Data Locality On Shared Memory Computers Under Two Programming Models

Ton A. Ngo   Lawrence Snyder
Dept. of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

Data locality is a well-recognized requirement for the development of any parallel application, but the emphasis on data locality varies with the programming model. In this paper, we examine two models at the extreme ends of the spectrum with respect to one general class of parallel machines. We present experiments on shared-memory machines that indicate that programs written in the nonshared-memory programming model generally offer better data locality and thus better performance. We study the LU decomposition problem and a Molecular Dynamics simulation on five shared-memory machines with widely differing architectures, and analyze the effect of the programming models on data locality. The comparison is done through a simple analytical model and measurements on the machines.

1 Introduction

Because of the wide variety of parallel machines, perhaps the single most important issue in the development of parallel software is program portability. Such variety exists because given the current technology, no single architecture can claim a clear superiority: while bus-based machines offer shared-memory efficiently and economically, large scale machines are only viable with nonshared-memory. On the other hand, the intermediate range consists of both shared and nonshared-memory machines.

A portable program by definition can execute correctly on a new machine with little programming effort. However, such requirements are not sufficient. To be meaningful, a portable program must also perform well across the spectrum of machines: the program must not only run, but must also run well. To achieve scalable performance in a parallel program, a critical factor is data locality. Although locality may have little effect on a small scale machine that presents a uniform access time to all memory, it can be critical on intermediate and large scale machines which by necessity have distributed memory with different access latencies.

The memory model intrinsic to a parallel language has a great influence on the performance of programs written in this language because it dictates how the data locality that exists in the algorithm can be exploited on a parallel machine. While exploiting data locality can be done at many different levels, this paper focuses on how programming models affect the performance of parallel programs through data locality.

One's interest in the effects of data locality will likely differ depending on whether one is predisposed to a shared or a nonshared memory model of parallel computation. From the shared memory model viewpoint, a programmer might ask, "How much is lost by ignoring locality?" Here the assumption is that although physical (shared memory) machines present a shared memory to the user, the mechanisms that hide the fact that the memory is actually distributed differ in their performance. When they work well, there should be no loss, and when they work poorly, the loss will be reflected by the degree to which the nonlocal data references could have been made local had locality not been ignored. Conversely, from the nonshared memory model viewpoint one might wonder, "How much is lost from the overhead introduced to exploit locality?" Here the assumption is that to exploit locality, the program performs explicit data movement operations, e.g. sends and receives, that incur costs such as buffer management which do not exist in a truly shared memory architecture. The benefits of a nonshared memory program will be realized only when the gain from referencing data locally instead of globally is sufficient to offset the overhead.

Thus, the question in both cases reduces to determining the benefit of exploiting locality on shared memory machines. It is widely hoped that compilers will play a major role on both sides of this question: finding locality for the shared memory programs or eliminating overheads for the nonshared memory programs. Accordingly, measuring the benefit of exploiting locality can be interpreted as quantifying the potential payoff of these compiler optimizations. If the benefits are small, compiler writers can ignore finding locality, and concentrate on removing overhead. If however, the benefits are large, then finding locality becomes crucial, while removing overhead can be de-emphasized.

To understand the effect of programming models on data locality, we compare a shared memory and a nonshared memory version of two applications, LU Decomposition and Molecular Dynamic simulation. The data reference pattern of each application is used to construct a simple analytical model of the parallel execution to predict the behavior. The performance of the programs is also measured on five shared memory machines with widely differing memory organization. Care is taken to ensure that the scalar computation is the same between the shared memory and nonshared memory versions so that the difference due to data locality can be isolated.

The remainder of the paper is organized as follows. Section 2 defines the scope of the problem and section 3 describes the methodology of the experiments, including the machines and the implementation. Sections 4 and 5 describe in detail each of the two applications and the results. Conclusions are found in section 6.

2 Problem Formulation

2.1 Scope

We begin by defining the two general classes of parallel programs and parallel computers using the following short-hand notation:

The memory organization visible to the program is classified as s for shared and ns for nonshared.
P_x is a program written using the memory model x.
C_x is a parallel computer with x memory organization.
The time to execute P_x on C_x is denoted $P_x|C_x$.
The compiling and runtime support to run programs of model x on computers of type y is $Sim_{x \rightarrow y}$.

Example of $Sim_{s \rightarrow ns}$ includes Shared Virtual Memory [10], while $Sim_{ns \rightarrow s}$ can be trivially implemented by emulating the send/receive operations using the shared memory. With such a classification of programs and computers, we have the following combinations:

	computer	
program	shared	nonshared
shared	$P_s\|C_s$	$P_s\|C_{ns}$
nonshared	$P_{ns}\|C_s$	$P_{ns}\|C_{ns}$

Clearly, P_{ns} can execute on both nonshared and shared memory parallel machines (C_{ns} and C_s) because in the first case the models of the program and the machine match, and in the second case $Sim_{ns \rightarrow s}$ can be implemented easily.

Since our focus is on the class of shared-memory computer C_s, we are interested in the difference in performance between $Sim_{ns \rightarrow s}(P_{ns})$ and P_s; we represent this difference by the factor α:

$$Sim_{ns \rightarrow s}(P_{ns})|C_s = (\alpha)P_s|C_s \tag{1}$$

The value of α depends on whether P_{ns} offers any advantage against the overhead of $Sim_{ns \rightarrow s}$. We hypothesize that this advantage derives from the high degree of data locality inherent in P_{ns}. The extent of this tradeoff depends on how critical data locality is to the particular shared memory machine C_s. Since the importance of data locality is a function of the memory hierarchy, the ratio of access time between the levels of memory can serve as an indicator of the success of P_{ns} on the C_s. In other words, if the memory hierarchy is relatively flat, we expect $\alpha \geq 1$ because it is difficult to profit from the locality of the nonshared memory model, but if there is a substantial hierarchy, the data locality in the nonshared memory model can reduce the expense of global references and we can expect $\alpha < 1$.

To gain an understanding of this factor α, we will compare two versions of programs for the same problem, P_s and P_{ns}, running on a shared memory machine. By ensuring that the programs perform the same scalar operations, we can attribute the differences in performance to differences in the memory model.

