Tradeoffs in Message Passing and Shared Memory Implementations of a Standard Cell Router

Margaret Martonosi and **Anoop Gupta**
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

This paper considers two implementations of the LocusRoute standard cell routing program. One implementation uses a message passing approach, where global data distributed among the processes is kept consistent through explicit updates using messages. The other implementation uses a shared memory approach and relies on underlying coherence mechanisms, such as hardware cache coherence protocols, to keep the data consistent. We discuss the performance of these two mappings in terms of the network traffic, execution time, and solution quality. We explore a number of update strategies for the data structures in the message passing implementation, classifying the methods according to whether they are *sender initiated* or *receiver initiated* and whether they are *blocking* or *non-blocking*. Further, we show that these explicit methods for interprocessor updates can reduce network traffic to as little as 1% of the traffic required for our shared memory approach. Also, we examine methods for task assignment which take advantage of locality. While these methods can improve solution quality and reduce the need for interprocessor communication in either paradigm, task assignment based strictly on locality can lead to load imbalances between processors. Finally, we examine the effect of increasing the number of processors on solution quality, execution time and network traffic.

1 Introduction

Two common parallel programming paradigms are the message passing programming model and the shared memory programming model. In the message passing style, data is distributed among the processes; if there are global data structures, they are kept up to date by sending messages. In such an approach, the programmer is responsible for maintaining the consistency of the data structures. In the shared memory model, the data structures are stored in the shared memory, and the consistency of the shared memory is guaranteed by an underlying coherence mechanism, such as a hardware cache coherence protocol.

In this paper, we explain the decisions made in encoding the LocusRoute [8, 9] standard cell routing program in a message passing style, and make comparisons to the original shared memory implementation. We compare the performance of these mappings in terms of network traffic, execution time, and solution quality. In the message passing version, the strategy used to keep the global data up to date is an important part of the implementation. Updates can be either *sender initiated* or *receiver initiated*. Receiver initiated updates are further broken down into *blocking* and *non-blocking* types, where a blocking update means that the processor that requests an update is held idle until the update arrives. Also, each update can contain either absolute data, or data relative to the last update that was sent. Results presented show that, for LocusRoute, sender initiated schemes produce quality which is slightly better on average than receiver initiated schemes, but with increased execution time compared to the receiver initiated approach, and network traffic that is nearly an order of magnitude larger than that for receiver initiated strategies. Most of the message passing update schedules we used produce quality within 10% of the original shared memory version, with network traffic reduced significantly compared to the traffic required by our shared memory approach.

We also examined different methods for task assignment in each paradigm. Both quality and traffic can be improved (by as much as 5% and 63% respectively) by making use of locality when assigning tasks. In this study, we exploit locality by assigning each processor wires which are physically close to each other in the circuit. However, strict application of this heuristic can lead to load imbalances which degrade execution time, if the circuit's wires are not spread evenly over the area of the circuit. Also, we examine the dependence of performance on the number of processors. In either paradigm, the quality of the solution degrades as the number of processors increases, because more work is being performed in parallel, and the processors do not know about the work other processors are doing simultaneously.

Because the message passing paradigm can be efficiently supported on a shared memory machine, the results presented here do not necessarily lead to architectural conclusions, but rather reflect more on the effect that programmer effort can have in efficient management of data structures, and on efficient parallel programming techniques in general. Although the results presented in the paper are specific to LocusRoute, the general techniques discussed are applicable to a variety of optimization problems.

The rest of the paper has the following structure. Section 2 gives a brief description of the tools and method-

In *Proc. 1989 International Conference on Parallel Processing*

ology used to collect the data presented here. Section 3 gives a general description of the LocusRoute application, and its shared memory implementation. Section 4 explains the design decisions made in mapping the application to a message passing paradigm. Results on the performance of LocusRoute with these decisions are presented in Section 5, and finally, conclusions are presented in Section 6.

2 Tools and Methodology

This section describes the tools and benchmark circuits used in collecting the data to be presented in this paper.

2.1 CBS Message Passing Architecture Simulator

To simulate execution of the message passing version of LocusRoute, we used the CBS [7] message passing architecture simulator. This simulator generates useful statistics on network traffic and execution time. CBS simulates a k-ary n-dimensional hypercube machine (with a total of k^n processors). For the work described here, CBS simulated a machine with deterministic wormhole routing [3], and with a two–dimensional mesh interconnection [4]. CBS also models contention on the network. The number of processors was varied by changing the arity of the simulated machine. There are unidirectional channels connecting each processor to two of its four neighbors. With no contention and with one byte wide channels, the total time required for a packet of L bytes to travel D hops on the network is:

2ProcessTime + HopTime(D + L).

ProcessTime is the time for the entire message to be copied from the processor node to the message network, and HopTime is the time required for one byte to travel one hop on the network. HopTime and ProcessTime were set to 100 ns and 2000 ns, respectively, to roughly model the performance of the Ametek Series 2010 message passing computer [1, 10]. The simulations were run on an Encore Multimax computer [5].[1]

2.2 Tango Shared Memory Tracing System

Traffic data presented for the shared memory version of LocusRoute was calculated by analyzing shared data reference traces collected from multiprocess runs of LocusRoute using the Tango tracing system [6]. These traces contain all shared data references made by the program during execution. For each reference, the time, address, and referencing processor are recorded. The traces are generated on a uniprocessor by spawning the specified number of processes and multiplexing their execution. This multiplexing is very fine grained and is controlled to closely model a run on a multiprocessor.

