is put in the work pool. The messages in the work pool are distributed (and periodically balanced) across the available processors by the kernel. The kernel services messages in the pool until no more are available. Quiescence is detected; the programmer may take necessary action at this point (for example, printing results).

A CHARM program comprises a set of object definitions. Each object definition has a set of entry points which have C-code associated with them (see Figure 6 for an outline of the object for circuit extraction). Instances of these object definitions may be created dynamically at run time.[2]. Messages may be sent to these objects at one of its entry points, and the servicing of a message entails executing the code associated with the entry point sequentially. No interrupts or blocking (e.g. for synchronous receives) are possible within a code block associated with an entry point. Only one instance of a special object called the *main* object is created. It has special entry points for initialization and detection of quiescence. These do not have messages sent to them unlike normal entry points. The initialization entry point performs data and object initialization during

[1] An object is created when this message is serviced.

[2] These objects are similar to *actors* [1], a type of concurrent object. Wegner [26] categorizes *actors* to be active imperative objects. Note, however, that features like *inheritance* are not supported by CHARM.

startup. The quiescence entry point is optional; it permits the user to provide the action to be taken upon detection of quiescence. If it is absent, the program terminates.

CHARM also provides a special type of object called a *branch office*. *One* instance of the branch office object is created per processor. As with the main object, branches are initialized automatically on the local processors by executing code associated with an *Init* entry point. Both types of objects permit the declaration of persistent data that is visible only when executing any code associated with the object (or branch). Both types of objects also permit the declaration of procedures or functions as part of its definition. The functions in an object are private to the object, whereas the functions in a branch office may be invoked by other objects. Branch office objects are useful in providing data and program abstraction and have a concurrent object-like behavior. For example, the access to *distributed* data can be managed by branch office objects. A program may comprise several branch offices, each of which manages a different complex data structure (like a circuit, BDDs, etc.). An example of a branch office object definition is provided in Figure 2.

Another interesting feature of the CHARM kernel is *conditional* packing. The program definition also includes routines for packing messages into contiguous buffers and unpacking them into a representation used by the program. A pair of such routines are provided for each message type in the program for which packing/unpacking is necessary. These are used by the CHARM kernel on nonshared memory machines when it is necessary for a message to cross process boundaries. Note that for shared memory machines packing is not necessary. Hence, the algorithm runs efficiently on shard memory machines as well.

Other features provided by CHARM are beyond the scope of this paper. Only features that are important to the ensuing sections are discussed above. Further details may be found in [6].

The primary objective of a CHARM program is to create a large number of messages representing parallel work. Typically, this is done by decomposing the problem hierarchically into smaller and smaller subproblems which can be evaluated in parallel, until a threshold is reached. This threshold is user defined, and is used to indicate that subproblems smaller than the threshold are to be evaluated sequentially. Ideally, the threshold determines the point at which it is cheaper to solve a subproblem sequentially in preference to decomposing it further into parallel subcomponents. Determining this threshold accurately is not necessary as long as sufficiently large number of messages are created each of which represents a reasonable amount of work (e.g. > 50 ms). The more the messages available to the kernel, the better its capability to perform dynamic load balancing. The problem decomposition is thus independent of the number of processors available.

CHARM has been ported to a variety of shared memory and nonshared memory machines including the Encore Multimax, the Sequent Symmetry, the Alliant FX/8, the Intel iPSC/2 and i860 hypercubes, the NCUBE 2 hypercube, and a network of Sun workstations. It is currently being ported to the BBN TC2000 Butterfly multiprocessor. Four portable implementations of the CHARM kernel have been developed so far, one for shared memory machines, one for nonshared memory machines, one for NUMA type machines, and one for a network of workstations. Every time a new parallel machine is announced, the kernel can be ported to the new machine with relatively little effort.[3]

3 VLSI Circuit Extraction

VLSI circuit layouts are typically described as a collection of rectangles in different mask levels. The problem of circuit extraction is to take such a layout and determine the circuit connectivity, and obtain estimates for various electrical parameters such as resistance of lines, capacitances of nodes and dimensions of devices. The circuit extraction problem has two components: netlist extraction and parameter extraction. The first component involves determination of the electrically connected regions (called nets). To do this, boolean task manipulations are performed on different layers to derive new layers, as specified in a technology file. For example, in CMOS technology, N-type transistors are obtained by intersecting poly, diffusion and pwell layers, whereas P-type transistors are obtained by intersecting poly, diffusion and complement of pwell layers. A new diffusion layer is obtained by intersecting the old layer with the complement of poly.

The rectangles in the device layers are grouped into maximally connected groups, which form the devices. The rectangles in the other layers are grouped into maximally electrically connected sets, which form the nets. The electrical connectivity information is also provided in a technology

[3]This is true unless the architecture of the new machine is radically different to existing architectures. In this case, a new implementation of the kernel best suited to the architecture will be developed.

file as mentioned earlier. This gives the layers that electrically connect on overlap. For example, in CMOS technology, the metal and contact layers electrically connect on overlap, so do the diffusion and contact layers.

The parameter extraction component involves device size extraction, parasitic capacitance extraction and resistance extraction of nets. Different models for parameter extraction with varying accuracy and computational requirements have been proposed. The more accurate the model, the more computation intensive it becomes. The HPEX model [23] is used for the circuit extraction algorithm in this paper. For reasons of brevity, we do not discuss it further here.

Sequential circuit extraction is a well studied problem. Several sequential circuit extractors of varying speed and accuracy already exist [7, 8, 10, 15, 21, 23]. Parallel algorithms for circuit extraction have also been recently proposed [3, 4, 14, 25]. These algorithms perform parallel circuit extraction in several phases, including a data distribution phase, a geometric extraction phase, a merge phase, a device extraction phase and a parameter extraction phase. Such approaches involve synchronization at the start of each phase of the execution. For example, it is necessary to uniquely determine the nets and transistors before proceeding to the parameter extraction phase. This reduces the processor utilization, especially on nonshared memory machines.

To improve the load balancing, two schemes were proposed for data distribution: area-based partitioning [3] and point-based partitioning [4]. The former partitions the circuit into different areas each of which was assigned to a processor which performed local netlist and transistor extraction for its region. This can result in load imbalance if certain areas of the circuit are denser than others. This drawback is addressed by a point-based partitioning scheme which which partitions the circuit so as to approximately assign an equal number of rectangles to each processor. This is costlier and more complicated than the area-based scheme, but yields better results for circuits that do not have its rectangles evenly distributed. In the final phase, the complete nets are also distributed across the available processors for load balancing reasons. These load balancing schemes adopted were different for shared memory [4] and message passing machines [3].

The ProperCAD approach requires us to design programs that are not tailored to a particular type of architecture. It also encourages the use of a coarse-grained data-flow style of execution where a operation can be executed as soon as the data necessary to execute it is available. In parallel circuit extraction, we adopt a hierarchical approach to decomposition. The circuit is partitioned into several regions and each region is assigned to a processor. (The details of the data distribution are provided in Section 3.1.) A *sequential* algorithm for local geometric extraction is then run on each the regions to determine the nets and transistors in that region. The nets and the transistors touching a border of the region they belong to are deemed to be incomplete. Incomplete nets and transistors are subject to a merge algorithm, whereas local nets and transistors are available for processing using a *sequential* algorithm for parameter extraction.

The merge algorithm proceeds in a hierarchical manner where at each stage two adjacent regions are merged. Following every stage of the merge algorithm, nets and transistors that become *complete* are available for parameter extraction. Nets that are available for parameter extraction are load balanced *as and when* they become complete, to ensure maximum utilization of processors.

As can be seen in the above brief description of parallel circuit extraction, the geometric extraction on each region as well as the parameter extraction are performed using a *sequential* algorithm. It is easy to see that the best sequential algorithm can be used for this purpose. The focus of the parallel algorithm is now simply that of **(1)** decomposing of the problem into subproblems, and **(2)** merging these subproblems together.

However, how can the overhead of parallelization be kept in check? To see this, consider this simple argument. Based on a property of trees, for a branching factor ≥ 2, the number of leaves in a tree is always greater than the number of internal nodes. If it is possible to ensure that the work done at a leaf node in a problem decomposition tree is atleast 10 times the work done at an internal node, the work done at the leaf nodes of the decomposition will dominate the execution time ($>$ 90%). Thus, for circuit extraction, if we can conform to this rough criterion for decomposition, by calling the best available sequential algorithm for geometric extraction and parameter extraction, the total overhead of the parallel algorithm can be bounded to within 10% of the best sequential algorithm.