2.2 Related work

P_s and P_{ns} have also been compared in similar and different contexts.

Lin and Snyder [11] made similar comparison of P_s and P_{ns} on several shared-memory machines using the Jacobi iteration and matrix multiplication: they found that P_{ns} can actually outperform P_s.

Leblanc [8] also compared $P_s|C_s$ and $Sim_{ns\to s}(P_{ns})|C_s$ using Gaussian Elimination (without pivoting) on the BBN Butterfly. He observed that the model should be chosen based on the nature of the application, and that the shared model may encourage too much communication.

The converse to the problem we are considering, $Sim_{s\to ns}$, has also attracted recent interest. An example is the Shared Virtual Memory system proposed by Li and Hudak [10], in which the operating system maintains a consistent cache of memory pages to create an illusion of shared memory in a distributed machine environment. More specifically, Priol and Lahjomri [12] developed a Shared Virtual Memory system on the iPSC/2 and compared $Sim_{s\to ns}(P_s)$ in this simulated environment against the native P_{ns}; they found that the P_s tends to have difficulty with the granularity of sharing.

Byrd and Delagi [3] used a model of network access cost model to compare $P_s|C_s$ and $P_{ns}|C_{ns}$; they found that $P_s|C_s$ is much more susceptible to high network latency, which is a likely characteristic of large scale machines.

Anderson and Snyder [1] analyzed P_s and P_{ns} of several common algorithms and found that the shared memory model tends to give overly optimistic performance prediction and, more importantly, can lead to the use of suboptimal algorithms.

3 Methodology

3.1 The Machines

Machine	Sequent	CSRD Cedar	Butterfly	Kendall Square	Stanford DASH
model	Symmetry A	Cedar	TC2000	KSR-1	DASH
nodes	20	32	128	32	48
processors	Intel 80286	MC 68020	MC 88100	custom	MIPS R3000
I cache	combined	16 Kb/node	32 Kb/node	256 Kb/node	64Kb/node
D cache	64 Kb/node	128 Kb/cluster	16 Kb/node	256 Kb/node	(64Kb+256Kb)/node 128 Kb/cluster
local mem	0	32 Mb/cluster	16 Mb/node	32 Mb/node	14 Mb/node
global mem	32 Mb	256 Mb	2 Gb*	1 Gb*	168 Mb*
network	bus-based	omega	butterfly	ring	mesh
access ratio	1	1:4.5	1:3.7:12.7	1:10:75:285	1:4.5:8:26:33

Table 1: Machine characteristics
(* sum of all local memories)

The shared-memory machines used in the experiment are the Sequent Symmetry, the BBN Butterfly, the Cedar at CSRD, Illinois, the Kendall Square Research KSR-1, and the DASH at Stanford. They represent a wide range of memory hierarchies from uniform access (Sequent) to non-uniform access (Cedar, Butterfly, KSR-1, and DASH). The Sequent, KSR-1 and DASH employ hardware coherent caches while the Cedar and the Butterfly do not. Figure 1 and Table 1 show the general organization and some relevant characteristics of each machine.

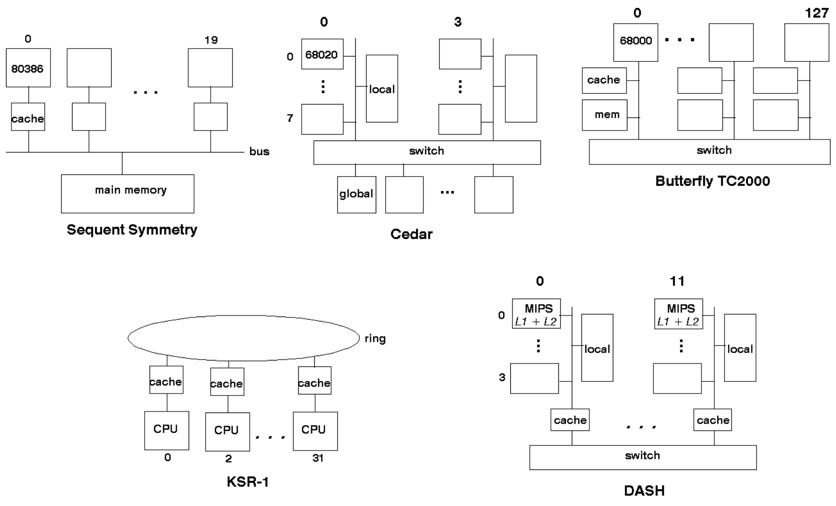

Figure 1: Machine memory hierarchy

The Sequent Symmetry is a small scale bus-based machine. Because of the low speed of the processors, the bus is able to support 20 nodes. Cache coherency is maintained by bus-snooping using a modified Illinois protocol. Since the main memory resides on the bus, the access time is the same from any processor cache, and since there is no control over the cache, each processor will only see a certain average access time. Because of these characteristics, the Sequent bears the closest resemblance to the PRAM model when compared to the other machines.

The Cedar has a cluster architecture: each bus-based cluster contains 8 processors sharing a local memory and the clusters are connected to the global memories through a switch. Therefore there are two levels of memory hierarchy. Note that the intra-cluster communication can be done at the cluster memory level, but inter-cluster communication must use the global memory.

In the Butterfly TC2000, each processor node contains a cache and a local memory. Each node is connected to the local memories of all other nodes through a multistage interconnection network (MIN); the global memory thus consists of the aggregate of all local memories. There are three levels of memory hierarchy: the cache, the local memory, and the remote memory. Although memory locations can be assigned a variety of cache attributes, the two default and most commonly used attributes are cacheable local and non-cacheable global, which represent only two levels of memory hierarchy. Since there is no hardware coherency mechanism, the machine provides various cache invalidation functions to support software caching. Finally, to avoid hot spots in this memory organization, the global address space is interleaved across all nodes.