[1] To simulate the Ametek's MC68020 processing nodes, all times from the Encore Multimax clock were divided by five, because the Multimax uses NS32032 microprocessors which are about five times less powerful.

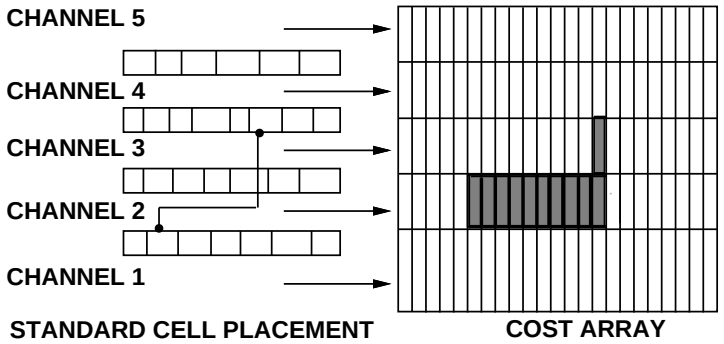

Figure 1: Standard cell placement and corresponding cost array.

2.3 Benchmark Circuits

The data presented here was collected for two benchmark circuits. The first circuit, bnrE, has 420 wires, a size of 10 channels by 341 routing grids, and represents an actual standard cell circuit developed at Bell-Northern Research Ltd. The second circuit, MDC, has 573 wires with a size of 12 channels by 386 routing grids, and was designed at the University of Toronto Microelectronic Development Centre.

3 LocusRoute: The Application

This section will give background information on the LocusRoute application, in preparation for later sections which describe the tradeoffs involved with mapping LocusRoute to a message passing approach.

LocusRoute [8, 9], originally written in a shared memory style, is a commercial quality VLSI standard cell router developed by Jonathan Rose at Stanford University. LocusRoute routes the wires of a standard cell circuit, attempting to minimize the circuit area. LocusRoute's central data structure is a `cost array` that keeps a record of the number of wires running though each routing grid of the circuit. The vertical dimension of the array is the number of routing channels in the circuit, and the horizontal dimension is the number of routing grids. Each wire is routed along the path with the minimal sum of the cost array entries. Figure 1 shows a standard cell circuit and one of its wires, with the corresponding `cost array`. The highlighted portions of the cost array will be incremented if this route is chosen. An iteration of routing is completed when each wire has been routed once. Performing several of these iterations, with all wires routed once per iteration, improves the final solution quality [8]. However, before rerouting a wire in a later iteration, the processor must "rip up" the previous routing of the wire by decrementing the cost array locations in its path.

In addition to producing the routed circuit, LocusRoute

also computes a measure of the overall solution quality. Overall quality, also referred to as *circuit height*, is computed as follows. For each channel, the number of wires using the channel will vary across the width of the circuit. The number of routing tracks required by the channel is the maximum number of wires running through the channel at any point. The circuit height is the total number of routing tracks required for all channels. This measure of quality is useful because it is proportional to the circuit area, and thus, gives a means of estimating the expected area of the circuit. A second measure of quality will be presented in some cases as well. The second measure of quality is called the *occupancy factor*. The occupancy factor is computed as follows. When a wire is routed, the cost of a path is the sum of the cost array entries along it. The occupancy factor is the sum, over all wires, of the path costs at the time the wire is routed. The occupancy factor gives an indication of the cost of the wire's path at the time it was chosen. For both of these measures, a lower value means better quality.

The shared memory implementation follows fairly naturally from the algorithm description stated above. The `cost array` is implemented as an array in global shared memory. All the processors perform reads and writes on it as they route the wires. Although it is possible for several processes to simultaneously attempt to read or write the same array location, the probability of such collisions is low, and for this reason, accesses to the cost array are not locked. Research has shown that the solution qualities obtained from the shared memory versions with and without cost array locking are not significantly different, and the performance of the non–locking version is better due to the decrease in serialization [9].

In this shared memory implementation, wire distribution can be easily accomplished using a distributed loop, in which processes are repeatedly given wires to route. When done with one wire, processes request another wire subscript. When all the wires have been given out, processes are blocked at a barrier until all the processors are finished. While this original shared memory approach has simplicity on its side, it does not attempt to take advantage of locality by assigning wires to processes such that each process works mainly in one region of the circuit and cost array. We will show that a wire assignment strategy which does take advantage of locality in this way can result in better solution quality and lower network traffic, because interference among processes as they access the cost array is greatly reduced.

4 LocusRoute in a Message Passing Style

This section describes the tradeoffs involved in implementing LocusRoute in a message passing style. LocusRoute is an optimization problem which can tolerate out of date information in its main data structure, the `cost array`, so the programmer has considerable flexibility in implementing it in a message passing style. In essence, applications like LocusRoute allow the programmer to choose

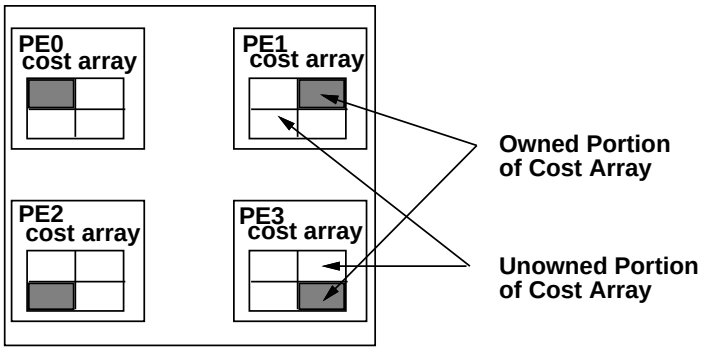

Figure 2: Division of the cost array among processors.

to simulate shared memory only up to the degree of consistency required by the algorithm for the desired solution quality. Many message passing implementations of LocusRoute are possible, depending on (i) how the programmer decides to distribute the cost array, (ii) how wires are assigned to processors to be routed, (iii) what types of updates are performed on the cost array, and (iv) how frequently updates are performed. The following subsections elaborate on these decisions, discussing alternatives for each.