The above argument has been simplified somewhat for ease of explanation. However, the conclusion is still valid. We demonstrate this in Section 3.6 where we discuss the performance of the parallel circuit extraction algorithm. In the following discussion, we describe the different phases of circuit extraction in more detail and how our algorithm avoids synchronization between

```
readonly int grainsize;

branch office RectangleManager {
HashTableEntry data-distribution[MaxHashSize];
int initmsgcount;
LocalPartitionTree mypartition;

    entry Init:
        Compute the circuit partitioning to determine which
            processor gets which region. The grainsize
            determines the depth of the recursive partitioning

    entry ReceivePartition: (message InitRectangles *msg)
        Receive and insert received rectangles into local partition tree

    entry ReceiveRectLoad: (message LoadMsg *msg)
        Receive the current rectangle load on other processors

    entry SendRectangles: (message SendRequest *msg)
        Send some of local rectangles to processors with less load

    }
    :
RequestRectangles(region)
    :
Continue(region, decompose)
    :
    :
/* other functions visible to other objects */
} /* RectangleManager */
```

Figure 2: The branch office object for data distribution of rectangles.

these phases.

3.1 Data Distribution

In order to effect proper load distribution for parallel circuit extraction, it is important to ensure balanced data distribution. This needs to be accomplished with minimum overhead during data distribution, and at the same time, it must not complicate the merge phase of the circuit algorithm which combines the results computed for each of the partitions of the circuit, as was the case in the point-based partitioning in the PACE algorithm [4].

The distribution of rectangles is implemented using the branch office object outlined in Figure

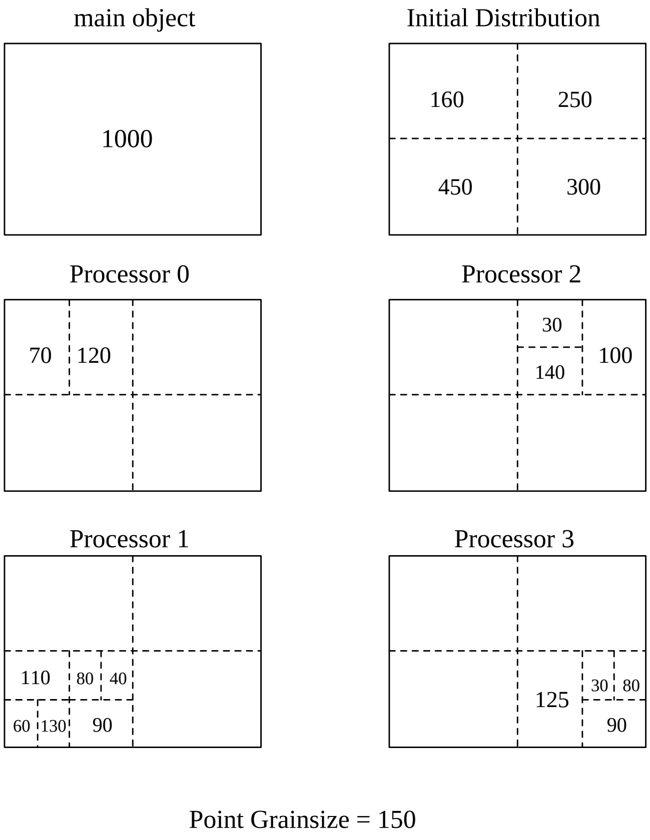


Figure 3: A simple example illustrating the data distribution for 4 processors. Note that the number of rectangles in an region is smaller than the sum of the rectangles in its 2 subregions since border rectangles are given to both subregions.

---

2. It also provides access routines to the distributed data. These routines are used as necessary by the dynamically created objects in the system. In Figure 2, the C-code associated with the *Init* entry point is executed on every processor upon creation of the branch office. A *main* object reads in the circuit description and partitions the rectangles area-wise into n partitions, where n is the number of available processors. The partitions are sent to the respective processors to the *ReceivePartition* entry point. The rectangles are locally partitioned further based on a user-defined threshold and the local load is broadcast to the *ReceiveRectLoad* entry point of the sibling branch offices on the other processors. The branch offices then determine the best distribution of the local partitions across the available processors. Some of the processors with surplus rectangles tag some

of their local regions as those to be processed by other processors. No movement of rectangles takes place at this point. We discuss how this is accomplished below in more detail.

Initially, the circuit is partitioned using an area-based partitioning scheme so as to assign a region of the circuit to every processor. The rectangles comprising the circuit from a file are read in and sent to the processor owning the circuit partitions to which they belong. A processor will also get all the rectangles that touch a border of the circuit region it owns. These rectangles are sent in several ‘rectangle’ messages to overlap the processing of these rectangles with the reading in of the input.

Upon initialization, on each processor, a hash table is created to store the data distribution. This table is used to store all the nodes resulting from the initial area-partitioning nodes, together with local nodes resulting from the local point-based partitioning in the parallel circuit partition tree (see Figure 3). Each processor also initialize two counts: a count of the number of ‘rectangle’ messages it expects to receive (*init-msg-count* = 1) and a count of the number of messages it expects to receive from the other processors indicating the local point-based distribution on the respective processors: (*rect-load-count* = *num-processors* -1). Every message except the last sent to the processors as the circuit is being read in carries a *send-count* field = zero. In the last message, however, the *send-count* field is set to number of messages sent + 1. Upon receipt of a message, a processor increments its *init-msg-count* by 1 and decrements it by *send-count*. This ensures that *init-msg-count* is zero if and only if all the messages have arrived, irrespective of the order of arrival. The use of *rect-load-count* is described below.

As and when the messages are received, the rectangles in the message are inserted into a partition tree. The root of the partition tree on every processor is the entire region owned by the processor. Initially, the root of the tree is the only node in the tree. Rectangles are only stored at the leaf nodes of this tree. When the number of rectangles at a leaf node L of the tree exceed a user defined limit (called *point grain size*), the region represented by L is split into two. Two leaf nodes L_1 and L_2 are created as children of L, and the rectangles stored at L are distributed between L_1 and L_2 (the rectangles on the border are given to both regions). Thus, when all the rectangles from the *main* process have been received and processed, every leaf node has $\leq$ *point grain size* rectangles. One triple (*region, penum, rectangle-list*) for every node in the partition tree is stored in the local data-distribution hash table. (The center of the region is used as the key to index the hash table.). This constitutes the *local* phase of data distribution.

Once all the rectangles bound for a processor p have been received and processed, a message containing the *number* of rectangles owned by p is broadcast to the other processors. This number typically exceeds the number of rectangles received when the circuit was read in because it accounts for the duplication of "border" rectangles. Note that no rectangles are sent across processor boundaries at this time. Each processor will receive *num-processors* -1 such messages. The local *rect-load-count* field is used to check the arrival of all such messages at a given processor. When all these messages arrive, processors having more rectangles than the average assign some leaf regions to lean processors. This is done by accessing the local hash table and changing the *penum* field in the triple (*region, penum, rectangle-list*) appropriately. The rectangles are *not* sent to the lean processors at this stage. Moreover, Care is taken to ensure that several surplus processors do not all assign rectangles to a same lean processor but distribute it across the lean processors uniformly[4].

After a processor receives its rectangles, creates its local partition tree, and broadcasts the number of rectangles to other processors, it is ready to begin the decomposition phase (Section 3.2). It does *not* wait for the receipt of all *rect-load-count* messages from other processors.

The hash table, the partition tree and count information is all managed by a local data object on each processor. These processes together provide a form of distributed data abstraction to the processes created during the execution of the circuit extractor (see below).

3.2 The Decomposition Phase

Once all the rectangles have been read in and sent to the respective processors, an object responsible for the entire circuit area is created. The object is named the *CircuitExtractor* object in Figure 4.