The KSR-1 employs an AllCache architecture [7]: instead of a main memory, each node possesses a large cache that is kept coherent with all other caches through a directory-based scheme. In addition to the main cache, there are also instruction and data subcaches on each node. The nodes are connected in a hierarchy of rings; therefore, there are multiple levels of memory hierarchy, beginning with the subcache, to the local cache, the caches within the same ring, the caches within the next ring level, and so on. Because the unit of communication on the ring is a 128 byte cache line, the granularity of shared data is an important issue. While the architecture provides a combining mechanism by servicing a cache miss at the lowest level of the ring hierarchy, the machine used in the experiment only contains one ring; therefore this combining effect is not visible.

The Stanford DASH also uses a directory scheme for cache coherency [9]; however, it is organized as a collection of bus-based clusters connected in a mesh topology. Each cluster contains 4 processor nodes, a cluster memory and a cluster cache. Each node in turn contains a level 1 and a level 2 cache, with coherency maintained at the level 2 cache through bus snooping. There are then at least 5 levels of memory hierarchy: level 1 cache, level 2 cache, cluster memory, remote cluster memory, and remote dirty cluster memory. The cluster cache which resides in the interface between a cluster and the other clusters combines requests to the same address. Since there is a home node for each memory location, the system also provides functions for data placement.

3.2 Implementing the programming models

Since both applications contain a high degree of data parallelism, all programs are written in the SPMD style.

In P_s, data reside in the global memory and the standard lock and barrier are used for synchronization.

Data in P_{ns} are placed in the level of memory closest to the processor. On machines with coherent caches, this effect is achieved automatically through the data reference pattern: a processor only accesses its data section, allowing it to migrate to the highest level of the cache. The send and receive are emulated with block copy, and simple one-reader/one-writer ports implement some connection topology (binary tree, mesh, etc.). Broadcasting is done through a binary tree. The buffers used for communication are placed in the global memory. The trade-off in P_{ns} will then be between the communication overhead and the use of the faster local memory. Note that synchronization in the nonshared programs is implicit in the communication since blocking sends and receives are used.

The performance is measured only for the useful computation; the initialization costs are not included.

4 LU Decomposition

The following two sections describe the two sets of experiments. The problem and the algorithm are first described, followed by an analysis of the data locality in the P_s and P_{ns} versions. Results and discussion follow.

4.1 The problem

Typical of matrix problems, LU decomposition is a numerical method for solving large systems of linear equations. Given the system of equations:

$$Ax = b$$

where A is the coefficient matrix and b is the constant vector, A is decomposed into a lower and upper triangular matrices L and U:

$$LUx = b$$

Letting Ux = y, we can solve directly for y by forward-elimination

$$Ly = b$$

and then for x by backward-substitution.

$$Ux = y$$

The sequential algorithm consists of iterating over the diagonal of the matrix, each iteration consisting of two steps: partial pivoting (line 2) and row update (lines 3:6).

LU_{seq}

```
(1)   for (k=0; k<row; k++)
(2)     partial pivot
(3)     for (i=k+1; i<row; i++)
(4)       $A_{ik} = A_{ik}/A_{kk}$;
(5)       for (j=k+1; j<col; j++)
(6)         $A_{ij} = A_{ij} - A_{ik} * A_{kj}$
```

The partial pivoting step is necessary for numerical stability in the division step due to the limited precision in digital computer: it involves searching the pivot column k for the largest element and then swapping that row with the pivot row k. With a complexity of $O(n^2)$ compared to the $O(n^3)$ of the row update step, this step only constitutes a minor part of the computation; however, it introduces additional serialization into the algorithm.

4.2 The parallel algorithms

Since LU decomposition is a well studied problem, optimized parallel algorithms are widely available in the literature [2, 6, 13].

The computations in the row update step for each iteration are independent and can be parallelized easily. The partial pivoting step is more difficult to parallelize effectively because the parallelism available is small compared to the communication/synchronization required for parallelization. Therefore, the optimized algorithm employs a pipelining technique: during the current iteration, a processor completely factorizes a set of t columns and saves the transformation, while the remaining processors update the submatrix using the saved transformation from the previous iteration. The value of t controls the granularity of the task partitioning and is chosen to best balance the workload and the communication/synchronization overhead.

The pseudo code for the optimized parallel LU decomposition algorithm is shown below.

```
        $LU_s$
(1)       P0 factors col[0:r]
(2)       for (k=r; k<row; k+=t)
(3)         switch transformation buffer
(4)         if (own_column(k))
(5)           updates col[k:k+t] using transformation (k-t)
(6)           factors col[k:k+t] saving transformation k
(7)         else
(8)           updates col[k+t:col-1] using transformation (k-t)
```

```
        $LU_{ns}$
(1)       P0 factors col[0:t]
(2)       and broadcasts the transformation
(3)       for (k=r; k<row; k+=t)
(4)         if (own_column(k))
(5)           updates col[k:k+t] using transformation (k-t)
(6)           factors col[k:k+t] saving transformation k
(7)           broadcasts the transformation k
(8)         else
(9)           updates col[k+t:col-1] using transformation (k-t)
```

LU_s and LU_{ns} thus implement the same algorithm. The differences are:

1. Any processors can update any portion of the matrix in LU_s, while the matrix is partitioned statically by columns in LU_{ns}. Because the iteration traverses the diagonal of the matrix, partitioning the columns by blocks will result in a poor load balance in LU_{ns}: some processors will run out of work once k has passed their sections. To alleviate this problem, sets of r columns are assigned to the processor in an interleaved fashion (cyclic).

2. LU_s requires barrier synchronizations before and after switching the transformation buffer in line 3. LU_{ns} requires broadcasting the newly computed transformation buffer to all processors in lines 2 and 7.

On the Cedar and the Butterfly where there exists a local memory but no coherent cache, we improve the data locality of LU_s by performing software caching in the innermost loop. On Cedar where there is a cluster level memory, we also optimize the communication in LU_{ns} by using the cluster memory for intra-cluster messages and the global memory for inter-cluster messages.

4.3 Volume of data references

Since LU decomposition is a static algorithm, the volume of scalar computation, and thus references of the matrix data, should be nearly identical for a sequential, shared memory, and nonshared-memory versions, discounting small variations due to variable reuse and the references of global parameters. The total number of processor cycles to compute and perform this volume of references represents a lower bound on the execution time of the program. Since the data placement is static in the memory hierarchy and the ratio of access times to each level of memory is known, we can derive an indicator of the relative performance of P_s and P_{ns}. We assume the synchronization cost is small and the load balance is perfect. For the following analysis, we only assume a local and global level of memory and that read and write have the same latency. The matrix is also assumed to be square, i.e. n=column=row.