4.1 Distribution of Cost Array

In the shared memory version of LocusRoute, all processors have access to a single, consistent copy of the cost array. When encoding the application in a message passing style, one must decide how the cost array should be implemented, since no shared memory is available. The cost array could be divided into portions of approximately equal size, allocating a portion to each processor. Each processor performs all routing within the region allocated to it. If routing extends into another region, the task is passed to the processor owning that region. This implementation has load balancing problems if many wires lie in a single processor's region of the cost array. Also, since most wires span multiple regions, and since many routes are explored for each wire, this method could generate a large amount of message traffic.

Instead, we chose a less strict approach. The cost array is divided into sections, and each processor is the owner of one section. However, each processor has a view of the whole cost array. The processor which owns a certain region of the cost array is the *owner processor* for the region, and the region itself is the *owned region*. The four processor example in Figure 2, shows each processor's cost array with the owned regions highlighted. Also, we add a new data structure, known as the `delta array`. The delta array has the same dimensions as the cost array, and keeps track of changes made to the cost array between updates. This delta array is used to notify other processors of changes that have been made, and will be further discussed in Section 4.3.

4.2 Wire Assignment and Locality

In the shared memory paradigm, wires can either be assigned dynamically, using a distributed loop for example, or statically, using locality and load balancing heuristics. In the message passing approach, dynamic wire allocation requires message transactions on the network. Also, the time spent waiting for a requested task can be large. If the processor receiving the task requests is also routing wires, and if processors only check for newly received messages between routing wires, a processor may have to wait for an entire wire to be routed before the wire assignment processor even retrieves the task request message from its queue.

A variation on the first scheme is possible if messages arriving at a processor can cause the processor to be interrupted. The wire assignment processor routes wires at a low priority level, and responds to wire request interrupts from other processors at a higher priority level. This solves the problem of the first method, where wire requests were only processed between wires. If the wire request interrupt can be serviced with low overhead, this method can offer wire distribution with lower latency than the first method. However, because this method is dynamic, it is difficult to examine how locality in the wire assignment affects performance. Also, CBS does not support the notion of interrupts occurring on message reception, so this wire assignment method is difficult to simulate.

For the above reasons, we chose a static wire assignment method. Because the cost array is divided into owned regions, the algorithm benefits from a wire assignment method that attempts to assign wires to the owner processor of the region they run through. To achieve this, wires are assigned to the owner processor of the leftmost pin of the wire. Unfortunately, trials with this heuristic had very poor load balancing, because no effort was made to even out the number of wires each processor had. We modified the assignment strategy to avoid this. A cost measure is computed for each wire, based on its length. Any wire with cost less than the parameter `ThresholdCost` is assigned to the owner processor of the wire's leftmost pin. All longer wires, which have cost greater than ThresholdCost and which have limited locality anyway, are held until a final step in the static wire assignment phase, where they are assigned to balance the load, ignoring locality. By increasing ThresholdCost, we increase the amount of locality exploited in the wire assignment. By decreasing ThresholdCost, more wires are assigned on the basis of load balancing instead of locality.

4.3 Update Mechanisms

In most circuits, wires assigned to one processor will extend into regions owned by other processors, so updates of cost array information between processors will have to occur. We explore the efficiency of different methods of updates. Figure 3 shows a classification of the types of updates that will be discussed in this section.

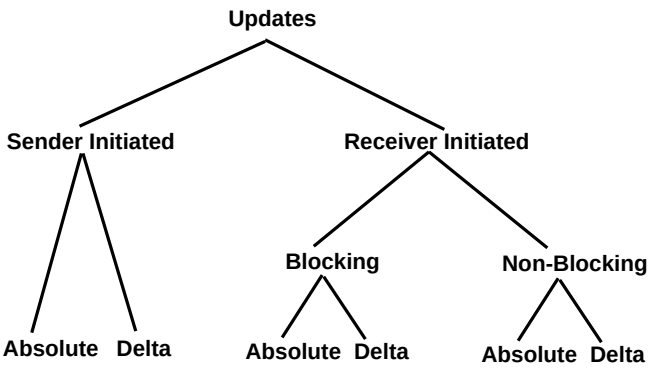

Figure 3: Classification of types of updates.

4.3.1 Structure of Updates

The structure of the update packets should satisfy several requirements. First, it should maximize the useful information per byte. However, it must also be convenient to assemble and disassemble, so the processing overhead of sending or receiving a message is not high.

One possible packet structure is based directly on the structure of the wires being routed. This packet structure would contain coordinates of the start and end points of each horizontal or vertical segment of the wire, along with a flag indicating whether this wire had been ripped up (decrement cost array) or routed (increment cost array).