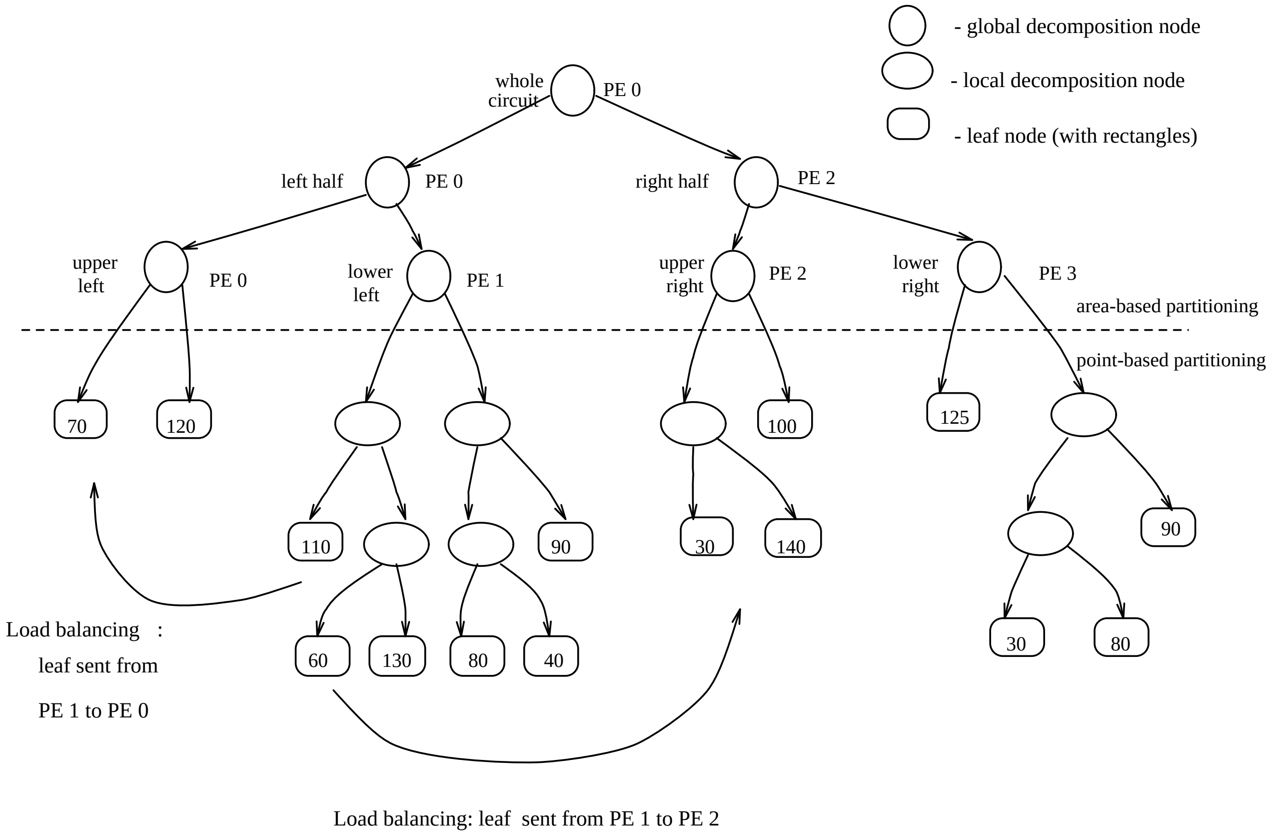
In Figure 4, an outline of the object used to perform circuit extraction is shown. Briefly, decomposition continues until the user-defined threshold is reached. This decomposition and the corresponding creation of objects mirrors the data partitioning performed by the branch office. When decomposition stops, the *RequestRectangles* function of the local branch office is queried for the rectangles in the specified region, with a "reply-to" entry point = *ReceiveRectangles*. If these

[4] No additional messages are sent to accomplish this. Each processor runs a local deterministic algorithm on the periodic load information received from the other processors.

```
chare CircuitExtractor {
    LocalDataType data;
    ObjectIDType  parentid;
    int           numchildmsgs;
    BorderMsg     *firstchildmsg;

    entry Decompose: (message CurrentRegion *msg)
    {
        If ( LoadManager.Continue(msg->region, &decompose))
            If (decompose)
                Divide msg->region into two equal regions by bisecting its longer sides
                Create 2 CurrentRegion messages to represent these regions
                CreateChare(Decompose@CircuitExtractor, msg1, pe1)
                CreateChare(Decompose@CircuitExtractor, msg2, pe2)
            Else
                parentid = msg->objectid; numchildmsgs = 0;
                LoadManager.RequestRectangles(msg->region);
    }
    entry ReceiveRectangles: (message RegionRectangles *msg)
    {
        ConstructLocalRectangleLists( msg->rectangles, data );
        Identify Local Connected Nets netlist and Transistors tranlist
        ProcessTransistors(tranlist, &bordertranlist, &localtranresults, data);
        ProcessNets(netlist, &bordernetlist, &localnetresults, data);
        Report local transistors
        Insert complete nets in netlist in LoadManager:localnets
        Create a message bordermsg containing all border net and transistor info.
        SendMsg(CircuitExtractor@MergeRegions, bordermsg, parentid);
    }
    entry MergeRegions: (message BorderMsg *msg)
    {
        numchildmsgs = numchildmsgs + 1;
        If (numchildmsgs == 2) /* both messages received */
            MergeRegions( firstmsg, msg, data);
            Identify Local Connected Nets netlist and Transistors tranlist
            ProcessTransistors( tranlist, &bordertranlist, &localtranresults, data);
            ProcessNets(netlist, &bordernetlist, &localnetresults, data);
            Report local transistors
            Insert complete nets in netlist in LoadManager:localnets
            Create a message bordermsg containing all border net and transistor info.
            SendMsg(CircuitExtractor@MergeRegions, bordermsg, parentid);
        Else
            firstchildmsg = msg;
    }
}
```

Figure 4: The object that implements the circuit extraction algorithm. Some liberties have been taken with notation for ease of exposition.



Initial Distribution:	Final Distribution:
PE 0: 190 rectangles -134	PE 0: 300 rectangles -24
PE 1: 510 rectangles $+510$	PE 1: 340 rectangles $+14$
PE 2: 270 rectangles -54	PE 2: 330 rectangles $+6$
PE 3: 325 rectangles $+1$	PE 3: 325 rectangles $+1$

average : 324 rectangles
Grain size = 150
Tolerance -50 to $+50$

Figure 5: The dynamic redistribution of objects to load balance the rectangles in Figure 3. Each node in this decomposition tree represents an *extractor* object.

rectangles are not available locally, the local branch office first determines the owner processor for the requested rectangles. It then sends a message requesting the rectangles to the *SendRectangles* entry point of its sibling branch office owning the rectangles. The owner branch office sends the requested rectangles to the *RequestRectangles* entry point of the requesting object.

A *CircuitExtractor* object queries the local branch office to determine whether further decomposition is necessary. This is determined as follows. If the region owned by the *Circuitextractor* object (its *current* region) is not present in the local hash table, further decomposition is deemed necessary. If the current region falls within the circuit region owned by the local processor, it is necessary to wait until the *local* data partitioning phase is complete. This is accomplished with a *do not* continue response. The querying process suspends and relinquishes the processor upon receiving a *do not* continue response. When local data partitioning is complete, these processes are woken up and this query is retried. If further decomposition is required, the hash table provides the processors on which the child process instances are to be created. Recall that the destination of leaf processes may change due to distribution of rectangles as explained in Section 3.1. Non leaf processes resulting from the initial area-based partitioning are assigned processors statically. A non-leaf processes resulting from point-based partioning is created on the same processor on which its parent resides (see Figure 5).

If the information necessary to answer the query is available locally, the data distribution hash table is checked for an entry corresponding to the current region. Recall that only hash table entries for leaf nodes in the decomposition tree carry rectangles. Hence, if the entry in the hash table has no rectangles, the process may continue execution, but must decompose the current region further. This is accomplished by dividing the current region into two equal parts by bisecting its longer sides. An instance of the *CircuitExtractor* object is created for each of these regions.

If the current region falls within a region owned by another processor, no further decomposition is necessary since only *leaf* nodes in the decomposition tree may cross processor boundaries for load balancing. The process then requests the rectangles that belong to its current region and relinquishes the processor. The local data process sends the necessary data and wakes up the requesting process. Sometimes, it may be necessary for the local data object to forward the query to a data object on another processor to satisfy the request. In this case, the data object on the processor owning the rectangles sends the rectangles to the requesting *CircuitExtractor* object. This is done transparently as far as the requesting *CircuitExtractor* object is concerned.

The requesting *CircuitExtractor* object is now primed for local processing. Due to the large sizes of the messages exchanged between processes, processes created during area-based partitioning (see Figure 5) are mapped onto processors so that a parent *CircuitExtractor* object and one of its children reside on the same processor.

3.3 The Local Extraction Phase

We first describe the local processing performed with the assumption that all nets and devices computed are completely local to the region. We then discuss how nets and devices touching the border are handled.

The local processing of a region of the circuit is very similar to that employed by the PACE algorithm. To avoid repetition, we describe it very briefly here, emphasizing the differences between the two algorithms. A scan line algorithm is used to determine the local connected components and to identify nets and transistors. This forms the netlist extraction component of the algorithm. The output of this component is a list of devices and a list of nets. A device is described as a collection of device rectangles. For each device, information about the nets connecting to the different terminals of the device is also computed. A net is also described as a collection of rectangles. For each net, information about devices that connect to the net is computed.

For parameter extraction, we use the resistance-capacitance model used in HPEX [23]. The resistance of a net are converted into a horizontally maximal non overlapping form. This is also accomplished by a scan line algorithm. This will produce a unique representation of the net. The horizontally long rectangles are then combined in the vertical direction. Two rectangles that are sufficiently longer in the x direction than the y direction are combined vertically if they abut on their horizontal edges. Once this is done, for every rectangle R that is longer in one direction than other rectangles abutting it, R's larger side is cut at the point of intersection with the abutting rectangles. Two overlapping rectangles resulting from such intersections are merged.