Referring to the LU_{seq} algorithm above, in each outermost iteration k the partial pivot step consists of scanning a column (1 read) and swapping two rows (2 reads + 2 writes) beginning from the diagonal element. The number of references is:

$$(3r + 2w) * (n - k)$$

The row update step consists of dividing the column by the pivot element (2 reads + 1 write) and adjusting the rows (3 reads + 1 write). The number of references is:

$$((2r + 1w) + (3r + 1w) * (n - k - 1)) * (n - k - 1)$$

Summing up over the diagonal iterations, we obtain the volume of references to the matrix:

$$\sum_{k=0}^{row-1} (3r + 2w) * (n - k) + ((2r + 1w) + (3r + 1w) * (n - k - 1)) * (n - k - 1)$$

Simplifying and substituting the summations with the equivalent polynomials, we obtain the expression:

$$n^3(r + \frac{1}{3}w) + n^2(r + w) + n(r + \frac{2}{3}w)$$

In addition to accessing the matrix data, the transformation data in P_{ns} needs to be broadcast to the worker processors. This is done through a binary tree: the buffer is sent to the root processor and is propagated down the tree. Each message requires a pair of sends and receives; each send and receive operation in turn involves a local read and a global write, or a global read and a local write, respectively. A message transmission then requires ((2 global + 2 local) * size) references. The elapsed time for propagating through the binary tree requires the equivalence of $(\log p + 1)$ transmissions. The size of the transformation buffer in each k iteration is (n-k)*t, where t is the parameter controlling the task granularity. The total time for all tree broadcasts is then:

$$(\log p + 1) * (2global + 2local) * \sum_{k=0,k=k+t}^{n-1} (n - k) * t$$

Substituting the summation and simplifying yields:

$$(\log p + 1) * (global + local) * n * (n + t)$$

We can compute an estimate of the relative performance of P_s and P_{ns} with the assumptions:

1. The matrix references for P_s will be to the global memory and for P_{ns} to the local memory.

2. The computation and thus the matrix references are perfectly distributed among the processors.

3. P_{ns} requires the additional tree broadcast operations.

4. The references are free of contention.

Table 2 summarizes the expressions for the reference counts and the FLOP counts in part (a); part (b) shows the reference counts that represent the elapsed time for the tree broadcast; and part (c) tabulates the reference counts to the local and global memory based on the assumption that data is placed in global memory in LU_s and in local memory in LU_{ns}. From this information, we can derive a simple model for the parallel execution of LU_s and LU_{ns}:

Phases	FLOP	read	write
partial pivot	$\frac{1}{2}(n^2-n)$	$\frac{3}{2}(n^2+n)$	n^2+n
column update	$\frac{1}{6}(4n^3-3n^2-n)$	$\frac{1}{2}(2n^3-n^2-n)$	$\frac{1}{3}(n^3-n)$
Total	$\frac{2}{3}(n^3-n)$	n^3+n^2+n	$\frac{1}{3}(n^3+3n^2+2n)$

(a) Flops and reference count for LU computation phases: n = size of nxn matrix

Phases	read	write
Total	$\frac{1}{2}n(n+r)(\log p+1)(loc+glob)$	$\frac{1}{2}n(n+r)(\log p+1)(loc+glol)$

(b) Elapsed time for LU_{ns} communication in terms of reference count:
n = size of nxn matrix, p = processor number in powers of 2

Program	local read+write	global read+write
LU_s	0	$\frac{1}{3}(4n^3+6n^2+5n)$
LU_{ns}	$\frac{1}{3}(4n^3+6n^2+5n)$ $+n(n+r)(\log p+1)$	$n(n+r)(\log p+1)$

(c) LU_s and LU_{ns} references to memory hierarchy

Table 2: Volume of data references for LU

$$total\_cycles = communication\_cost + parallel\_task$$

$$parallel\_task = \frac{1}{p}(FLOP * FLOP\_cycle + global * global\_cycle + local * local\_cycle)$$

Figure 2 plots the speedup based on the number of cycles required by P_s and P_{ns} for ratios of memory hierarchy of 1:1, 1:2 and 1:4, with n=512, t=4, and FLOP_cycle=1.

Our simple model predicts that P_{ns} easily outperforms P_s when there is any gap between the local and global memory. Naturally, many factors are ignored, such as the load balancing, the synchronization, the network contention, the actual higher cost for emulating the communication, etc. However, if the data locality controls the first order effects and these factors are secondary, then the estimate can be qualitatively correct. In the next subsection we will look at the results measured on the five machines.

4.4 Performance results

Figures 3 and 4 show the performance and the speedup of the two versions of LU decomposition on the five machines for three different problem sizes. The speedups are based on the performance of a straightforward sequential version of LU decomposition.

Referring to the predicted speedup curves in Figure 2 and the local:global access ratio for each machine in Table 1, we find that the results match the model prediction well.

We first consider the results from the Sequent, the Cedar and the Butterfly since the ratios of these machines are more precise. The ratio of 1:1 on the Sequent gives LU_s a slightly better performance than LU_{ns} for all problem sizes. On the Cedar where the ratio is 1:4.5, LU_{ns} offers the better performance. Although the Butterfly has three levels of memory hierarchy, the program only uses two levels (the default cacheable local and noncacheable global attributes); therefore the effective ratio is 1:12.7. This large gap in access latency translates directly into a large gap in the performance between LU_s and LU_{ns}; the steep hierarchy on the Butterfly gives $Sim_{ns\rightarrow s}(P_{ns})$ an advantage that far outweighs the simulation overhead. We note a number of program optimizations on the Cedar and the Butterfly:

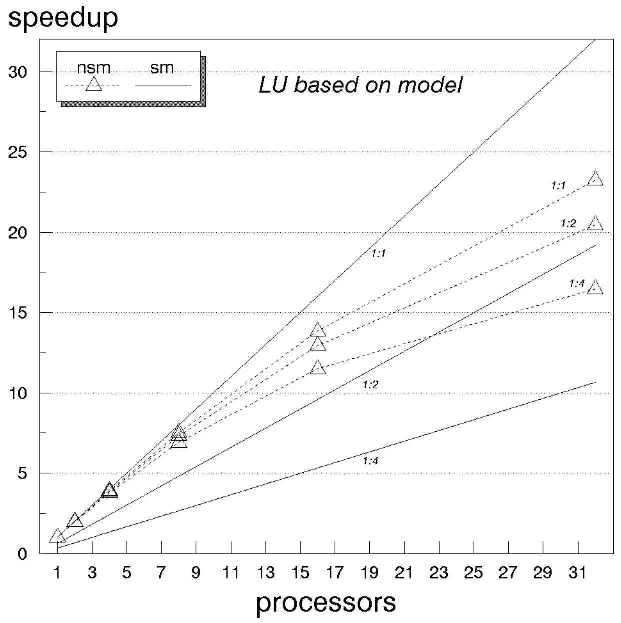

Figure 2: LU speedup based on model: n=512, r=4

1. For LU_s on Cedar and Butterfly, software caching is performed in the innermost loop by copying a column into local memory for performing the column update; this prevents repeated access of the same column from global memory while updating using the saved transformation.

2. For LU_{ns} on Cedar, intra-cluster communication uses the cluster memory, while inter-cluster communication uses the global memory.

Since the Butterfly cache is controlled by software, more advanced caching techniques may improve the performance of LU_s, but it seems difficult to recapture the large performance gap. It thus appears that $Sim_{ns \to s}(P_{ns})$ matches well a large scale shared memory machine with *private* per processor cache. Since the Sequent has a coherent cache while the Butterfly does not, a natural question is whether the behavior found on the Sequent would be observed on the Butterfly if a coherent cache were implemented. A coherent cache improves the performance of both P_s and P_{ns}: it will reduce the global data references in P_s and effectively eliminate the spin-lock traffic in P_{ns} communication.

To search for an answer, we consider the results from the KSR-1 and the DASH, two machines that employ large scale coherent caches. Given the very large ratios for these machines, our simple model predicts that the shared-memory version would not yield any speedup. On the other hand, if we extrapolate from the performance on the Sequent, the coherent cache would be able to hide the memory hierarchy and to present a more uniform access to memory, thus giving LU_s a slight advantage over LU_{ns}. The results show that LU_s achieves a reasonable speedup on both machines, but that it trails LU_{ns} by a significant amount in both actual performance and speedup. Clearly, the actual behavior lies in the middle ground between our two conflicting predictions: by reducing the number of remote accesses, the coherent cache is very effective in reducing the gap in the memory hierarchy, but not to the degree where data locality is rendered unnecessary. The difference in fetching from local and remote memory by the cache is visible in the program performance.

With respect to scaling with problem size, the relative difference in performance between LU_s and LU_{ns} is maintained in every case, and the speedup improves as the problem size increases. Not surprisingly, this behavior reflects the dominant computation cost relative to the cost of communication or memory references, $O(n^3)$ versus $O(n^2)$ for LU.

With respect to scaling with the number of processors, we observe that while the speedup and performance appear reasonable for up to 32 processors on Cedar, KSR-1 and DASH, neither LU_s nor LU_{ns} achieves any speedup beyond 32 processors on the Butterfly when the number of processors approaches 100. A partial explanation for the poor scaling in LU_{ns} can be found in our simple implementation of the communication: (1) all messages are point to point, and (2) a processor spin-locks on a global variable while waiting for an empty write port or a full read port. Each of these factors constitutes a component in the overall communication cost which increases with the number of processors. It is also possible that the problem sizes used are not large enough. Karp and Boris [16] showed that speedup for LU on the Butterfly can be maintained if the problem size is increased proportionally with the number of processors. However, such problem size would quickly exceed the storage capacity of the machine.

