A Comparison of Shared and Nonshared Memory Models of Parallel Computation

RICHARD J. ANDERSON AND LAWRENCE SNYDER

Invited Paper

Four algorithms are analyzed in the shared and nonshared (distributed) memory models of parallel computation. The analysis shows that the shared memory model predicts optimality for algorithms and programming styles that cannot be realized on any physical parallel computers. Programs based on these techniques are inferior to programs written in the nonshared memory model. The "unit" cost charged for a reference to shared memory is argued to be the source of the shared memory model's inaccuracy. The implications of these observations are discussed.

I. Introduction

The single goal of parallel computation is performance. As always, performance is achieved by an efficient program optimally compiled for and executed on a powerful machine. We are interested here in designing efficient parallel algorithms and writing fast parallel programs. It should be clear that being successful at these tasks requires an accurate understanding of the costs involved. How time consuming is a barrier synchronization? How expensive is a memory reference? Unfortunately, there is a complication with accurately knowing the costs in the parallel setting that does not arise in sequential computing.

In sequential computation, the von Neumann machine model is a sufficiently accurate description of the physical hardware to be the basis for efficient algorithm design and programming, as well as language design and compiler optimizations. In parallel computation, however, it is not yet clear which architecture will emerge as the most performant, or even if it will be a single machine type. We are confronted with a number of very diverse parallel architectures: hypercubes, butterflies, clusters, and shared bus machines, to name only representative MIMD machines. These machines differ in fundamental ways that affect the costs algorithm designers and programmers must know.

One approach, illustrated by "blue collar" programs [8], is to write low level, machine specific programs that exploit the characteristics of particular computers. Although this may achieve high performance, it greatly decreases the portability of the program, more severely limiting its wide use than in the sequential case. At the other extreme, many researchers have advocated high level languages that are sufficiently "far" from any architecture as to be independent of particular machine features. This solves the portability problem in the sense that the program does not rely on any specific machine features, but portability in the parallel context requires more. The program not only must run on another machine, it must run *well.* Compilation techniques are not yet powerful enough "to span the distance" between high level programs and low level parallel machines with efficient code.

An intermediate approach, allowing us to overcome the problems of many diverse machines while possibly achieving portability and high performance, is to develop a compromise abstract machine incorporating features critical to a variety of architectures. Such a machine has been called a parallel *type architecture* [15]. The goal of the type architecture is to possess those "essential" capabilities that all or most parallel machines can realize efficiently. It avoids "features" peculiar to specific machines that do not scale well. Programs that are efficient on the logical type architecture should also be efficient on physical parallel machines, assuring portability with performance.

The most important feature of the type architecture approach is that it enables the algorithm designer and the programmer to have a realistic model of the costs of parallel computation. For example, the type architecture can indicate by its characteristics whether barrier synchronization is an expensive operation or an inexpensive operation and thus can be avoided or exploited by the programmer. Since there is not yet a consensus on a parallel type architecture (only a candidate has been proposed [15]), it is not possible to consider all cost sensitive aspects of parallel machines. Accordingly, we will concentrate here on one specific feature of parallel computers—memory structure.

The goals are to demonstrate how the choice of memory

Manuscript received October 30, 1990; revised December 24, 1990.

The authors are with the Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195.

IEEE Log Number 9143456.

0018–9219/91/0400–0480$01.00 © 1991 IEEE

model affects the performance of parallel programs, to illustrate that the nonshared memory model is superior to the shared memory model because it leads to better performance, and to justify the advice to algorithm designers and programmers that the nonshared memory model is to be preferred over the shared memory model. As a corollary, it will follow that nonshared memory should be a component of the parallel type architecture.

The approach to realizing these goals is first to define shared and nonshared memory models to be used as the basis for analyzing parallel algorithms and to introduce two problems that exhibit different characteristics when solved with the different memory models. Then for each problem and each model an algorithm will be developed. The performance of the four resulting algorithms will be the basis of analysis and comparison. The results, that the nonshared memory solutions are superior and the shared memory programs are inferior, will guide the subsequent discussion.

II. Shared and Nonshared Memory Models

To understand the impact of the memory model on algorithm design, we will introduce shared and nonshared memory parallel computer models. Two variants of the shared memory model will be defined in order to cover the two most commonly used forms.

The models will have P processors, each with the usual processing capabilities and its own program counter, i.e., the models are MIMD. All models will be assumed to have the same memory capacity. They will differ principally in memory organization.