Two rectangles are said to be electrically connected if they abut each other. A rectangle that connects to only one other rectangle, or atleast three or more rectangles is defined to be a *knot.*

Rectangles associated with the terminals of devices are defined to be *ports*. The remaining rectangles all have exactly two connections each and are defined to be *branch* rectangles. Every knot and port is assigned a globally unique number which identifies a point of connection in the circuit. A net can be thought of as an undirected graph where the knots and ports are nodes that are connected to each other by edges which represent a chain of one or more branch rectangles.

Resistance calculations are then performed for every edge in the graph representing a net. The contribution of the knots and ports are also factored into the calculations. Capacitance calculation is also performed at the same time. The capacitance of each of the knots and ports is first determined. The capacitance of each of each edge in the graph is computed by adding the contributions of all the branch rectangles on the edge. This capacitance is equally divided between the two end points. The result of this phase is a distributed RC network for each net. Currently, we do not perform a node reduction phase on the resulting network as is done in HPEX [23], but these features can be included easily.

For parallel execution, the maximally connected nets and devices are identified as described above. Following that, they are subject to the horizontal and vertical transformations and unique identification of knots and ports. However, both nets and devices may touch a border. All knots touching a border are marked as *border* rectangles. Rectangles abutting an incomplete transistor are also treated like border rectangles. Furthermore, for every edge in the graph representing a net, if any rectangle on the edge (including the end points) touches the border, all the rectangles including the end points are marked. Only the marked regions of a net will be sent to a parent process during the merge phase. The resistances and capacitances are computed locally for the non-marked regions of the net and the rectangles are then discarded.

Resistance and capacitance calculations are computed as before with some exceptions. **(a)** Capacitance is not computed for marked rectangles. **(b)** Resistance is not computed for marked rectangles. **(c)** Resistances and capacitances are computed for edges that are not marked even if one or both of their end points are marked. In this case, the computed capacitance is divided equally between the end points as described earlier. Resistances are reported immediately using the unique node identifiers assigned to their end points. Marked end points carry the partially computed capacitances and their unique node identifiers up the decomposition tree until they become unmarked. The results are reported at this stage. Note that that same unique identifiers will be used to report the capacitances. (**d**) Knot rectangles that are marked but do not touch the border are tagged as knots to avoid being identified as branch rectangles in an ancestor region.

The marked regions of a net together with partially computed resistances and capacitances at the knots not touching a border are sent up to the parent process. The computed values for the unmarked regions of a net are reported with a globally unique number identifying the net. However, if the reported results correspond to an incomplete net, the results are tagged as incomplete.

Local transistors, like local nets, pose no problem. Local transistors are reported as soon as they are encountered and processed. Border transistors, however, can pose potential problems. For each complete transistor, unique node numbers identify the gate, source and drain terminals. One node is created to represent the gate of the transistor and is tagged as a poly port. Two additional nodes are created for the source and drain nets of a transistor and tagged as diffusion ports. A rectangle bounding the channel rectangles of the transistor is used to represent the diffusion ports. This guarantees that the relevant sections of the source and drain nets will be marked as border rectangles if the transistor touches the border. The terminals of a transistor are only determined once a transistor is complete. The channel rectangles of an incomplete transistor are sent up to the parent process together with references to all nets abutting it. Unlike nets, device extraction results are only reported once the device is complete.

We illustrate this with the help of the simple example circuit in Figure 6. Figure 6(a) describes an entire circuit in a region R_0 in a form ready for parameter extraction if executed on one processor. We consider the case where it has been divided into two regions R_1 and R_2 (Figure 6(b)) which are processed by two processors. Figure 6(c) describes the state after local netlist and device extraction. Transistors T_1 and T_2 are recognized as incomplete on their respective processors. The determination of the source and drain nets is deferred until the transistor becomes complete. The regions of the nets that are connected to the common border are marked as shown. The resistance and capacitances for the unmarked regions are then computed. For example, the capacitance computed and lumped at knot K_2 is reported (as belonging to net 2). The resistances between K_1 and K_2, K_2 and K_3 etc. are also reported. The partial capacitances computed at knots K_1, K_3, K_5 and K_6 together with the node identifiers assigned to them are sent up to the parent process

Figure 6: A simple example illustrating the behavior of the parallel circuit extraction algorithm.

responsible for R_0. No resistance or capacitance values are computed for knot K_4.

As soon as a leaf process completes local geometric extraction, it processes the local complete transistors, deposits the local complete nets in a *local nets* list in the local data object, and sends a message to its parent containing the border nets and transistors. It then terminates and frees up any allocated memory.

3.4 The Merge Phase

At the leaf processes of the circuit decomposition tree, four lists of border rectangles are created, one for each border. For efficiency, these rectangles are chopped to line segments along the border. Overlapping line segments are then combined to reduce the number of line segments sent up to a parent process. Note that two adjacent regions will have the same set of line segments on the common border since rectangles touching the common border are made available to both the regions. These line segments point to the net or device that they belong to.

When a non-leaf process in the decomposition tree receives one message each from its children, it is ready to begin processing them. The common border between the two child regions is determined and the set of line segments corresponding to the respective borders of the child regions are sorted. The nets or devices corresponding to the same line segment in the two lists are merged.

Merging two nets will result in one net subsuming another. However, the child process may have reported results corresponding to local parts of these nets using the globally unique identifiers given to these nets during the local extraction. Thus, the resulting net creates a list of net identifiers of all nets it subsumes. A net may subsume more than one other net during the merge process. This suggests that a list of merged net identifiers need to be created in the general case. It is also possible for two nets, that have subsumed one or more nets each, to be merged. This means that corresponding list of merged net identifiers also need to be combined when two nets are merged. Once the merge operation is completed at an internal node in the decomposition tree, for every net N resulting from the merge, the list of nets subsumed by N is reported.

For transistors, the list of channel rectangles of the transistors being merged are combined and carried by the resulting transistor. As mentioned during the local extraction phase, it is necessary to keep the information relating to the connecting nets consistent. Thus, in Figure 6, when Net 1 subsumes Net 2, Net 2 is marked as invalid and then carries a reference to Net 1, the net subsuming it. This is done by merging the list of abutting nets for the two transistors. For example, after the merge of transistors T_1 and T_2 in Figure 6, the resulting transistor T_1 will carry references to Net 1, Net 4 and Net 5. This is done by following references to subsumed nets until a reference to a net that has not been subsumed by another is found. If a transistor becomes complete (i.e., does not touch a border) following a merge, port nodes are created for the gate, source and drain nets to enable computation of the capacitances at the terminals of the transistor.

All border line segments of the other borders of the two regions being merged are then combined as necessary to create the border line segments of the new parent region. The nets and devices pointed to by some of these border segments may have become subsumed by another net; this is also resolved during the creation of the border segments of the new region. The subsumed nets and devices are then released.

The partial nets that are produced as a result of the merge operation are then processed as in the local extraction phase. Once again, some partial results may be computed, and some rectangles may be marked as border rectangles. This is identical to the local extraction phase with the caveat that only the nets involved in the merge operation are processed. Incomplete nets that are sent up by the child processes, but do not touch the common border, are not processed at this point.

Once processing is complete, local results are reported. Results that correspond to complete nets and transistors are tagged as such. Other partial results on nets are reported but tagged as incomplete.

Thus, at each level of the circuit decomposition tree, the partial nets sent up get smaller and smaller since parts of them become eligible for processing, and the capacitance and resistance computation for these parts is performed. Only the rectangles of a net marked as border rectangles are sent up to the parent for further processing.

In Figure 6(d), we see the result of the merge phase on the simple example. K_1', K_3', K_5' and K_6' denote knots in the child process carrying partial resistance and capacitance information. Note that net 2 is a disconnected net at this point. A new knot K_7 is created after merging nets 1 and 3. The identifier of the new net is arbitrarily assigned to be one of the two nets. Thus Net 1 subsumes Net 3 and net 5 subsumes Net 2. Once the merge is complete, the gate, source, and drain nets of

```
branch office NetlistManager {
Nets *longnets;
Nets *localnets;

    entry Init:
        Initialize longnets and localnets to NIL

    entry ReceiveNetLoad: (message LoadMsg *msg)
        Receive the current net load on other processors

    entry ReceiveNets: (message NetList *msg)
        Receive nets that need to be processed and add it to local lists

    entry ProcessNet: (message DummyMsg *msg)
    {
        Process one net in localnets list and report results;
        SendMsg(ProcessNet@NetlistManager, msg, MyPeNum());
    }
    :
/* other operations visible to other objects */
    :
    :
} /* NetlistManager */
```

Figure 7: The branch office object for distribution and parameter extraction of nets.

the complete transistor T_1 are determined. (These nets may not be complete at this stage. Only the region abutting the transistor has to be complete.) The computed values at knots K'_3 and K'_6 are reported for net 5 together with the information that nets 1 and 5 have subsumed nets 3 and 2 respectively.