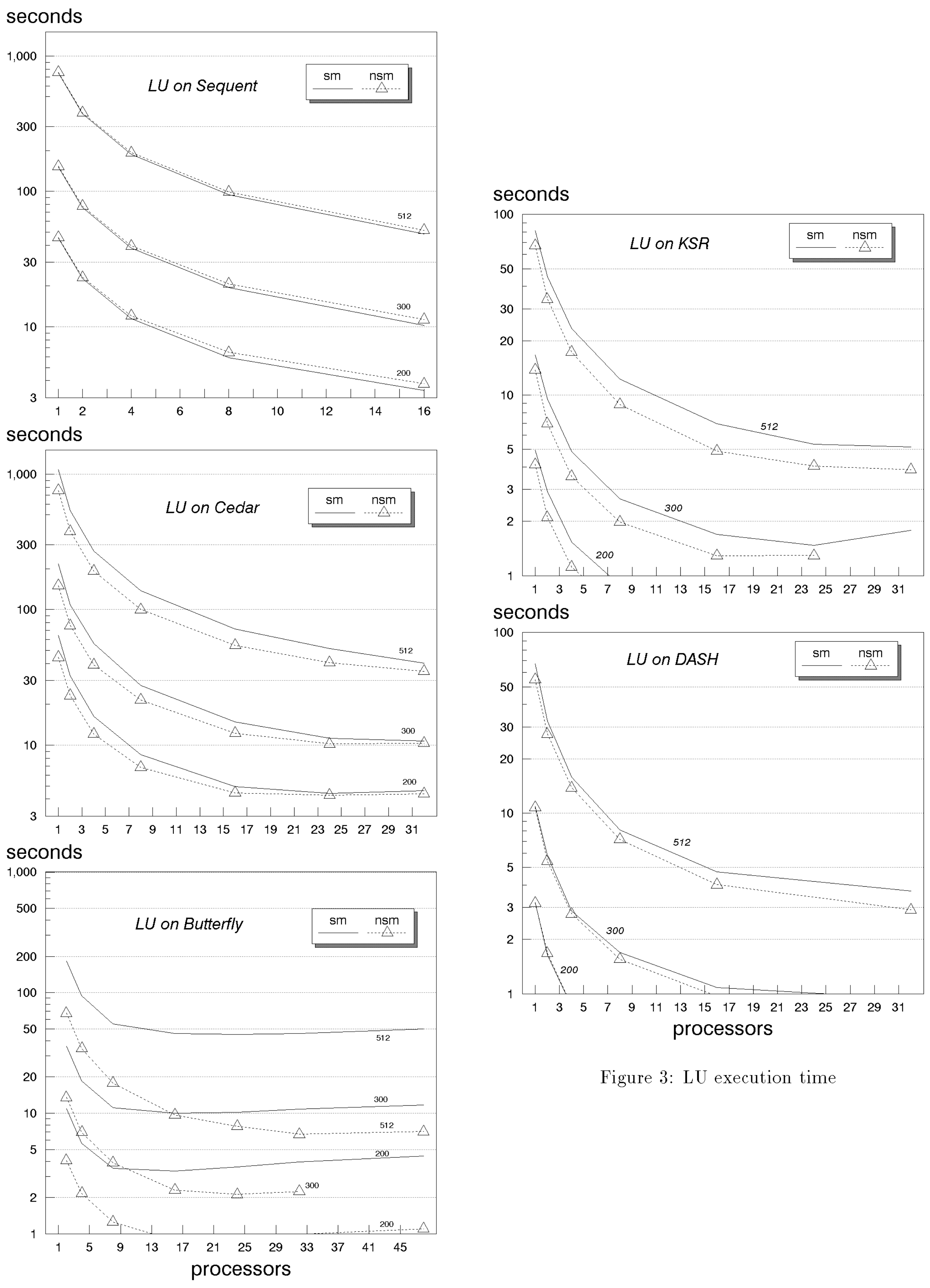

Figure 3: LU execution time

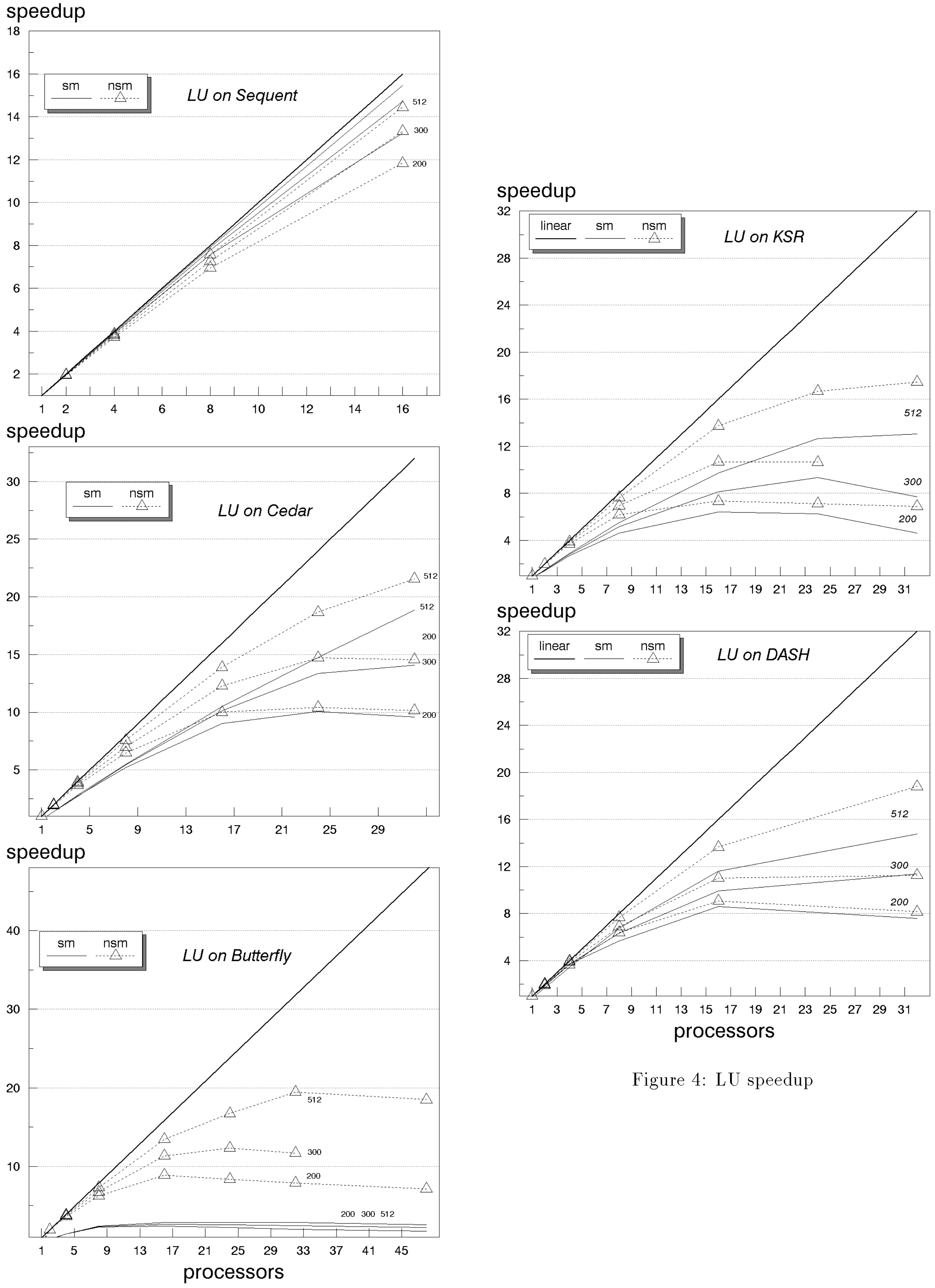

Figure 4: LU speedup

5 Molecular Dynamics Simulation

5.1 The problem

The WATER benchmark from the Stanford SPLASH suite is a simulation of several hundred water molecules in a cubical box in the liquid state at room temperature [14]. The program is representative of the n-body problem, in which each body interacts in certain ways with all other bodies in the system. In this case, the simulation computes the forces and potentials among the water molecules to predict various static and dynamic properties of water. To compute all pairwise interactions, a processor in any partitioning scheme will have to access the data in all sections, giving an initial appearance of poor data locality in the problem. However, this particular version, WATER, achieves a good speedup thanks to a favorable ratio of computation to communication, as will be described in the following subsections.