Another possible structure for the packet is to have the update packet contain the values of an entire region of the cost array owned by one of the processors. This is simple for the sender and receiver to process, and may be more efficient than the wire based packet structure if many wires are changed between updates. On the other hand, it uses a large number of bytes, which can increase the processing overhead at both the sending and receiving ends and congest the network.

We chose a third update strategy, which is a simple optimization of the second one. In this structure, the sending processor scans the delta array for changes, shown by non-zero elements in the delta array. For each cost array region, the sender constructs a packet which contains the bounding box of all the changes made within that region, as well as the coordinates of the bounding box being sent. Because the sender has to scan the array for changes, this method has more overhead at the sending end than the second method described. However, it reduces network traffic compared to the other method, because usually only a small fraction of the owned region has been changed. It also reduces message reception overhead at the receiver, because there are fewer bytes for the receiver to incorporate into its cost array.

4.3.2 Sender Initiated Updates

We must also decide how updates should be initiated. We propose a general classification into four types of update transactions. The first two types of transactions are called

sender initiated because the processor to determine that an update should be initiated is the one which sends the data. The other two are *receiver initiated* and are discussed in section 4.3.3.

SendLocData is used to inform other processors of the owner processor's view of its owned region. The receiving processors replace their view of the region being updated with the update data. In general, this type of update can be sent to any processor, but as an optimization in LocusRoute, these packets are sent only to the North, South, East, and West neighbors of the owner processor sending the update.

SendRmtData is used to inform an owner processor of changes that another processor has made to the owner processor's owned region. The processor sending this update is not the owner processor of the region, so it does not send the absolute cost array entries. Rather, it sends the corresponding locations from the delta array, indicating the changes made to this region since the last update.

To study the sensitivity of the LocusRoute algorithm to inconsistencies in the cost array, the frequency with which updates are performed was allowed to vary. For each type of sender initiated update, there is a parameter indicating how many wires should be routed between updates. In addition, if an update is supposed to occur, but no changes have been made in the region to be updated, the update will not be sent out. The size of the update varies, but it is always the rectangular bounding box of all the changes that have occurred in a given owned region.

4.3.3 Receiver Initiated Updates

In the receiver initiated schemes, a processor determines (using methods to be discussed later) that an update should occur. It sends either a *ReqLocData* or a *ReqRmtData* packet to the appropriate processor. Request packets include coordinates of the bounding box of the region for which an update is requested. When the destination processor receives the request, it returns a response packet with the requested data, so in this case, the initiator of the transaction is the receiver of the data.

ReqRmtData is used when a processor wants to update its view of a remotely owned region of the cost array. For each wire, a processor determines which regions contain the wire, and increments a counter for each of those regions. When the count for a region reaches a specified (variable) parameter, a ReqRmtData packet is sent to the owner processor of that region. The wire assignment is static so a processor knows in advance which wires will be routed and can request updates for the appropriate regions ahead of time. Requesting updates in advance helps ensure that the update will arrive before routing for that wire actually begins, but updates ordered too far in advance will be out of date before they are used. As a compromise between these two opposing factors, we chose to have processors request updates for five wires at a time.

ReqLocData is useful when an owner processor wishes to bring its owned region into a consistent state. The owner processor monitors the number of ReqRmtData packets it receives from all the other processors. If a remote processor has made more than a certain number of ReqRmtData requests, the owner processor requests an update from the remote processor, because the remote processor has been routing in the owner processor's region.

In the case of receiver initiated updates, one further decision remains. If a processor has requested an update, it can either block, and wait until the requested update arrives, or it can proceed with its work. If it proceeds, the processor never needs to sit idle, waiting for the packet. On the other hand, if it does not wait for the packet, the processor may actually finish routing the wire for which the update was requested, before the update even arrives. Due to locality, the processor is likely to be routing wires in that region again soon, and an update that arrives too late for one wire can often be useful for other ones. Also, because several iterations of routing are performed, the processor is guaranteed (except on the last iteration) to route the same wire again, so the update, if not yet out of date, will be useful for that. In fact, results presented in Section 5.1.3 show that the quality using the non-blocking scheme is not worse than when using the blocking scheme, and the execution time is much improved.

5 Results

The previous sections introduced the implementations of LocusRoute in shared memory and message passing styles. Here, we present results to support the decisions made in those approaches. Unless otherwise noted, the message passing results are presented for 16 processors arranged in a four by four grid. The shared memory results are for 16 processors using a Write Back with Invalidate cache coherence protocol [2].

5.1 Effectiveness of Message Passing Update Strategies

The goal of the update strategy is to produce a good quality result with the smallest amount of communication possible, and with a minimal amount of computational overhead. The following subsections will present data showing the effect of the update strategies outlined in Section 4.3 on solution quality, execution time, and network traffic.

5.1.1 Sender Initiated Strategy

Table 1 shows results of running message passing LocusRoute on the bnrE circuit using a purely sender initiated update scheme. The table shows the frequency of both types of sender initiated updates, given in terms of the number of wires between updates, and the circuit height resulting with each update schedule. The quality of the message passing version using sender initiated updates is within 10% of the shared memory quality for most trials. Unfortunately, quality measured in terms of circuit height seems to have little correlation with the update frequency. This is because the

Table 1: Network traffic using sender initiated updates.