In the shared memory models there is a single (flat) memory, each location of which is accessible by any of the P processors. In accordance with prevailing definitions, a reference to the shared memory takes "unit" time independent of the size of the memory or the pattern of the other processor's references [6]. Since any processor can reference any memory location, it is necessary to define what happens when two processors reference the same address. The two most common formulations either permit multiple processors to reference the same location, called the concurrent-read-concurrent-write (CRCW) model, or forbid multiple processors from referencing the same location, called the exclusive-read-exclusive-write (EREW) model. (Because the situation is asymmetric, making multiple reads somewhat more plausible than multiple writes, there is also the CREW model too.) Various approaches have been used to give the CRCW model a well-defined outcome when multiple processors write different values to the same location, including selecting the value from the processor with the smallest ID, selecting a random processor to be "winner," etc. Since our use of the CRCW will have the property that all of the processors will be writing the same value, no specific choice is required. We will refer to the two shared models as *SC* and *SE*. Notice that these are both *models* of computation that cannot be physically realized [15].

The nonshared or local memory model will have its memory partitioned into P modules, one associated with each processor. Reference by a processor to a location in its associated memory takes "unit" time. It is not possible for a processor to reference any other memory directly. Information is exchanged among the processors via messages sent over a communication network that connects the processors together directly. The specific topology of the network is not important, provided it has two properties needed in the later development: logarithmic diameter and bounded degree. Many topologies have these properties including the shuffle exchange [11], the cube connected cycles [9], and certain single-stage interconnection networks [13]. The time to communicate between two adjacent processors over the network will be considered to take a constant amount of time.[1] This local memory model will be referred to as L.

III. Two Problems

The two problems, selected to illustrate the effects of shared memory on practical program performance, exhibit several important properties. They are simple in concept. The solutions in both models are representative of common programming techniques for the models, and so will illustrate typical stylistic forms. Most importantly, the algorithms can be proved to be optimal for their respective memory models, thus removing the possibility that the poor problem solution is due to a poor choice of algorithm.[2]

The problems:

FINDING THE MAXIMUM: Given N distinct integers, find the value with the largest magnitude.

Take $N = P$ unless otherwise stated. Also notice that because the "word model" is being used, i.e., every value fits into a word and a whole word is processed in any operation, the magnitude of the integers is unimportant.

EVEN/ODD PAIR TEST: Given a permutation $a_1, a_2, \cdots, a_{nP}$ of the integers $1, \cdots, nP$, determine if any of the n-blocks $a_{nk+1}, a_{nk+2}, \cdots, a_{nk+n}$, $0 \leq k < P$ contain an even/odd pair, i.e., an a_{nk+i} and a_{nk+j}, such that $a_{nk+i} = 2x$ and $a_{nk+j} = 2x + 1$ for some x, and $1 \leq i, j \leq n$.

Take $n = \log P$ unless otherwise stated. This is a specialized computation to illustrate features of the different models and is not necessarily of practical utility.

IV. Four Algorithms

In the following subsections four algorithms will be developed to solve the maximum and even/odd pair problems using the global and local memory models.

[1] This assumption is not literally true since as larger and larger computers are considered the networks just mentioned must have longer and longer wires [10]. It has been argued, however, that such considerations apply to all computers independent of memory organization [15].

[2] For the shared memory models, the correctness proofs are based on SIMD versions of the machines. At the level of detail of the models, however, the SIMD version can simulate the MIMD version with only a constant difference in performance. The conclusions will not be affected.

A. Maximum Finding, Shared Memory Model

The best approach to finding the maximum, assuming a CRCW shared memory model, is an ingenious algorithm due to Valiant [17]. It finds the maximum in $O(\log\log N)$ time, based on the shared memory assumptions. By grouping the numbers into sets and making all possible comparisons for each set simultaneously, it finds a maximum element for each set; these go on to be inputs to another application of the same strategy.