5.2 The algorithm

The program was manually parallelized from a sequential version, the MDG benchmark in the Perfect Club suite. After initializing the displacements and velocities, the algorithm consists of iterating over a large number of time steps until the system converges to a steady state. Each time step consists of seven computation phases separated by barrier synchronizations:

1. predict new values for displacement and the derivatives.
2. compute the intramolecular forces between the atoms of each molecule.
3. compute the intermolecular forces between the atoms of each pair of molecules.
4. correct the predicted values for forces.
5. handle the boundary conditions by moving the molecules back into the box if they are out of the box.
6. compute the kinetic energy in each of the three spatial dimensions.
7. compute the potential energy as the sum of the intermolecular and intramolecular potentials.

The computation complexity is $O(n^2)$, but the actual number of pairwise interactions to be computed is reduced by defining a cutoff radius of half the box dimension.

The $WATER_s$ version was ported to the various machines strictly by substituting the parallel macros for lock and barrier synchronization.

$WATER_{ns}$ is derived from $WATER_s$ by replacing the shared data structures with distributed structures and performing a global update at each point where (1) each partition has to access all other partitions to compute the pairwise interactions, or (2) a global sum has to be computed and broadcasted to all partitions. Communicating through a ring topology, each process computes the interactions within its partition, then sends a copy on a complete trip around the ring; as a partition is received, the process updates both its partition and the traveling partition. When the modified partition returns to its source, it is merged into the original partition. Computing the global sum is done similarly by sending the partial sum around the complete ring.

In both versions, the processor workload is statically assigned and load balancing is not considered a problem due to the uniform distribution of the input data. $WATER_s$ and $WATER_{ns}$ thus perform the identical computation; the only differences are in the placement of the data and the resulting communication.

The algorithm has a computation complexity of $O(n^2)$ and a communication complexity of $O(n)$, which will be described in more detail in the next subsection.

5.3 Volume of data references

As with the LU experiment, we create a simple model of the parallel execution of WATER based on the FLOPS and data reference counts.

The computation phases are listed in order in Table 3 part (a) together with the count of floating point ops, reads and writes for each time step; the counts are obtained manually from the program. To account for the cutoff range of half the box length, the molecule distribution is assumed to be uniform and those counts which are dependent on the range are divided in half. Part (b) shows the elapsed time of the ring communication in $WATER_{ns}$ in terms of reference counts; note that only 4 phases actually require communication. In part (c), the reference counts to local and global memory are tabulated based on the assumption that data is placed in global memory in $WATER_s$ and in local memory in $WATER_{ns}$; the counts include those references for emulating the communication in $WATER_{ns}$.

Phases	FLOP	read	write
predict val	432n	243n	54n
intra force	42n+223	24n	3n+3
inter force	$163n^2 + 9n$	$46n^2 + 9n$	$4n^2 + 9n$
correct val	135n	81n	63n
boundary	9n	9n	9n
kinetic	24n+3	18n + 3	3
potential	$122n^2 + 128n + 3$	$42n^2 + 33n + 3$	3n+3
Total	$285n^2 + 779n + 229$	$88n^2 + 417n + 6$	$4n^2 + 141n + 9$

(a) WATER computation phases: n = number of molecules

Phases	read	write
intra force	3 (loc+glob)	3 (loc+glob)
inter force	(84n+3) (loc+glob)	(84n+3) (loc+glob)
kinetic	3 (loc+glob)	3 (loc+glob)
potential	(84n+3) (loc+glob)	(84n+3) (loc+glob)
Total	(168n+12) (loc+glob)	(168n+12) (loc+glob)

(b) Elapsed time for $WATER_{ns}$ ring communication:
n = number of molecules; loc, glob = local, global access

Program	local read+write	global read+write
$WATER_s$	0	$92n^2 + 558n + 15$
$WATER_{ns}$	$92n^2 + 894n + 39$	336n+24

(c) $WATER_s$ and $WATER_{ns}$ reference to memory hierarchy

Table 3: Volume of data references for WATER

As with the LU experiment, a simple model can be derived as:

$$total\_cycle = communication\_cost + parallel\_task$$

$$parallel\_task = \frac{1}{p}(FLOP * FLOP\_cycle + global * global\_cycle + local * local\_cycle)$$

Figure 5 shows the speedup for $WATER_s$ and $WATER_{ns}$ based on the number of cycles required; several ratios of local to global are shown, while the cycle per FLOP is set to 1. The model predicts that for a local:global ratio of 1:1, $WATER_s$ has the better speedup, but as the ratio increases, $WATER_s$'s speedup degrades quickly and falls below $WATER_{ns}$. Note also that in general the speedups for both versions are better than those for LU; the reason is evident in the large FLOP count relative to the reference count, indicating a more abundant amount of parallelism.

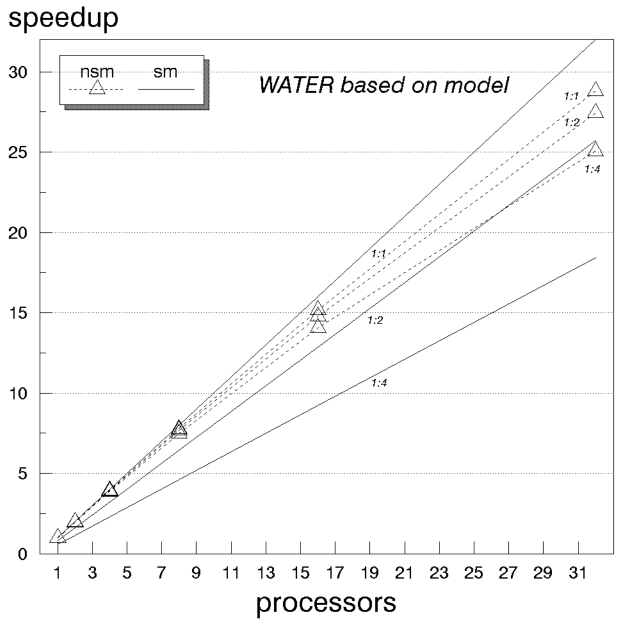

Figure 5: WATER speedup based on model

5.4 Performance results

Figures 6 and 7 show the performance of the two versions of WATER on the five machines for three different problem sizes. The speedup is based on the better uniprocessor performances of the two versions.