3.5 Dynamic Load Balancing

In Section 3.1, we described the algorithm that is used to effect good data distribution. This, however, only serves to distribute the effort of local extraction effectively across the available processors. As described in Section 3.3, the merge algorithm may detect and identify completed nets at different non-leaf nodes in the circuit partition tree in Figure 5. The device and parameter extraction of these completed nets can be the most time consuming part of the execution, especially if a computationally complex model for resistance/capacitance computation is used.

In the PACE algorithm [4], all completed nets are identified in one phase. They are then randomly distributed across the available processors. In our algorithm, completed nets may be identified during the local extraction phase as well as every stage in the merge phase. The processing of these nets is done as and when they become available. Moreover, in our approach, we also process 'completed' regions of incomplete nets during the merge phase to the extent it is possible to do so uniquely. As a result, a more interesting load balancing scheme is used. Figure 7 outlines the branch office object used to manage netlists and load balance them across the available processors.

Nets are identified as large or small based on a user-specified value for the number of rectangles contained in the net. All small nets are initially retained on the processor on which they were identified in a *local nets* list in the local data object. One process is created to perform device and parameter extraction on each long net as and when it is identified and randomly assigned to a processor.

During execution, on each processor, a count of the total number of rectangles in completed small nets is maintained. In addition, whenever a long net process is picked up, the number of rectangles in the long net is added to this count. Each processor periodically broadcasts this count to all the other processors.

Upon receipt of the rectangle counts on the other processors, each processor independently runs a simple balancing algorithm to determine the best distribution of rectangles within a predefined tolerance limit. Note that all processors will arrive at the same distribution since they run a deterministic algorithm on the same data. This identifies the *donor* and *recipient* processors as was done for data distribution. Care is taken to match donor and recipient processors to ensure that load distribution is even. A donor processor then sends a *set* of small nets to the appropriate recipient processor. Recipient processors do not take any action during the load re-distribution stage. Any nets received by a recipient object are added to the *local nets* list in the local data object.

An interesting feature of the the ProperCAD environment is that it provides support for prioritized execution of objects. To ensure effective load balancing, priorities are assigned to the different phases of execution. Messages that periodically exchange the load information (the rectangle count for extraction) get the highest priority to ensure prompt action upon detecting unbalanced load. Leaf *CircuitExtractor* objects get the next highest priority. When no leaf objects are available, non-

Table I: The characteristics of the benchmark circuits used.

Circuit Characteristics			
Circuit	Rectangles	Nets	Transistors
Prog. Logic Array	25384	912	2508
Hypercube Router	52893	1744	3476
Multiplier Array	64031	4227	8384
Static Ram	128073	5136	14296
Placement Coprocessor	253556	10266	28494

Table II: Execution times (in seconds) of benchmark circuits on the Network of Sun (SPARC) workstations. These data are subject to wide variations due to context switching between unix processes and network traffic. The data here is provided primarily as a proof of concept.

Network of Sun workstations				
Circuit	Sequential	1 PE	2 PEs	4 PEs
PLA	20.1	22.6	12.3	6.8
H. Router	65.2	73.3	56.2	34.5
M. Array	74.4	82.2	71.5	41.2
S. Ram	107.8	122.8	100.0	66.2
Coprocessor	–	–	–	314.8

leaf objects are picked up. Long nets get priority lower than *CircuitExtractor* objects but higher than short nets. In this way, local pools of short nets can be used to maximum effectiveness to correct any load imbalance that may be recognized.

3.6 Performance

We can now demonstrate the performance of the ProperEXT circuit extractor on a variety of parallel machines. Table I lists the benchmark circuits used in the experiment and their characteristics. The benchmarks used were real circuits some of which had been designed as projects for a graduate course in VLSI design. These circuits are the same as those used in [3, 4] to demonstrate the performance of the PACE algorithm. Circuits ranging in size from 25000 rectangles to 250000 rectangles were used. The largest circuit had over 32000 transistors. The circuits were all in hierarchical CIF format and were flattened before data distribution. Tables II – VI report the

Table III: Execution times (in seconds) of benchmark circuits on the Encore Multimax.

Encore Multimax				
Circuit	PACE Algorithm		ProperCAD Algorithm	
	1 PE	8 PEs	1 PE	8 PEs
PLA			64.5	10.4
H. Router	196	43	211.8	29.4
M. Array	221	41	238.1	64.2
Ram	305	63	332.9	55.0
Coprocessor	691	137	723.7	124.6

Table IV: Execution times (in seconds) of benchmark circuits on the Sequent Symmetry.

Sequent Symmetry		
Circuit	1 PE	8 PEs
PLA	119.9	18.2
H. Router	409.7	51.7
M. Array	447.8	96.6
S. Ram	630.5	96.2
CoProcessor	1276.7	233.2

Table V: Execution times (in seconds) of benchmark circuits on the NCUBE/2 hypercube.

NCUBE/2 (hypercube)					
Circuit	4 PEs	8 PEs	16 PEs	32 PEs	64 PEs
PLA	25.7	16.3	8.0	6.8	4.4
H. Router	87.2	47.7	38.2	34.0	34.7
M. Array	–	97.1	84.3	80.5	–
S. Ram	–	–	41.8	34.0	28.6
CoProcessor	–	–	–	59.7	48.9

Table VI: Execution times (in seconds) of benchmark circuits on the Intel i860 hypercube.

Intel i860 (hypercube)				
Circuit	1 PE	2 PEs	4 PEs	8 PEs
PLA	13.0	6.6	3.7	2.4
H. Router	45.7	20.1	11.3	6.2
M. Array	51.0	31.1	16.2	20.1
S. Ram	–	37.7	19.2	12.6
CoProcessor	–	–	–	27.7

execution time in seconds on all these circuits. The reported times exclude the time for input and output. The grain size used for all the circuits was 500: i.e. the circuits were partitioned into regions containing 500 or fewer rectangles.

In Table II we report the performance of the ProperEXT circuit extractor on a network of 4 Sun Sparc 1 workstations each with 24MB of memory. Only workstations with identical configurations were used for the experiment. Only 4 workstations with the above configuration (and the requisite memory) were available for the experiment. Data is also presented in Table III for an 8-processor Encore 510 Multimax with XPC processors running UMax 4.3 operating system with 64MB of main memory. In Table IV, the results of running the extractor on a Sequent Symmetry with 8 Intel 386 processors and 32MB of memory are presented[5]. Table V provides data on a 64-processor NCUBE 2 at Sandia National Laboratories with 4 MB per processor[6]. It is important to emphasize that the circuit extractor ran *unchanged* on all these machines.

We now consider the performance on each of these machines. First, modest speedups were evident on the network of Sun workstations. Due to context switching by the Sun operating system, network traffic due to page faults and other workstations on the network, wide variations in the performance of the extractor were observed. Furthermore, in the presence of context switching between unix processes, the execution time across different workstations on the network was quite

[5]We thank Argonne National Laboratories for access to the Sequent Symmetry and the Intel i860 hypercube. The Symmetry had 26 processors. However, the presence of other users in addition to the limited available memory made it impossible to use more than 8 processors for our experiment.

[6]The NCUBE 2 at Sandia National Laboratories has 1024 nodes. However, due to heavy use, only 64 nodes were available at the time of running the experiment.

inconsistent. As a result, we present the results in Table II more as a proof of concept: that a network of suns can be used as a parallel machine to distribute the computation. Column 2 in Table II provides the time taken by a uniprocessor implementation of PACE [4]. In Column 3, the time taken by the ProperEXT extractor on one processor is presented. The difference between Column 3 and Column 2 represents the overhead of parallelization in the ProperEXT extractor. As can be seen by these two columns, the overhead of parallelization was approximately 10-12%. (As mentioned earlier, this overhead can be controlled by the programmer by specifying the grain size appropriately.)

In Table III, we compare the results of the ProperEXT circuit extractor with the PACE circuit extractor [4] on the Encore Multimax. In spite of the fact that the PACE extractor was designed and programmed specifically for the Encore Multimax, the ProperEXT extractor is marginally slower on one processor, but significantly faster on 8 processors for 4 out of 5 circuits. Data for the PLA circuit was not available for the PACE extractor. The ProperEXT extractor does not perform as well as the PACE extractor on the Multiplier Array circuit. This was observed to be due to the completion of a single large net at the root of the decomposition tree. Since few other nets are available for balancing the load, the processor performing parameter extraction on this net is the sole active processor at this time. In the PACE algorithm, no processing is done until all the nets are complete. This makes it possible to distribute the load across processors more effectively in this example. This approach proves significantly costlier on the other circuits, however.

In Tables V and VI, we demonstrate the performance of the ProperEXT extractor on the NCUBE 2 and Intel i860 hypercubes. Again, with the exception of the Multiplier Array circuit, the benchmark circuits exhibit good speedups on all circuits.