All Compares Algorithm: The maximum is found in stages. At stage i the input, $n_1, n_2, \cdots, n_{s(i)}$, is partitioned into the fewest number r of sets $S_1, S_2, \cdots, S_r$ of like size ($|S_i| = |S_j|$, $1 \le i, j < r$ and $|S_r| \le |S_1|$) such that

$$\sum_{k=1}^{r} \binom{|S_k|}{2} \le P. \qquad (*)$$

A separate processor is assigned to perform each of the $\binom{|S_k|}{2}$ distinct comparisons $n_u : n_v$ of elements in each set S_k. There are enough processors by the condition $(*)$ on the cardinality of the sets so all comparisons can be performed simultaneously; the CRCW model permits the necessary concurrent reads to memory

Each set S_k has a bit vector of length $|S_k|$ associated with it that is set to 1's at the start of the stage. Once the comparisons are made, the processor performing the $n_u : n_v$ comparison sets to 0 the bit corresponding to the smaller value; the CRCW model permits the necessary concurrent writes to memory. (Notice that the value stored by any two processors to the same memory location is always the same.) One position, the vth say, will remain set to 1, and so n_v was the maximum of the set; it moves on to be an input to stage $i+1$. □

Many details, which can be found in [17] and [12], have been omitted. (See Example 1.) The key points of the algorithm are these: Each stage can be performed by P processors in *constant* time assuming a CRCW shared memory. There are $O(\log\log P)$ stages and so $O(\log\log P)$ running time. No algorithm can solve this problem (asymptotically) faster using shared memory and P processors [17].

Example 1: The All Comparisons maximum finding algorithm applied to 1000 values takes four stages.

Stage 1: The 1000 values are partitioned into 333 sets of 3 elements each and a set of 1 element requiring 999 comparisons.

$1000 = 333 \times 3 + 1$ values.

$$999 = 333 \times \binom{3}{2} + 1 \times 0 = 333 \times 3 + 0 \text{ comparisons.}$$

Sets	Comparisons
$\{a_1, a_2, a_3\}$	$a_1 : a_2, a_1 : a_3, a_2 : a_3$
$\{a_4, a_5, a_6\}$	$a_4 : a_5, a_4 : a_6, a_5 : a_6$
...	...

Stage 2: The winners of the 334 sets of Stage 1 advance and are partitioned into 47 sets of 7 elements each and a set of 5 elements requiring 997 comparisons.

$334 = 47 \times 7 + 5$ values.

$$997 = 47 \times \binom{7}{2} + \binom{5}{2} = 47 \times 21 + 10 \text{ comparisons.}$$

Sets

$\{a'_1, a'_2, a'_3, a'_4, a'_5, a'_6, a'_7\}$

...

Comparisons

$$\begin{aligned} a'_1 : a'_2, a'_1 : a'_3, a'_1 : a'_4, a'_1 : a'_5, a'_1 : a'_6, a'_1 : a'_7 \\ a'_2 : a'_3, a'_2 : a'_4, a'_2 : a'_5, a'_2 : a'_6, a'_2 : a'_7 \\ a'_3 : a'_4, a'_3 : a'_5, a'_3 : a'_6, a'_3 : a'_7 \\ a'_4 : a'_5, a'_4 : a'_6, a'_4 : a'_7 \\ a'_5 : a'_6, a'_5 : a'_7 \\ a'_6 : a'_7 \end{aligned}$$

...

Stage 3: The winners of the 48 sets of Stage 2 advance and are partitioned into 2 sets of 24 elements requiring 552 comparisons.

$48 = 2 \times 24$ values.

$$552 = 2 \times \binom{24}{2} = 2 \times 276 = 552.$$

Stage 4: The winners of the 2 sets of Stage 3 advance and are compared, yielding the result. □

B. Maximum Finding, Local Memory Model

The best algorithm for finding the maximum of N numbers stored in the unit-cost local memories of $P = N$ processors is the tournament algorithm. This assumes that the machine's communication structure "contains" a tree of height = diameter of the network, which is the case for the graphs cited in local memory model definition. Notice that by the assumption of a constant degree, the tree has height $\Omega(\log P)$.

Tournament Algorithm: Assume a value in each processor's memory and assume a tree is embedded in the communication structure of the machine. Name the processors according to their role in the tree and permit them to refer to their neighbor processors by their logical relationship. Initially, each leaf sends its value to its parent processor:

process leaf(val : **int**, parent : **port**);
begin parent ← val **end**.

Interior nodes read the values from their children, waiting if the value hasn't arrived; they then pass to their parents the largest of their children's values and their own:

process intnode(val : **int**, lchild, rchild, parent : **port**);
begin parent ← max(val, lchild, rchild) **end**.

The maximum is the value "emitted" by the root. □

With a unit time memory reference and a constant time channel transmission, the execution of the algorithm can be divided into constant time steps, where a processor at height i is active only at step i. This gives an execution time of $O(\log P)$, the best possible for the local memory model because the assumed $\Omega(\log P)$ network diameter implies $\Omega(\log P)$ time is required to bring the values together.