SendRmtData,	SendLocData	Ckt Ht.	Occup. Factor	MBytes Xfrd.	Time (s)
2	1	142	426109	.862	1.893
	5	143	428558	.222	1.515
	10	141	429589	.140	1.445
	20	145	432360	.101	1.426
5	1	144	425576	.859	1.668
	5	143	430046	.212	1.306
	10	146	430580	.133	1.260
	20	145	431366	.094	1.240
10	1	142	426706	.840	1.553
	5	143	429423	.208	1.282
	10	146	431662	.128	1.243
	20	145	432169	.087	1.219

circuit height only changes when routes which affect the number of routing tracks per channel are changed. Further, the table indicates that the occupancy factor is only slightly sensitive to changes in SendLocData with SendRmtData held constant. In bnrE, with SendRmtData equal to 2 wires, the occupancy factor changes only 2%.

The execution time varies a great deal with the frequency of updates, from a maximum value of 1.893 seconds to minimum value of 1.219 seconds. Because the same static wire assignment is used for all the runs shown here, the execution time is clearly a function of the frequency of updates. Timing the assembly and disassembly of packets shows that these operations take up to one fourth of the processing time in runs with frequent updates. To compare the execution time with that of the shared memory version, recall that CBS is simulating processors which are five times faster than the NS32032 processors of the Encore Multimax. Therefore, a rough comparison can be obtained by multiplying the execution times of the message passing version by five.[2] The best time for bnrE is 1.219 seconds, which when multiplied by five, is comparable to the shared memory version.

The number of bytes transferred is also a clear function of the update frequency. However, the increase in network traffic is less than linear with the update frequency. This is due to the form of the updates: a bounding box of all changes made to a region. When updates are performed after many wires, changes may have been made in several different areas of the cost array region, and the bounding box will contain all the unchanged locations between the changed areas. When updates occur more frequently, the updates more closely match the changes that occurred, and fewer extra bytes are sent. This leads to the sublinear increase in network traffic with update frequency.

[2] Note that simple multiplication by a factor of five when comparing the execution times favors the bus-based shared memory architecture. This is because if the processors in the shared memory machine really were five times faster, there would be more contention on the bus, and the overall performance would not improve by a factor of five.

5.1.2 Non-Blocking Receiver Initiated Strategy

Table 2 shows the quality, execution time, and network traffic for several purely receiver initiated strategies. All

Table 2: Traffic using non-blocking receiver initiated updates.

ReqLocData,	ReqRmtData	Ckt Ht.	Occup. Factor	MBytes Xfrd.	Time (s)
1	5	144	430686	.130	1.166
	10	150	436496	.056	1.159
	30	151	437956	.009	1.099
2	5	143	431936	.112	1.156
	10	149	437088	.045	1.126
	30	151	437956	.009	1.113
10	5	142	430868	.088	1.133
	10	149	437797	.039	1.135
	30	151	437956	.009	1.097

of these strategies use the non-blocking receiver method, in which processors do not wait for the responses to their requests.

The best circuit height measure for the non-blocking receiver initiated case is slightly worse than the best circuit height measure for the sender initiated case. The small overall degradation in output quality is due to the loose coupling between update requests and wire routing. One can see that the circuit height is quite sensitive to changes in ReqRmtData. When ReqRmtData is increased beyond 5, the quality of the routing drops by about 5%.

The execution time is improved over that for the sender initiated strategies. The best execution time is 1.097 seconds, as compared to 1.219 seconds for the sender initiated version. The large variation in execution time with update frequency that was present in the sender initiated trials is not present here, however. The reason for this is that the processors in the sender initiated approach are spending a large fraction of time processing messages. There is less traffic in the receiver initiated approach, so the message processing overhead is lower, leading to both less execution time overall, and less dependence of the execution time on the frequency of updates.

5.1.3 Other Strategies

Intuitively, one expects a blocking receiver initiated scheme to give the best quality. First, any processor can determine if it needs an update and order one. Second, because the processor is forced to block until the ordered updates arrive, the processor is guaranteed to make the routing decisions using information that is only as out of date as the network delay that was required to send it to the requester. Actually though, the average quality of the blocking receiver initiated runs is about the same as that for the non-blocking

runs. Further, blocking strategies have execution times as much as 75% larger than non-blocking schemes using the same update schedule. With a higher performance interconnection network, lower overhead on message reception, and a better heuristic for requesting updates, the blocking strategy would probably become more effective than the non-blocking strategy.

We also experimented with several update schedules which were mixtures of sender and receiver initiated updates. When the occupancy factor is used as the measure of quality, mixed update schemes generally give an improvement over sender or receiver initiated schemes alone. For example, a mixed scheme with parameters SendLocData = 5, SendRmtData = 2, ReqLocData = 1 and ReqRmtData = 5 gives an occupancy factor of 424337. This quality is better than that from the first line of Table 1, 430686, and is obtained using only .311 MBytes, less than half the network traffic of the sender initiated scheme. However, when a comparison of quality is made using the circuit height metric, the sender initiated approach always produces results of similar or better quality.

5.1.4 Summary of Update Strategy Results

The sender initiated update strategy gives the best results in terms of circuit height. However, the network traffic can be more than ten times larger for a sender initiated scheme than for a receiver initiated scheme. In situations where quality is required at any cost, the sender initiated approach will be the preferred update method for the message passing implementation.

5.2 Comparison of Shared Memory and Message Passing Approaches

The previous section gave the performance of the various message passing update strategies in terms of network traffic, solution quality and execution time. In this section, we will compare those results with data from a shared memory approach.