3.7 Varying the Grain Size

An important question that needs to be addressed is the importance of the choice of grain size. How does the programmer determine the right grain size to be chosen. In Figures 8 and 9 we study the effect of varying the grain size on the execution time. Two experiments are reported: one on a shared memory machine: a Sequent Symmetry with 8 Intel 386 processors and a message passing machines: an Intel i860 hypercube with 8 processors.

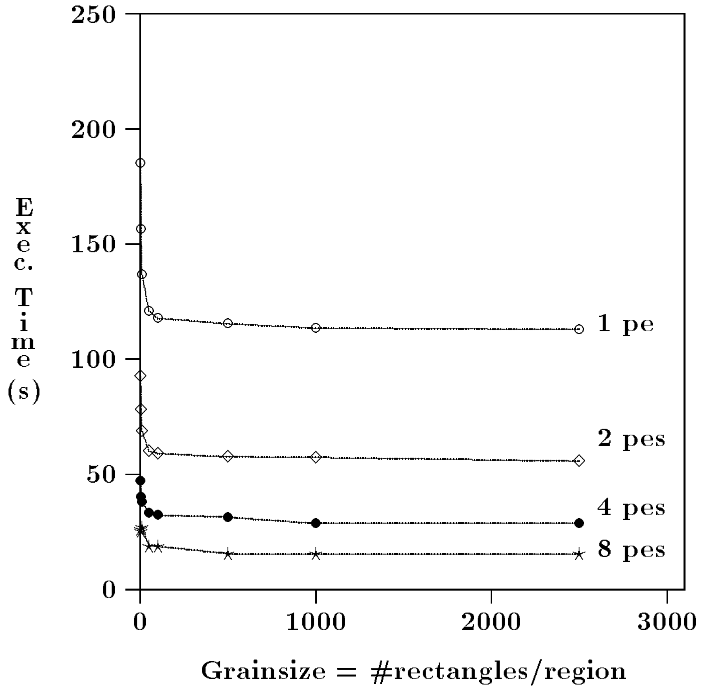


Figure 8: The effect of varying the grain size from 25 to 25000 rectangles per region for the PLA benchmark circuit on a shared memory machine: the Sequent Symmetry.

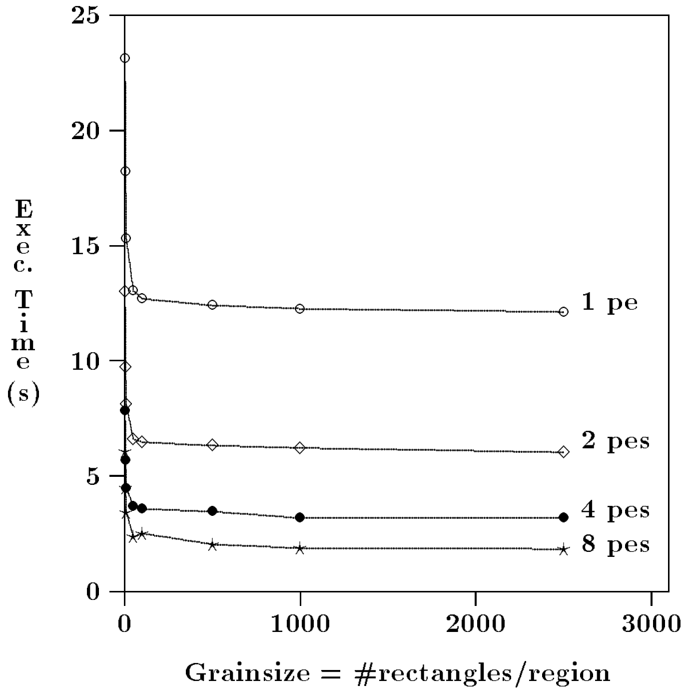


Figure 9: The effect of varying the grain size from 25 to 25000 rectangles per region for the PLA benchmark circuit on a message passing machine: the Intel i860 hypercube.

Table VII: Execution times (in seconds) of ProperTEST sequential test pattern generator running ISCAS89 sequential benchmark circuits on a network of Sun Sparc workstations

Network of Sun workstations (Distributed Proc essing)

Circuit	#PE s	Time/Flt	T.Gen.	F.Sim.	Overhead	Coverage	Efficiency	#Vectors	#Procs.
	1	15	311.1	3.0	2.1	81.8	100	333	1713
s386	2	15	148.0	1.9	-	81.8	100	293	1483
	4	15	89.4	1.8	-	81.8	100	361	1525
	8	15	39.2	1.2	-	81.8	100	369	1307
	1	2	42.2	3.9	1.9	81.6	97.6	182	720
s713	2	2	23.3	2.7	-	81.9	98.5	204	725
	4	2	11.1	1.8	-	81.9	98.5	236	735
	8	2	7.1	1.2	-	81.9	98.3	246	728
	1	2	8.8	13.2	1.8	99.8	100	365	1243
s1196	2	2	6.2	10.4	-	99.8	100	387	1243
	4	2	3.7	6.9	-	99.8	100	380	1243
	8	2	1.7	4.1	-	99.8	100	370	1243
	1	2	18.6	15.8	2.4	94.7	100	383	1363
s1238	2	2	11.5	10.9	-	94.7	100	374	1362
	4	2	7.0	7.4	-	94.7	100	398	1363
	8	2	3.7	5.2	-	94.8	100	406	1362
	1	1	1384.2	200.2	69.4	70.6	72.1	849	4604
s5378	2	1	899.6	113.8	-	70.5	72.0	929	4604
	4	1	523.3	72.4	-	72.3	71.8	950	4604
	8	1	298.7	49.6	-	68.6	70.2	1095	4604

We varied the point grain size (see Figure 3) from 25 to 25000. As the grain size is increased, the amount of parallelism exploited is reduced. For very small grainsizes, (i.e. < 100 rectangles per region), the execution time is quite high, indicating a high overhead of parallelization. However, as can be seen, a wide range in the grain size is observed for which the execution time exhibits little or no change. Thus, any choice of grain size within this wide range is suitable for executing the program.

4 Other Applications

Several other applications have already been developed as part of the ProperCAD project. They include test pattern generation for sequential circuits [18], combinational logic synthesis [5] and standard cell placement. Currently, all these applications have also been developed using the CHARM environment. As soon as the ProperCAD $C++$ environment is ready, these applications will all be reimplemented in that environmemt.

Table VIII: Execution times (in seconds) of the ProperTEST sequential test pattern generator running ISCAS89 sequential benchmark circuits on the Sequent Symmetry.

Sequent Symmetry (Shared Memory MIMD)

Circuit	#PE s	Time/Flt	T.Gen.	F.Sim.	Overhead	Coverage	Efficiency	#Vectors	#Procs.
s386	1	15	1164.4	15.1	20.4	78.4	96.6	255	1986
	4	15	413.8	9.1	-	78.6	96.9	330	2374
	8	15	236.7	9.8	-	78.9	97.1	399	2529
	16	15	143.5	5.5	-	78.9	97.1	403	2621
s713	1	1	41.6	25.9	5.7	81.9	95.9	206	582
	4	1	12.7	11.8	-	81.9	96.0	246	582
	8	1	7.2	8.6	-	81.9	95.9	282	582
	16	1	4.6	5.8	-	81.9	95.7	300	582
s1196	1	1	46.2	72.0	9.0	99.6	99.8	369	1303
	4	1	12.5	30.0	-	99.8	100	384	1305
	8	1	6.7	19.2	-	99.8	100	398	1310
	16	1	3.9	11.7	-	99.8	100	412	1331
s1238	1	1	85.3	87.3	9.3	94.5	99.0	376	1356
	4	1	23.2	35.8	-	94.5	99.0	383	1356
	8	1	12.2	21.0	-	94.5	99.0	394	1356
	16	1	7.7	14.8	-	94.6	99.0	442	1356
s5378	1	1	1648.4	1078.7	462.27	66.1	67.2	769	4604
	4	1	425.7	320.9	-	65.7	66.8	769	4604
	8	1	239.3	245.7	-	65.9	67.0	1090	4604
	12	1	181.2	198.7	-	65.4	66.5	1141	4604

Table IX: Execution times (in seconds) of the ProperTEST sequential test pattern generator running ISCAS89 sequential benchmark circuits on the Intel i860 hypercube.