Schwartz [11] observed that the Tournament Algorithm can find the maximum of $N = O(P \log P)$ values stored $O(\log P)$ per processor with no increase in asymptotic time complexity. This is accomplished by initially employing each processor to find the local maximum of the $O(\log P)$ values using a sequential search. The results are then treated as the values of the original algorithm. Indeed this seems to be a satisfactory way to solve the problem no matter how many elements there are.

Compare the two algorithms for finding the maximum. 1) The All Compares algorithm requires $O(N \log\log N)$ total instructions (work), while the Tournament algorithm executes $O(N)$ instructions. 2) The All Compares algorithm has essentially full processor utilization, while the processors in the Tournament algorithm, when $N = P$, execute only a constant amount of time.

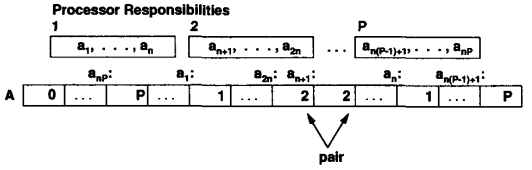
C. Even/Odd Pair, Shared Memory

To contrast with the use of the powerful CRCW model to solve the maximum problem, we use the weaker EREW model to compute the Even/Odd Pair test. There will be no simultaneous memory references to any cell. The provably best algorithm uses a global array and three phases.

Indexing Algorithm for Even/Odd Pair Test: Recalling that the input $a_1, a_2, \cdots, a_{nP}$ is a permutation, declare an array $A[0 : n^*P]$. In the first phase of the algorithm each of the processors loops through the n integers for which it is responsible; using each value a_{nk+i} as an index into A, it assigns k, its processor ID, i.e., $A[a_{nk+i}] := k$. In the second phase of the algorithm, each processor again loops through its n integers; for each a_{nk+i} that is odd, it checks if $A[a_{nk+i} - 1] = k$; if equality is found for any of the checks, the processor reports **true** during the final combining phase; otherwise it reports **false**. The combining phase OR's the Boolean values from each processor together. The details of the combining phase are irrelevant here. □

The reason the details of the combining phase are unimportant is because we take $n = \log P$. This implies that the first two phases each use $O(\log P)$ time, since each has performance proportional to the size of the list. There are many solutions, such as a combining tree, that will implement the combining phase in $O(\log P)$ time. The Indexing algorithm thus runs in $O(\log P)$ time on P processors with an EREW shared memory model. Notice that the algorithm relies on the fact that the input is a permutation, and that $A[0]$ is required simply to guarantee "in bounds" indexing.

$O(\log P)$ is the fastest possible performance for these assumptions, since it takes that long to touch all of the data.



Example 2:

D. Even/Odd Pair, Local Memory

Assuming each processor has n values stored in its local memory, the optimal local memory solution also uses three phases.

Ordering Algorithm for Even/Odd Pair Test: In the first phase, each processor sorts its n values into order locally. In the second phase each processor loops through the sorted list, checking the successor of any even element to determine if it is one larger, i.e., checking if $a_{nk+i} = 2x$ for some x and if $a_{nk+i+1} = a_{nk+i} + 1$; if so, it reports **true** during the final combining phase; otherwise it reports **false**. The details of the combining phase are again irrelevant. □

Selecting $n = \log P$, which is large enough to permit the combining phase to be a combining tree (it could even be the Tournament Algorithm on the Boolean values of the outcomes), the performance is dominated by the sorting phase. Thus the Ordering Algorithm can achieve $O(\log P \log\log P)$ time, which is optimal for the Local Memory model.

Notice that because local memory references suffice for determining whether a processor has an even/odd pair, the indexing tactic used in the Indexing algorithm could have been used here. But this would require each processor to have an $nP + 1$ element array. Totaled, this is substantially more memory (by a factor of P) than was used by the Indexing Algorithm. Given that we want to compare the algorithms, it is necessary to keep the memory requirements similar.

To summarize the results of this section:

Maximum Finding—	
All Compares Algorithm (Shared)	$O(\log\log P)$
Tournament Algorithm (Local)	$O(\log P)$
Even/Odd Pair—	
Indexing Algorithm (Shared)	$O(\log P)$
Ordering Algorithm (Local)	$O(\log P \log\log P)$.