The metric of network traffic is an interesting basis for comparison because it reflects the amount of interprocessor communication required by each approach to achieve similar results. In a message passing architecture, there is processing overhead associated with sending and receiving update messages, so one would like to update as infrequently as possible. In a shared memory architecture, hardware cache consistency protocols cause extra bus traffic due to cache line invalidations. These operations cause the processor to stall, and so, also hurt performance. Traffic in the shared memory approach is made up of three parts. First, the processor's initial access to a location always results in a miss, and brings the line into the cache. Second, the first write to a clean location causes a word write on the shared bus. The other processors see this write and invalidate that cache line if it is in their cache. Third, once a line has been invalidated by a cache, it may need the line again. This leads to refetches of the data from memory. Clearly, traffic in the shared memory approach is a function of the cache coherence protocol and the line size of the cache.[3] For all the results given here, we used a Write Back with Invalidate coherence protocol [2].

Increases in the cache line size can have the effect of either increasing or decreasing traffic. First, with a longer cache line, data items that will never be used are more likely to be brought into the cache. This will increase the traffic on the bus. Also, increasing the line size means there will be more data in the cache (under the infinite cache assumption) and so processors are more likely to interfere with each other, and force invalidations in other caches. These invalidations, as well as the subsequent refetches, also cause the traffic to increase. On the other hand, it is possible for a longer cache line to cause a traffic decrease as well. If there are several shared data items stored relatively close to each other, then a single invalidation of a long cache line could cause them to all be invalidated in one operation. This can decrease traffic compared to the case of several individual invalidations. For the cases considered here, this last situation happens infrequently, so its effect is minor compared to the first two. Thus, we expect that increasing the cache line size will lead to an increase in the number of bytes transferred.

Table 3: Traffic as a function of cache line size in shared memory version.

Circuit	Cache Line Size	MBytes Transferred
bnrE	4	2.15
	8	3.73
	16	6.87
	32	13.5

As predicted, the data in Table 3 clearly shows that the traffic increases significantly as the line size increases. For example, a cache line size of 4 bytes causes the total traffic to be 2.15 megabytes while a 32 byte cache line causes the traffic to increase to 13.5 megabytes, more than six times as much.

Comparing these figures with those presented in Section 5.1, we see that the communication traffic in the message passing approach is 1-3 orders of magnitude less than that for the shared memory approach. This surprisingly large difference can be explained by several factors. The updates being performed in the message passing version occur, at most, once per wire, so the write performed at the wire rip up stage is handled at the same time as the write performed at the wire routing stage. Since much of the wire's path will remain the same after rerouting, these two writes will often cancel each other in the delta array, and many of the locations will not need to be updated at all. By

[3] Traffic is also a function of the cache size, because a small cache will have a higher miss rate requiring more data fetches from main memory. For the purposes of this study, we have assumed an infinite cache.

contrast, the shared memory approach may require consistency operations on any individual read or write operation. The cancellation possible in the message passing approach removes many of the write operations—a significant accomplishment since over 80% of the bytes transferred in the shared memory version are caused by writes.

The shared memory version gives a circuit height of 131 for the bnrE circuit. This is about 8% better than that given by the sender initiated message passing approach. Clearly, the cost of this improved quality is in the increased network traffic required to maintain consistency of the cost array to such a large degree. However, for mass production situations where an improvement in routing quality can lead to reduced materials costs and higher yields, the shared memory version appears to be the method of choice.

5.3 Effect of Locality

The solution quality, execution time and amount of network traffic generated in both the shared memory and message passing versions of LocusRoute depend on the degree to which locality can be exploited. Here, locality is a measure of how often a processor is routing wires within its owned region or regions close by. (A quantitative measure is described in Section 5.3.3.) Architectures benefit in different ways from exploiting locality. Message passing architectures benefit from locality because the need for message traffic to produce a certain level of solution quality is reduced. In this section we show that message passing implementations taking advantage of locality can reduce the total network traffic by as much as 63%. Shared memory architectures benefit from locality through better cache behavior. Specifically, exploiting locality in a shared memory approach results in less processor interference causing cache coherence traffic, and provides better spatial locality. In the past, locality has not played a major part in the design of shared memory parallel programs. However, in hierarchical shared memory architectures, now being considered because of their scalability, a local reference can be more than an order of magnitude faster than a non-local reference. This architectural trend indicates that locality will become an important part of future program design.

5.3.1 Locality in the Message Passing Approach

Table 4 shows the effect of various wire assignment strategies on the quality of the routed circuit, the execution time, and the number of bytes transferred. The extreme non-local case is one which uses round robin wire assignment, and the extreme local case (ThresholdCost = infinity) is one where each wire is assigned to the processor whose owned region contains its leftmost pin. Clearly, wire assignments which do not take advantage of locality, such as round robin, result in poorer quality than those that do, such as assignments made with ThresholdCost set equal to 1000 or infinity. (See Section 4.2 for an explanation of the ThresholdCost parameter.)

The effect of locality on the network traffic depends on the type of update strategy used. In the sender initiated scheme, updates are sent out if the sender's array has changed. A reduction in traffic due to locality will occur because changes are made in fewer and smaller regions of the cost array. The change in traffic for sender initiated updates from a fully local assignment to a round robin assignment is 11%. The receiver initiated scheme is more sensitive to locality, because in this strategy, poor locality results in frequent interprocessor data requests. Traffic is reduced as much as 63% going from a round-robin assignment policy to a local one.

Table 4: Effect of locality (Sender Initiated).