Intel i860 hypercube (Message Passing MIMD)

Circuit	#PE s	Time/Flt	T.Gen.	F.Sim.	Overhead	Coverage	Efficiency	#Vectors	#Procs.
s386	1	15	184.4	2.7	1.8	81.8	100	330	1783
	2	15	87.0	1.9	0.7	81.8	100	306	1629
	4	15	49.0	1.6	0.3	81.8	100	370	1673
	8	15	28.8	1.3	1.8	81.8	100	418	1733
s713	1	2	27.0	3.7	1.5	81.9	98.8	217	751
	2	2	15.5	2.4	1.0	81.9	98.8	199	766
	4	2	8.8	1.2	1.7	81.9	98.8	213	761
	8	2	6.6	1.0	2.0	81.9	98.8	221	787
s1196	1	1	5.5	10.1	0.8	99.8	100	365	1243
	2	1	3.3	7.6	0.9	99.8	100	386	1243
	4	1	1.5	4.5	0.7	99.8	100	362	1243
	8	1	0.9	3.2	0.4	99.8	100	394	1243
s1238	1	1	11.7	12.3	1.3	94.7	100	383	1369
	2	1	6.3	8.7	1.0	94.7	100	385	1368
	4	1	4.1	5.4	0.8	94.7	100	387	1356
	8	1	2.2	3.7	0.6	94.8	100	406	1356
s5378	1	5	6016.5	184.8	140.9	73.4	75.3	985	13233
	2	5	3548.6	97.4	86.7	71.6	73.4	932	13727
	4	5	1748.8	59.3	51.3	72.5	74.3	1009	13511
	8	5	901.7	38.6	84.1	70.8	72.7	1228	13468

In Tables VII-X, we provide the performance of ProperTEST, the sequential test pattern generator based on the PODEM search algorithm on a variety of parallel MIMD machines. The benchmark circuits used were the standard ISCAS 89 sequential circuits. For reasons of space, results on a subset of the entire benchmark suite are presented.

In Tables XI-XIII we present the performance of ProperSYN, a portable combinational logic synthesis algorithm that is based on the transduction method. The benchmark circuits used were the standard MCNC combinational circuits. Once again, like the other CAD applications, the programs run unchanged on all the target machines.

Finally, in Tables XIV-XVI we present the performance of ProperPLACE, a portable parallel algorithm for standard cell placement using simulated annealing. The parallel algorithm is built on top of TimberWolf 6.0, one of the most widely used sequential programs for standard cell placement based on simulated annealing. The results reported are the best for standard cell placement among parallel algorithms that preserve both the quality of the results and yet obtain speedups on parallel

Table X: Execution times (in seconds) of the ProperTEST sequential test pattern generator on ISCAS89 sequential benchmark circuits on the Encore Multimax.

Encore Multimax (Shared Memory MIMD)

Circuit	#PE s	Time/Flt	T.Gen.	F.Sim.	Overhead	Coverage	Efficiency	#Vectors	#Procs.
s386	1	15	740.8	11.5	17.7	81.2	99.5	316	2350
	2	15	388.8	7.9	11.9	80.5	98.7	322	2431
	4	15	230.3	5.4	5.7	81.0	99.0	375	2628
	8	15	138.8	4.9	1.6	80.5	98.7	370	2907
s713	1	2	53.9	17.0	9.2	85.7	97.2	206	813
	2	2	30.4	9.2	5.2	85.7	97.2	204	826
	4	2	22.5	7.6	1.0	85.7	97.2	241	910
	8	2	12.9	5.7	0.2	85.7	97.2	272	934
s1196	1	2	27.5	45.2	4.7	99.8	100	360	1243
	2	2	14.3	30.5	3.4	99.8	100	364	1243
	4	2	7.54	18.5	2.6	99.8	100	367	1243
	8	2	4.6	12.8	3.4	99.8	100	404	1243
s1238	1	2	56.2	56.2	5.8	94.7	99.7	375	1356
	2	2	31.7	39.6	0.7	94.6	99.6	392	1356
	4	2	17.1	22.7	1.6	94.5	99.6	377	1356
	8	2	9.2	14.8	1.4	94.6	99.6	414	1356
s5378	1	2	2879.8	913.0	771.7	68.8	70.3	985	13339
	2	2	1681.4	533.4	506.2	70.6	72.2	1057	13875
	4	2	884.1	283.4	256.9	70.5	72.0	998	14346
	8	2	479.9	217.1	143.5	68.8	70.2	1395	16030

Table XI: Performance of the ProperSYN combinatorial logic synthesis algorithm on MCNC benchmark circuits on a network of SUN workstations.

CKT	1 Processor		2 Processor		4 Processor	
	Run Time	Speedup	Run Time	Speedup	Run Time	Speedup
5xp1	102.99	1.0	55.49	1.86	31.97	3.22
b9	121.25	1.0	69.19	1.75	38.51	3.15
bw	468.19	1.0	264.80	1.76	172.58	2.71
f51m	180.97	1.0	118.07	1.53	72.24	2.51
misex1	41.68	1.0	22.66	1.84	13.92	2.99
misex2	109.00	1.0	60.83	1.79	35.85	3.04
rd73	387.74	1.0	251.32	1.54	129.42	3.00
rd84	4792.74	1.0	2476.32	1.94	1279.05	3.75
sao2	525.47	1.0	284.74	1.84	172.97	3.04
vg2	735.72	1.0	400.70	1.84	225.16	3.27
apex7	2272.41	1.0	1311.86	1.73	595.02	3.82

machines.

5 Summary

We have developed an environment for the portable object-oriented parallel execution of CAD algorithms. The main objectives of this research have been to make automatic porting of parallel software feasible and practical, and exploit the current and future advances in sequential CAD algorithms. As mentioned in the introduction, since inception, the ProperCAD project (see Figure 1) is designed to be completed in two phases. In the first phase, we are designing portable parallel algorithms for a large set of CAD applications using CHARM. The second phase of the project will involve the design and implementation of a run-time support system for portable parallel programming in C++. This system, although inspired by CHARM, will be tailored specifically for CAD applications. This will make the programming environment truly object-oriented and will support features like *inheritance* and *classes*. The ProperCAD applications will then be rewritten and ported onto the new C++ platform. It should be noted that the parallel algorithms in the ProperCAD project are being designed *around* existing sequential algorithms and extensively reuses existing sequential code.

We have demonstrated the feasiblity of this approach through several applications, namely, flat circuit extraction, test generation for sequential circuits, combinational logic synthesis and stan-

Table XII: Performance of the ProperSYN combinatorial logic synthesis algorithm on MCNC benchmark circuits on the Intel i860 hypercube.

CKT	1 Processor		2 Processor		4 Processor		8 Processor	
	Run Time	Speedup	Run Time	Speedup	Run Time	Speedup	Run Time	Speedup
5xp1	60.24	1.0	32.67	1.84	18.61	3.24	12.80	4.70
b9	70.23	1.0	33.12	2.12	21.90	3.20	15.42	4.55
bw	217.54	1.0	122.54	1.77	77.62	2.80	42.10	5.17
f51m	89.92	1.0	57.55	1.56	37.22	2.42	21.11	4.26
misex1	21.43	1.0	18.41	1.17	13.45	1.60	11.73	1.80
misex2	64.68	1.0	37.86	1.71	22.83	2.83	11.55	5.60
rd73	233.63	1.0	107.39	2.17	66.15	3.53	29.13	8.02
rd84	2381.77	1.0	1190.08	2.00	563.25	4.22	308.66	7.72
sao2	356.66	1.0	197.76	1.80	108.92	3.27	50.04	7.13
vg2	390.08	1.0	224.59	1.74	112.09	3.48	53.00	7.36
apex7	1478.43	1.0	884.24	1.67	461.96	3.20	222.94	6.63
apex6	15418.68	1.0	7700.81	2.00	3555.27	4.34	1842.22	8.37
duke2	12190.12	1.0	6371.77	1.91	3248.30	3.75	1686.32	7.23
misex3c	15257.09	1.0	7842.69	1.95	4074.12	3.75	1907.11	7.98

Table XIII: Performance of the ProperSYN combinatorial logic synthesis algorithm on MCNC benchmark circuits on the Encore Multimax.

CKT	1 Processor		2 Processor		4 Processor		8 Processor	
	Run Time	Speedup	Run Time	Speedup	Run Time	Speedup	Run Time	Speedup
5xp1	177.62	1.0	122.52	1.45	57.20	3.10	35.30	5.02
b9	231.24	1.0	123.35	1.87	57.82	3.99	40.21	5.75
bw	755.39	1.0	409.75	1.84	198.64	3.80	108.21	6.98
f51m	296.07	1.0	184.32	1.60	101.76	2.91	69.84	4.24
misex1	70.56	1.0	40.27	1.75	22.98	3.07	11.18	6.31
misex2	212.96	1.0	108.11	1.97	60.70	3.51	36.15	5.90
rd73	769.28	1.0	410.88	1.87	207.46	3.54	110.92	6.93
rd84	9159.49	1.0	3998.18	2.29	2390.73	3.83	1378.20	6.64
sao2	1174.38	1.0	659.78	1.77	315.15	3.73	152.03	7.72
vg2	1373.67	1.0	822.33	1.67	392.75	3.50	188.59	7.28
apex7	4868.01	1.0	2383.75	2.04	1341.27	3.63	727.92	6.69
apex6	54062.65	1.0	26976.26	2.00	14271.42	3.79	7241.64	7.47
duke2	40138.20	1.0	25390.33	1.58	12142.03	3.30	-	-
misex3c	50236.75	1.0	26412.00	1.90	13721.43	3.66	7011.62	7.16

Table XIV: Performance of the ProperPLACE algorithm for standard cell placement on a network of Sun workstations.