Evidently, the memory reference time significantly affects the asymptotic performance of algorithms.

V. Programming the Problems

Having introduced two problems and having developed four optimal algorithms to solve them with different memory models, it is now possible to consider programming realistic machines. We imagine a programmer, confronted with the task of writing programs in programming languages differing in their memory models.

A. Shared Memory

Suppose the programming language provides a shared memory for data storage. If the details of the language

are best characterized by the CRCW model, a rational programmer should choose algorithms that are optimal for it. Our example is the All Compares algorithm for finding the maximum. Performance of $O(\log \log P)$ can be predicted for the resulting program based on the model. The Tournament algorithm, which can be compiled for this model by simulating the embedded tree, can be dismissed since its $O(\log P)$ predicted performance is too slow.

Similarly, if the programming language is best characterized by an EREW shared memory model, the rational programmer will choose algorithms that are optimal for it. Our example is the Indexing algorithm for testing for an even/odd pair. Its $O(\log P)$ predicted performance is superior to the $O(\log P \log \log P)$ predicted performance of the Ordering algorithm in this model.

Thus the programmer considers the capabilities of the model presented by the language and selects the best algorithm with respect to them. The problem arises when these predicted times are compared with the times that can be realistically expected when the program is executed on a physical machine. To estimate what realistic performance to expect, we return to our previous observation that the two shared memory models are not physically realizable.

There are numerous problems with a physical implementation of these shared memory models [15] and a complete treatment would be too great a diversion at this point. The principal problem with both is the "unit" time memory reference assumption. Simultaneous reference by P processors to P memory locations must take at least $\Omega(\log P)$ time, based simply on constant fanin/fanout considerations. This bound does not take into account delays due to collisions in the interconnection network, if that is how sharing is implemented; Borodin and Hopcroft [3] show, for oblivious routers, today's state-of-the-art, that these delays can be as large as $\Omega(N^{1/2})$. Also, this bound does not account for serialization at the memory module if multiple processors reference the same module (not just the same location, cache line or page). These can increase the reference time substantially. A variety of tricks have been proposed to neutralize the $\Omega(\log P)$ lower bound, but none has proved adequate, as will be explained.

Even though the "unit" cost assumption is not literally true, the algorithms can be run on physical parallel hardware provided the requirements of the memory model are simulated. Being optimistic, assume that the shared memory references can be accomplished in $O(\log P)$ time. Then a simulation will be slowed down by a factor of $O(\log P)$ giving the following times for each of the four programs:

All Compares$_{SC}$	$O(\log P \log \log P)$
Tournament$_{SC}$	$O((\log P)^2)$
Indexing$_{SE}$	$O((\log P)^2)$
Ordering$_{SE}$	$O((\log P)^2 \log \log P)$.

The realizable performance is worse than the predicted performance.

B. Nonshared Memory

Suppose now that the programming language provides nonshared memory for data storage. The rational programmer will select algorithms that perform well for that model. For the two problems considered above, the predicted performance is $O(\log P)$ for the Tournament algorithm and $O(\log P \log \log P)$ for the Ordering algorithm. These must be compared with the shared memory algorithms which can be implemented in the local memory model by simulating shared memory at a cost of $t(P)$ for each step. Accepting the $\Omega(\log P)$ lower bound as the estimate for $t(P)$, gives a predicted performance of $O(\log P \log \log P)$ for the All Compares and $O\left((\log P)^2\right)$ for the Indexing algorithms in the local memory model. As expected, the optimal local memory algorithms are predicted to be best.

But, how do the programs perform on physical machines? Here the contrast between models is apparent, since unlike the shared memory case, the nonshared memory programs can be physically realized by machines such as hypercubes. That is, the local memory references are unit time and the logarithmic height tree can be directly embedded with a constant time traversal of each edge. Thus the realizable performance is:

All Compares$_L$	$O(\log P \log \log P)$
Tournament$_L$	$O(\log P)$
Indexing$_L$	$O((\log P)^2)$
Ordering$_L$	$O(\log P \log \log P)$.

The realizable performance for the programs based on the local memory model matches the programmer's expectations, unlike the case of shared memory model.

VI. Summary of the Results

We have compared the predicted performance of programs written in different programming languages with the realizable performance for those programs on physical machines. In order to estimate the time $t(P)$ required to implement the shared memory model on a physical machine, we chose $t(P) = O(\log P)$. The following table summarizes the results.