Ckt.	Asmt. Method	Ckt. Ht.	MBytes Xfrd.	Time (s)
bnrE	round robin	147	.156	1.478
	ThresholdCost = 30	141	.153	1.392
	ThresholdCost = 1000	141	.140	1.445
	ThresholdCost = inf.	140	.139	2.468
MDC	round robin	150	.242	2.181
	ThresholdCost = 30	146	.232	1.768
	ThresholdCost = 1000	147	.217	1.866
	ThresholdCost = inf.	146	.220	3.684

Locality also has an effect on the quality of the output and execution time required by LocusRoute. By using a totally local wire assignment, the solution quality improves as much as 5%. When processors route in localized regions, each has a fairly consistent view of the area it is routing in. Ultimately, this is a more effective way to produce good solution quality than nonlocalized routing with periodic updates.

5.3.2 Locality in the Shared Memory Approach

This section examines the sensitivity of the shared memory approach to locality. Table 5 shows the solution quality and amount of traffic generated, as a function of the amount of locality exploited in the application.

Table 5: Effect of locality in shared memory version.

Ckt.	Asmt. Method	Ckt. Height	Mbytes Xfrd.
bnrE	round robin	139	3.96
	ThresholdCost = 30	134	3.77
	ThresholdCost = 1000	131	3.73
	ThresholdCost = infinity	139	3.73
MDC	round robin	144	4.833
	ThresholdCost = 30	138	4.625
	ThresholdCost = 1000	143	4.600
	ThresholdCost = infinity	143	4.687

For bnrE with 8 byte cache lines, total global bus traffic can be reduced 5.8% by taking advantage of locality in the assignment of wires. While a small reduction, it will still be an important factor in the performance of a hierarchical shared memory machine, in which non-local memory accesses can be more than an order of magnitude slower than local memory accesses. In most cases, a local wire assignment results in improved quality over the round robin assignment as well. Quality is improved by nearly 6%. Because the improvement in quality comes without any increase in network traffic, it is definitely a useful optimization. These small gains in circuit quality can lead to large payoffs in terms of material costs and yield.

5.3.3 Limitations on Exploiting Locality

Exploiting locality to reduce network traffic and increase quality has clear benefits. However, several factors limit the amount to be gained by taking advantage of locality in a problem.

First, the standard cell circuits have only a limited amount of locality. If the circuit's wires are long enough to pass through the regions of several processors, there is an unavoidable amount of interprocessor communication that will take place to perform the necessary updates. To determine an upper bound on the locality LocusRoute could exploit, we developed a measure of the locality in standard cell circuits. The locality measure is a weighted average indicating the average distance (in horizontal or vertical hops) between the processor actually routing a wire segment, and the processor that owns the region that segment lies in. Thus, a locality measure of 0 indicates that all segments were routed by the region owner, giving perfect locality. Increases in this measure indicate that the average segment is being routed at a distance further from the owner.

Computing the locality for the bnrE circuit with several wire allocation strategies, we find that even with the most local wire assignment strategies, wire segments are routed an average of 1.21 processors away from the owner processor. The MDC circuit had better locality, with wires routed an average of 0.91 processors away from the owner. As the number of processors is increased, the locality of the circuit will be degraded, because the region of the cost array owned by each processor will become smaller.

The second limitation to exploiting locality is the constraint that the processors have balanced workloads. If many wires lie within a single processor's region, then considering locality alone, that processor should route them all. However, this can give that processor an unfair amount of work, resulting in a load imbalance and poor performance. To some extent, a circuit with good locality will require fewer updates, and therefore, less time to execute. However, the effect of a load imbalance can outweigh the subtle effect of the difference in update time. Therefore, in terms of execution time, the optimal point is neither a fully load balanced circuit, nor a fully local circuit, but rather a point between the two. For example, in Table 4, the best execution time is always given by the wire assignment with a ThresholdCost of 30.

5.4 Number of Processors

The behavior of a parallel program as the number of processors is increased is an important consideration when parallelizing an application. In this case, the main limitation to scaling the message passing LocusRoute application is the distributed cost array. As the cost array is divided among more processors, more frequent updates will be needed to keep it consistent. These updates will take extra processing time. Also, the optimization in the sender initiated update strategy to send absolute updates only to the four near neighbors of the sender will be less effective, because as the owned regions decrease in size, an increasing number of processors will need the update. Table 6 shows the effect of increasing the number of processors on the measures of quality, time, and network traffic.

Table 6: Effect of number of processors (Sender Initiated).

Ckt	Num Procs.	Ckt. Ht.	Occup. Factor	MBytes Xfrd.	Time (s)
bnrE	2	131	415142	.245	8.438
	4	137	421041	.263	4.378
	9	143	425426	.178	2.184
	16	141	429589	.140	1.445

As expected, the occupancy factor and circuit height are degraded by the addition of more processors. For these circuits, the circuit height degradation is around 6%. As the number of processes is increased further, the number of wires being simultaneously routed will increase. Also, the number of cost array regions spanned by each wire will increase. For a fixed update schedule, both of these factors will lead to poorer quality results as the number of processors is increased. The shared memory version also produces poorer quality results as the number of processors is increased [9]. For a larger number of processors, better heuristics for locality based wire assignment, and more strict methods for maintaining consistency in the distributed cost array will be required to maintain a useful level of routing quality.

Next, we examine the effect increasing the number of processors has on execution time. (Here we calculate speedup with respect to the two processor run, and then multiply by two.) Using the bnrE circuit, for 16 processors, the speedup is 12 with a sender initiated update strategy. For MDC, a larger circuit, the speedup is slightly better, reaching 12.8 for the sender initiated strategy. These speedup values are comparable with the speedup measured for the original shared memory program.