	1 PE		2 PEs		4 PEs		8 PEs	
Circuits	Wire Length	Time (sec.)	Wire Length	Time (sec.)	Wire Length	Time (sec.)	Wire Length	Time (sec.)
s298	32120	780	32274	458	32395	282	32938	194
s420	38451	814	38480	525	38905	270	39032	201
fract	22067	640	22708	426	22592	213	23050	152
primary	372561	914	373034	605	381830	351	390743	241
primary								

Table XV: Performance of the ProperPLACE algorithm for standard cell placement on the Encore Multimax.

	1 PE		2 PEs		4 PEs		8 PEs	
Circuits	Wire Length	Time (sec.)	Wire Length	Time (sec.)	Wire Length	Time (sec.)	Wire Length	Time (sec.)
s298	32052	1538	32603	899	32842	480	33106	317
s420	38627	1678	39083	969	39130	559	39852	373
fract	21575	1419	21692	857	21854	489	22540	368
primary	375870	2054	376991	1194	380492	760	386403	503
primary								

dard cell placement. New algorithms for global routing, fault simulation and behavioral simulation are currently under development. All the applications exhibit good speedups on shared memory machines including an Encore Multimax and a Sequent Symmetry, message passing machines including an NCUBE 2, an Intel i860 hypercube and a network of Sun workstations. This is significant especially given that the applications were all executed *unchanged* on all the above machines.

When the ProperCAD environment is available on a new architecture, say the Intel Paragon multiprocessor, these algorithms will *not* need to be rewritten, unlike most prior algorithms. It is only necessary to port the underlying programming platform (which itself is largely portable with the exception of a small machine specific component).

All these applications are being developed on a parallel object-oriented platform, using a coarse-grained data-flow style of execution. In all cases, the algorithms are being interfaced with uniprocessor implementations of the respective applications. In circuit extraction, for example, sequential modules were used to perform local geometric extraction and device and parameter extraction.

Table XVI: Performance of the ProperPLACE algorithm for standard cell placement on the Intel i860 hypercube.

	1 PE		2 PEs		4 PEs		8 PEs	
Circuits	Wire Length	Time (sec.)	Wire Length	Time (sec.)	Wire Length	Time (sec.)	Wire Length	Time (sec.)
s298	32512	191	32603	120	32969	68	33106	47
s420	38066	288	37960	178	38943	103	39836	73
fract	22717	534	22904	322	23010	190	23109	137
primary	373905	769	374042	447	381839	307	387592	198
primary								

The parallel algorithm was primarily concerned with the decomposition of the circuit into regions that could be processed in parallel, and the merging of these regions together. In cell placement, we have interfaced the parallel algorithm with TimberWolf 6.0, a state-of-the-art widely used cell placement program.

We believe that this multilevel separation of a parallel run-time system, a parallel library, a parallel algorithm and a sequential algorithm with well-defined interfaces between them, as outlined in Figure 1, is the most efficient way to develop parallel CAD algorithms. This permits the experts in each of these different areas to concentrate on their fortes. An environment such a ProperCAD is best written by an expert in parallel programming who has intimate knowledge about the target machines. The parallel algorithms can then be developed with the constraint that the algorithms are expressed using the ProperCAD environment. Finally, experts in the area of circuit extraction, test generation, logic synthesis, cell placement, etc. should be designated the responsibility ofr developing efficient sequential algorithms for their respective problems. We constrain them to express their algorithms in a modular fashion: a desirable requirement for program design and maintenance in any case. The ProperCAD environment serves to bridge the effort in these various different areas of specialization.

Work is also under way to expand the set of target architectures for the ProperCAD environment. We are awaiting access to parallel machines like the Intel Paragon and the Thinking Machines CM-5 to initiate the port to these machines.

[1] Agha, G.A. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT press, 1986.

[2] Banerjee P., Jones M.H., Sargent J. Parallel Simulated Annealing Algorithms for Standard Cell Placement on Hypercube Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1:91–106, January 1990.

[3] Belkhale, K.P, Banerjee, P. PACE: A Parallel VLSI Circuit Extractor on the Intel Hypercube Multiprocessor. In *Proceedings of the International Conference on Computer Aided Design*, November 1988.

[4] Belkhale, K.P, Banerjee, P. PACE2: An Improved Parallel VLSI Extractor with Parameter Extraction. In *Proceedings of the International Conference on Computer Aided Design*, pages 526–530, November 1989.

[5] De, K., Ramkumar, B., Banerjee P. ProperSYN: A Portable Parallel Algorithm for Logic Synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, November 1992.

[6] Fenton, W., Ramkumar, B., Saletore, V.A., Sinha A.B., Kalé, L.V. Supporting Machine Independent Programming on Diverse Parallel Architectures. In *International Conference on Parallel Processing*, August 1991.

[7] Fitzpatrick, D.T. MEXTRA: A Manhattan Circuit Extractor. Technical Report Electronics Research Lab M82/42, University of California at Berkeley, January 1982.

[8] Gupta, A. ACE: A Circuit Extractor. In *Design Automation Conference*, pages 721–725, June 1983.

[9] Harrison, D.S., Moore, P., Spickelmier, R.L., Newton, A.R. Data Management and Graphics Editing in the Berkeley Design Environment. In *Proceedings of the International Conference on Computer-Aided Design*, November 1986.

[10] Hon, R., Gupta, A. *HEXT: A Hierarchical Circuit Extractor*. Computer Science Press, 1983.

[11] Jayaraman, R., Rutenbar, R.A. Floorplanning by Annealing on a Hypercube Multiprocessor. In *Proceedings of the International Conference on Computer Aided Design*, pages 346–349, November 1987.

[12] Kalé, L.V. The Chare Kernel Parallel Programming System. In *International Conference on Parallel Processing*, August 1990.

[13] Kravitz, S.A., Rutenbar, R.A. Multiprocessor-Based Placement by Simulated Annealing. In *Proceedings of the 23rd Design Automation Conference*, pages 567–573, June 1986.

[14] Levitin S. MACE: A Multiprocessor Approach to Circuit Extraction. Master's thesis, MIT, Cambridge, Mass., June 1986.

[15] McCormick, S.P. EXCL: A Circuit Extractor of IC Designs. In *Design Automation Conference*, pages 624–628, June 1984.

[16] Niermann, T.M., Patel, J.H. HITEC: A Test Generation Package for Sequential Circuits. In *Proceedings of the European Conference on Design Automation*, pages 214–218, February 1991.

[17] Patil, S., Banerjee P., Patel, J.H. Parallel Test Generation for Sequential Circuits on General Purpose Multiprocessors. In *Proceedings of the 28th Design Automation Conference*, June 1991.

[18] Ramkumar, B., Banerjee P. Portable Parallel Test Generation for Sequential Circuits. In *Proceedings of the International Conference on Computer-Aided Design*, November 1992.

[19] Ravikumar, C.P., Sastry S. Parallel Placement on Hypercube Architectures. In *International Conference on Parallel Processing*, pages III: 97–100, August 1989.

[20] Rose, J.S., Snelgrove W.M., Vranesic, Z.G. Parallel Cell Placement Algorithms with Quality Equivalent to Simulated Annealing. *IEEE Transactions on Computer-Aided Design*, 7, no. 3:387–396, March 1988.

[21] Scott, W.S., Ousterhout, J.K. Magic's Circuit Extractor. In *IEEE Design and Test*, pages 24–34, February 1986.

[22] Sechen, C., Sangiovanni-Vincentelli A.L. The TimberWolf Placement and Routing Package. *IEEE Journal of Solid-State Circuits*, 23/2:410, 1988.

[23] Su, S-L., Rao, V.B., Trick, T.N. HPEX: A Hierarchical Parasitic Circuit Extractor. In *Design Automation Conference*, pages 566–569, June 1987.

[24] Suaris P., Kedem, G. A Quadrisection-Based Combined Place and Route Scheme for Standard Cells. *IEEE Transactions on Circuits and Systems*, 8:234–244, March 1989.

[25] Tonkin, B.A. Circuit Extraction on a Message Passing Multiprocessor. In *Design Automation Conference*, pages 260–265, June 1990.

[26] Wegner, P. Conceptual Evolution of Object-Oriented Programming. In *Object-Oriented Programming Systems, Languages and Applications*, 1989. keynote talk.