	Predicted	Realizable
Shared Memory Pgmming Model		
All Compares	$O(\log \log P)$	$O(\log P \log \log P)$
Tournament	$O(\log P)$	$O((\log P)^2)$
Indexing	$O(\log P)$	$O((\log P)^2)$
Ordering	$O(\log P \log \log P)$	$O((\log P)^2 \log \log P)$
Local Memory Pgmming Model		
All Compares	$O(\log P \log \log P)$	$O(\log P \log \log P)$
Tournament	$O(\log P)$	$O(\log P)$
Indexing	$O((\log P)^2)$	$O((\log P)^2)$
Ordering	$O(\log P \log \log P)$	$O(\log P \log \log P)$.

The two observations, to be discussed in the next section, are that the local memory model has the best realizable performance for both problems and that the local memory model's predictions match what should be realizable.

Before interpreting the results, however, it is important to ask whether the comparison is meaningful. Specifically, the items in the northeast quadrant of the table represent execution of the programs on a shared memory machine

while the programs in the southeast quadrant of the table represent execution on a nonshared memory machine. It might be argued that since the time to make a shared memory reference on a shared memory machine is an order of magnitude faster *on today's machines* than referencing an adjacent processor on a nonshared memory machine, it is meaningless to compare the two quadrants.

The comparison is justified for several reasons. First, the behavior of present day machines is an unreliable guideline since they are small, they are implemented in a variety of technologies, they represent early designs that will likely be improved, etc. Counting time in seconds on present machines does not produce useful information for this purpose. Second, the principal cost in PE-to-PE communication on present nonshared memory computers is packet formation time, which is independent of machine size, i.e., it is not likely to get larger as machines do. Shared memory reference times must increase. Third, the cost of implementing shared memory, $O(\log P)$, is generous in that it does not include congestion-related problems; such features may prevent shared memory scaling as $\log P$. Finally, many constants are being ignored; the fact that the shared memory programs make considerably more shared memory references than the local memory program's nonlocal references might alone erase the performance difference between the specific reference times.

Thus the analysis can give a coarse indication of how programs scale in the different models.

VII. Interpretation and Discussion

The first conclusion from the preceding analysis is that the nonshared memory model is accurate in that the predictions are realizable. An accurate cost model must be an essential property of a programming model or else programmers will be unable to make the multitude of decisions needed to produce efficient programs.

Two counter arguments to the need for accuracy prevail, the it's-a-small-cost argument and the latency-can-be-hidden argument.

It is true that the difference between the realizable performance of the shared and local memory programs is small. The factor is $\log \log P$ in one case and $\log P / \log \log P$ in the other. Indeed, it could only be as large as $t(P)$. The issue is not the magnitude of the difference, but that there is any difference at all. The difference is large enough to make inferior algorithms *appear* to be superior, causing the rational programmer to prefer an algorithm that is unrealizable. Since we do not know what the actual costs are of implementing this algorithm on physical hardware, it is not possible to assess how grievous this error is. But in accordance with the opening assertion of this paper, it must be assumed to be serious enough to be avoided.

Of course, the shared memory models are also claimed to be easier to program than the local memory models and so part of the it's-a-small-cost argument becomes one of cost benefit: perhaps the reduced programmer investment is worth whatever the cost is. However, this is more of a criticism of the sorry state of nonshared memory programming languages than a defense of shared memory, since presumably local memory languages can be made as convenient as shared memory languages with suitable development [16].

The latency-can-be-hidden argument states that there are techniques wherein the overhead required to implement shared memory, the $t(P)$ in our analysis, can be hidden and thus logically eliminated. Among the techniques that have been mentioned are: caching active data values at the processor to reduce congestion and speed communication, but there is the problem of keeping the memory coherent that probably limits how large P can realistically get [2]. Time multiplexing instruction interpretation [14] seems to be a scalable solution, but it relies on having $\Omega(Pt(P))$ active instructions underway at all times, which is not a property of many problems, such as the All Comparisons algorithm. Combining can reduce serialization at a memory location and reduce traffic in the network [4], but hot spots remain and there is no benefit when the collisions are for a different reference in the module. In summary, no technique has been demonstrated on a large machine, and each has serious drawbacks.