The network traffic also depends on the number of processors. After four processors, the network traffic actually begins to decrease as the number of processors increases. However, this is not an indication that less communication is required. Clearly, since the quality is rapidly degrading, more frequent updates are required as the number of processors increases. What this decrease in network traffic shows

is that the updates sent out for each owned region, which are bounding boxes of all the changes made in the region, contain fewer wasted bytes, because the owned region is smaller. Except for this effect, one expects the need for communication to increase as the number of processors is increased.

6 Conclusions

We have examined two versions of the LocusRoute application, written in message passing and shared memory styles. In the message passing approach, the global cost array is kept consistent using messages to perform explicit updates. For this approach, we explored a number of strategies for updating the cost array. The shared memory approach uses a centralized cost array, relying on underlying coherence mechanisms to keep the data consistent.

In general, the shared memory version of LocusRoute gives better solution quality than the message passing approach for any of the update strategies. Intuitively, this is because the shared memory approach maintains greater consistency of the cost array than any of the message passing strategies. For a circuit that will be mass produced, the 5-15% improvement in quality afforded by the shared memory approach can represent a significant cost savings. Of the message passing strategies, sender initiated updates produced the results with the best circuit height.

However, better solution quality has a cost associated with it. The network traffic measured for the shared memory approach was significantly higher than that for any of the message passing approaches, especially the receiver initiated update method which had the lowest network traffic. The network traffic of the shared memory approach was about 10 times higher than that for the sender initiated message passing strategy, which in turn was about 10 times higher than that for the receiver initiated strategy.

Exploiting locality in the task assignment is an effective way of improving the quality of results in both the shared memory and message passing approach. In the message passing approach, solution quality was improved nearly 10% by assigning wires to processors on the basis of their location in the circuit. Exploiting locality also decreases the network traffic by up to 63%. Making use of locality in the shared memory approach also had a favorable effect on solution quality and network traffic. Quality can be improved by about 6% by assigning wires to processors based on their location in the circuit. With this quality improvement comes a reduction in network traffic as well. We further note that the amount of locality in a typical standard cell circuit is limited because long wires can stretch across the owned regions of several processors. Also, wire assignment policies which strictly enforce locality can lead to poor load balancing, with large execution time degradation. More sophisticated wire assignment heuristics may further improve quality and reduce traffic, but may increase the time spent on the static wire assignment phase.

The solution quality and network traffic are also strongly dependent on the number of processors being used. As the number of processors increases, the number of wires being routed in parallel increases, so the information available to each processor as it routes a wire is less accurate. Both the shared memory and message passing approaches suffer quality degradations of 5-10% as the number of processors is increased to 16. However, our methods for exploiting locality can mitigate somewhat the effects of scaling to a larger number of processors.

Since the message passing style of programming can be efficiently implemented on shared memory machines, the results presented here are not intended to make a statement about the relative merits of shared memory or message passing *architectures*. Rather, our results stress the effect that programmer effort can have on the efficiency of the data structures, and on the parallel decomposition in general. The message passing paradigm requires the programmer to explicitly grapple with how to keep shared data structures consistent. This extra effort can result in lower network traffic than the shared memory approach which might be considered the more "natural" implementation. However, if the amount of traffic present in the shared memory approach is not excessively high, then the extra effort required to implement the distributed approach may not be worthwhile. Finally, while our results apply specifically to LocusRoute, the analysis performed here should serve as a guide for the implementation of other similar applications.

7 Acknowledgments

We would like to thank Jonathan Rose for his help with the LocusRoute application, and Andreas Nowatzyk for his quick replies to questions about CBS. Thanks should also go to Monica Lam and Wolf-Dietrich Weber for their helpful comments on earlier versions of this paper. This work was supported by DARPA contract N00014-87-K-0828. In addition, Margaret Martonosi is supported by a fellowship from the National Science Foundation and Anoop Gupta, by a faculty award from Digital Equipment Corporation.

References

[1] Ametek Computer Research Division. *Series 2010 System General Description Issue 3*. 1988.

[2] J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Trans. Computer Systems*, 4(4):273–298, Nov. 1986.

[3] W. J. Dally. *A VLSI Architecture for Concurrent Data Structures*. Kluwer Publishers, 1987.

[4] W. J. Dally. Wire Efficient VLSI Multiprocessor Communication Networks. In *Stanford Conference on Advanced Research in VLSI*, pages 391–415, 1987.

[5] Encore Computer Corp. *Multimax Technical Summary*. 1986.

[6] S. R. Goldschmidt. Tango Methodology. Unpublished report, 1989.

[7] A. Nowatzyk. CBS: A Message Passing Cube Simulator. Unpublished report, 1988.

[8] J. Rose. LocusRoute: A Parallel Global Router for Standard Cells. In *Design Automation Conference*, pages 189–195, June 1988.

[9] J. Rose. The Parallel Decomposition and Implementation of an Integrated Circuit Global Router. In *Proc. ACM SIGPLAN Parallel Programming: Experience with Applications, Languages and Systems (PPEALS)*, pages 138–145, July 1988.

[10] C. L. Seitz, W. C. Athas, et al. The Architecture and Programming of the Ametek Series 2010 Multicomputer. In *Hypercube Concurrent Computers and Applications*, 1988.