Postulate, however, that instruction multiplexing proves to be an effective technique for hiding latency. Then can we say that the shared memory model is accurate? No. The model's unit time memory reference assumption is accurate only when there are enough instructions available for interpretation to hide the latency (at least $\Omega(P \log P)$, but perhaps more). The model is still inaccurate when there are fewer, say only P, instructions available to execute at a time. Parallel computation still involves many one-per-processor operations (synchronizing, aggregation, broadcasting, etc.) that do not produce enough threads.[3] It is possible, of course, to invent new models that characterize the instruction multiplexing capability. They would have relatively expensive memory reference when there are only a few threads of execution and relatively cheaper memory reference when there are many threads. Valiant's *bulk-synchronous parallel* model is such an alternative, where the concept of *parallel slackness* captures the notion of "sufficiently parallel" [18]. The essential point is that the new models would differ from the shared memory model at least by not having a universal unit cost memory reference.

The second conclusion of the analysis is that the local memory programs have the best realizable performance. It is important that this performance be realizable on all machines, both shared and nonshared memory and also among nonshared memory machines with different topologies. Consider both cases.

A preliminary set of experiments shows that executing local memory model programs on shared memory machines produces better results than shared memory model programs [7]. The experiments were run on the BBN Butterfly and Sequent Symmetry. The local memory program was able,

[3] This is a parallel analogue to Amdahl's law. He observes that multiple processors will not speed up the sequential components of a program; here latency hiding cannot reduce the proportional-to-P components of the program.

for example, to exploit the fact that the Butterfly partitions its memory into shared and local regions by concentrating the computation in the faster local memory. The exploitation of local memory was not the only advantage of the nonshared memory programs: They were also coarser grain, a property that is crucial in nonshared memory computers. Large grain, which is one way to encapsulate locality, is a generally useful feature for most parallel computers. This suggests that there are other characteristics of parallel programs besides memory usage that influence parallel program performance.

The change in topology of a computer would seem to be a feature that shouldn't concern the programmer, yet an essential feature of the Tournament and Ordering algorithm's performance was the fact that the tree be directly embedded into the structure. This would seem to limit the portability of the program while the shared memory, having no connection to the implementing topology, should be more portable. In fact, recent advances in programming abstractions for nonshared memory parallel languages have provided a means of exploiting the topology in the algorithm and still being customizable to different machines [1]. As with the difficult-to-program criticism of local memory languages, the portability criticism may simply represent our relative inexperience with such languages.

A final point to emphasize is the distinction between the models presented by parallel programming languages and used by programmers for creating practical programs, and the theoretical models used by computer scientists to understand the fundamental limits of computation. The former must be accurate; the latter should have whatever properties are needed to expose the phenomena being studied. Indeed, the PRAM model, of which the CRCW and EREW instances were used here to justify algorithm optimality, have been the source of numerous fundamental insights [6]. It is sensible, when seeking to understand the limits of concurrency, to employ a model such as the PRAM where communication costs are completely ignored; one discovers, as we did in the All Compares algorithm, that finding the maximum is so easy that parallel processors can compute the result "faster" than the time required to bring the values together. In practical parallel computation, where communication costs exist, accuracy is essential.

VIII. Conclusion

We have analyzed the shared and nonshared memory programming models by comparing the realizable performance of programs written in each. It was shown that the unit-cost memory reference of the shared model, though perhaps a simplifying assumption, nevertheless leads to algorithms that are "impossibly" efficient. A rational programmer, following the dictates of the model, should prefer such solutions over the less efficient, though more realistic, alternatives. When the programs based on the impossibly efficient algorithms are run, their performance is worse than the apparently slower but more practical competitors. We conclude that the unit-cost memory model is therefore counterproductive.

The nonshared memory model has been shown to be more realistic, but it has also been criticized as being difficult to use. Clearly, creating abstractions to support convenient nonshared memory parallel programming should be a research priority. The apparently simple expedient of adding local memory to a shared memory language should be avoided unless it is made plain that the shared memory references require $t(P)$ units of time to complete.

References

[1] G. A. Alverson, W. G. Griswold, D. Notkin, and L. Snyder, "A flexible communication abstraction for nonshared memory parallel computing," in *Proc. Supercomputing '90,* pp. 584–593.

[2] J. Archibald and J. L. Baer, "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM TOCS,* vol. 4, no. 4, pp. 273–298, 1986.

[3] A. Borodin and J. Hopcroft, "Routing, merging and sorting on parallel models of computation," *J. Comput. Syst. Sci.,* vol. 30, pp. 130–145, 1985.

[4] A. Gottlieb, R. Grishman, C. Kruskal, P. McAuliffe, L. Rudolph, and M. Snir, "The NYU ultracomputer—Designing an MIMD shared memory computer," *IEEE Trans. Comput.,* vol. C-32, pp. 175–189, 1983.

[5] W. G. Griswold, G. A. Harrison, D. Notkin, and L. Snyder, "Scalable abstractions for parallel programming," in *Proc. 5th Distributed Memory Comput. Conf.,* 1990, pp. 1008–1016.

[6] R. Karp and V. Ramachandran, *Handbook of Theoretical Computer Science.* Vol. A. Cambridge, MA: MIT Press, 1990.

[7] C. Lin and L. Snyder, "A comparison of programming models for shared memory multiprocessors," in *Proc. Int. Conf. Parallel Processing,* vol. II, 1990, pp. 163–170.

[8] D. Mizell, "First 'blue collar' production parallel program survey," in *Conf. Rec. Comput. Aerospace,* AIAA, 1989.

[9] F. Preparata and J. Vuillemin, "The cube-connected cycles: A versatile network for parallel computation," *CACM,* vol. 24, no. 5, pp. 300–309, 1981.

[10] A. L. Rosenberg, "Three dimensional VLSI: A case study," *JACM,* vol. 30, no. 3, pp. 397–416, 1983.

[11] J. T. Schwartz, "Ultracomputers," *ACM Trans. Programming Languages and Systems,* vol. 2, no. 4, pp. 484–521, 1980.

[12] Y. Shiloach and U. Vishkin, "Finding the maximum, merging and sorting in a parallel computation model," *J. Algorithms,* vol. 2, pp. 88–102, 1981.

[13] H. J. Siegel, *Interconnection Networks for Large-scale Parallel Processing.* London, U.K.: Heath, 1985.

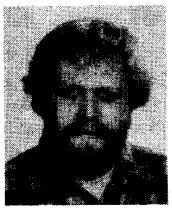
[14] B. J. Smith, "Architecture and applications of the HEP multiprocessor computer system," in *Proc. SPIE Symp.,* 1981, pp. 242–248.

[15] L. Snyder, "Type architecture, shared memory and the corollary of modest potential," *Ann. Rev. Comput. Science,* vol. 1, pp. 289–317, 1986.

[16] L. Snyder, "Applications of the 'phase abstractions' for portable and scalable parallel programming," in *Proc. ICASE Workshop,*1990, to be published.

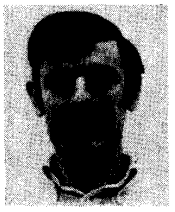
[17] L. G. Valiant, "Parallelism in comparison problems," *SIAM J. Comput.,* vol. 4, no. 3, pp. 348–355, 1975.

[18] L. G. Valiant, "A bridging model for parallel computation," *CACM,* vol. 33, no. 8, pp. 103–111, 1990.



Richard Anderson received the B.A. degree in mathematics from Reed College, Portland, OR, in 1981, and the Ph.D. degree in computer science from Stanford University, Stanford, CA in 1986.

He spent a post-doctoral year at the Mathematical Sciences Research Institute in Berkeley, CA, before becoming an Assistant Professor of computer science at the University of Washington, Seattle, WA. His current research interest is the design and implementation of parallel algorithms for shared memory multiprocessors.



Lawrence Snyder received the bachelors degree in mathematics and economics from the University of Iowa, Iowa City, and in 1973 received the Ph.D. degree from Carnegie Mellon University, Pittsburgh, PA, in computer science.

He was a visiting scholar at the University of Washington, Seattle, WA, from 1979 to 1980 and joined the faculty permanently in 1983 after serving on the faculties of Yale and Purdue. During 1987–1988 he was a visiting scholar at MIT and Harvard. His research has ranged from the design and development of a 32-bit single chip (CMOS) microprocessor, the Quarter Horse, to proofs of the undecidability of properties of programs. He created the Configurable Highly Parallel (CHiP) architecture, the Poker Parallel Programming Environment and is the coinventor of Chaotic Routing. He is a codeveloper of the Take/Grant Security Model and the cocreator of several new algorithms and data structures. Following the completing of the Blue CHiP Project he is now Principal Investigator for the Orca Project and Chief Scientist of NWLIS. He is an associate editor of the *Journal of Computer and Systems Sciences* and parallel systems editor of the *Journal of the ACM*. In 1989, he was program chair for the first Symposium on Parallel Algorithms and Architectures.