As evident in Table 3, WATER differs from LU in that the computation dominates the memory accesses and the memory reads dominate the memory writes, although all have the same asymptotic complexity.

On the Sequent where the local:global ratio is 1:1, $WATER_s$ has a near linear speedup while $WATER_{ns}$ is only slightly behind. On Cedar, both versions give virtually the same performance and speedup, although the model predicts that $WATER_{ns}$ would be faster. It is possible that there are some other factors involved that are not included in the model.

On the Butterfly, neither version achieves a speedup beyond 20; indeed, $WATER_{ns}$'s performance degrades below $WATER_s$ when the number of processors approaches 100. This behavior is consistent with our model although a graph for this configuration was not shown: as the number of processor increases, the parallel task decreases while the communication cost increases as in LU or at best remains constant as in WATER; therefore, at a certain point, the communication cost in $WATER_{ns}$ can no longer be hidden by the saving in using the local memory and the performance suffers.

On the KSR-1, the performance and speedup of both versions are high and nearly identical. On the DASH, the results of WATER actually differ from that of LU: although both shared and nonshared-memory versions have the same performance for small numbers of processors, $WATER_{ns}$ has the worst performance when the number of processors is large.

Model prediction for the DASH and KSR-1 is uncertain because of the combination of the cache and the steep memory hierarchy behind the cache. The effectiveness of the cache depends on the characteristic of the application and the overhead for maintaining consistency. In this case, the high ratio of read to write access implies that the degree of data sharing is relatively small and that the cache hit rate is high. This would decrease the significance of the memory hierarchy, allowing the cache to present a more effective model of uniform shared-memory. In other words, given the memory access characteristic of the WATER program, the DASH achieves an effective local:global ratio of 1:1 while the KSR-1 gives a ratio that is only slightly worse.

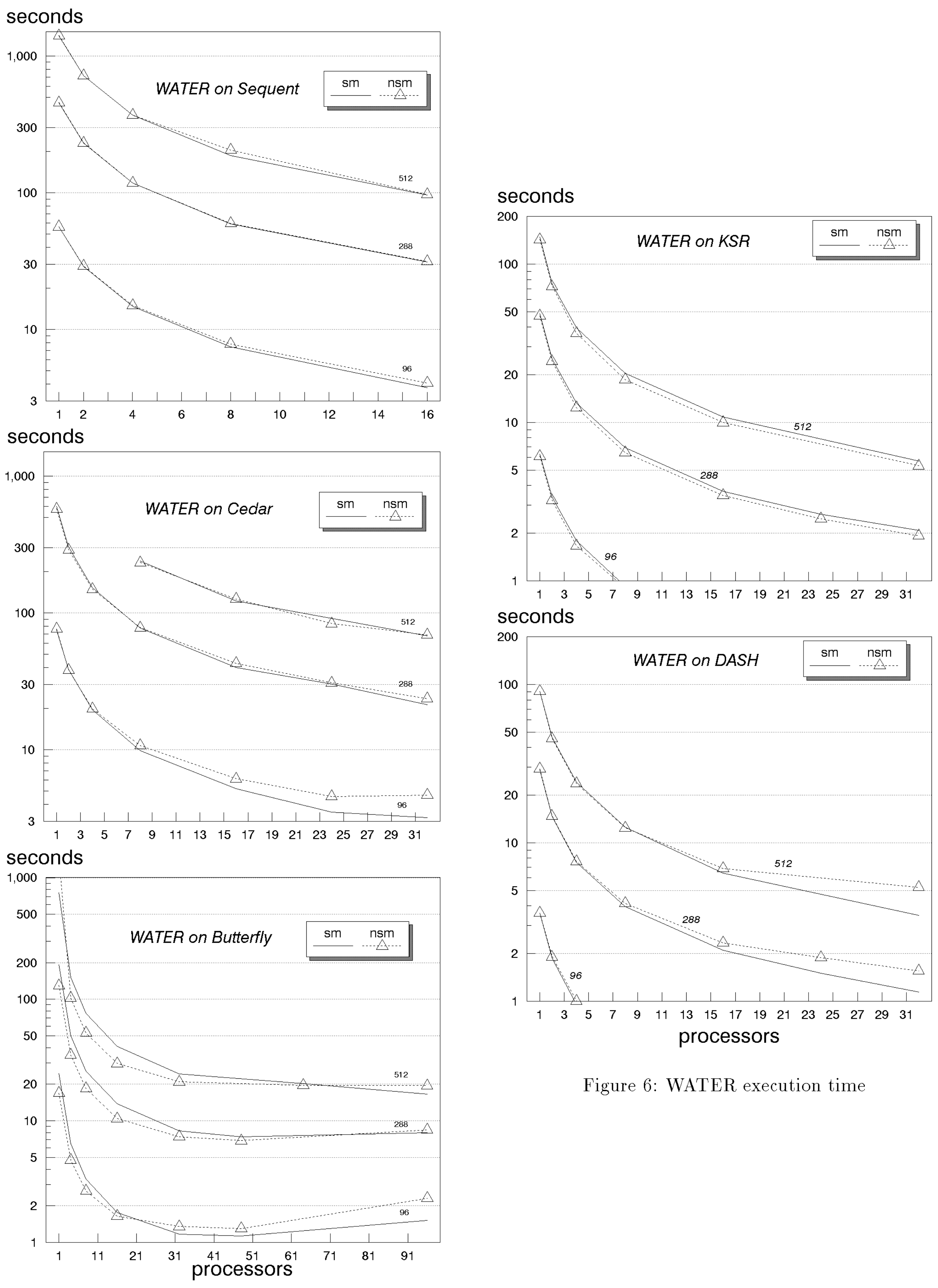

Figure 6: WATER execution time

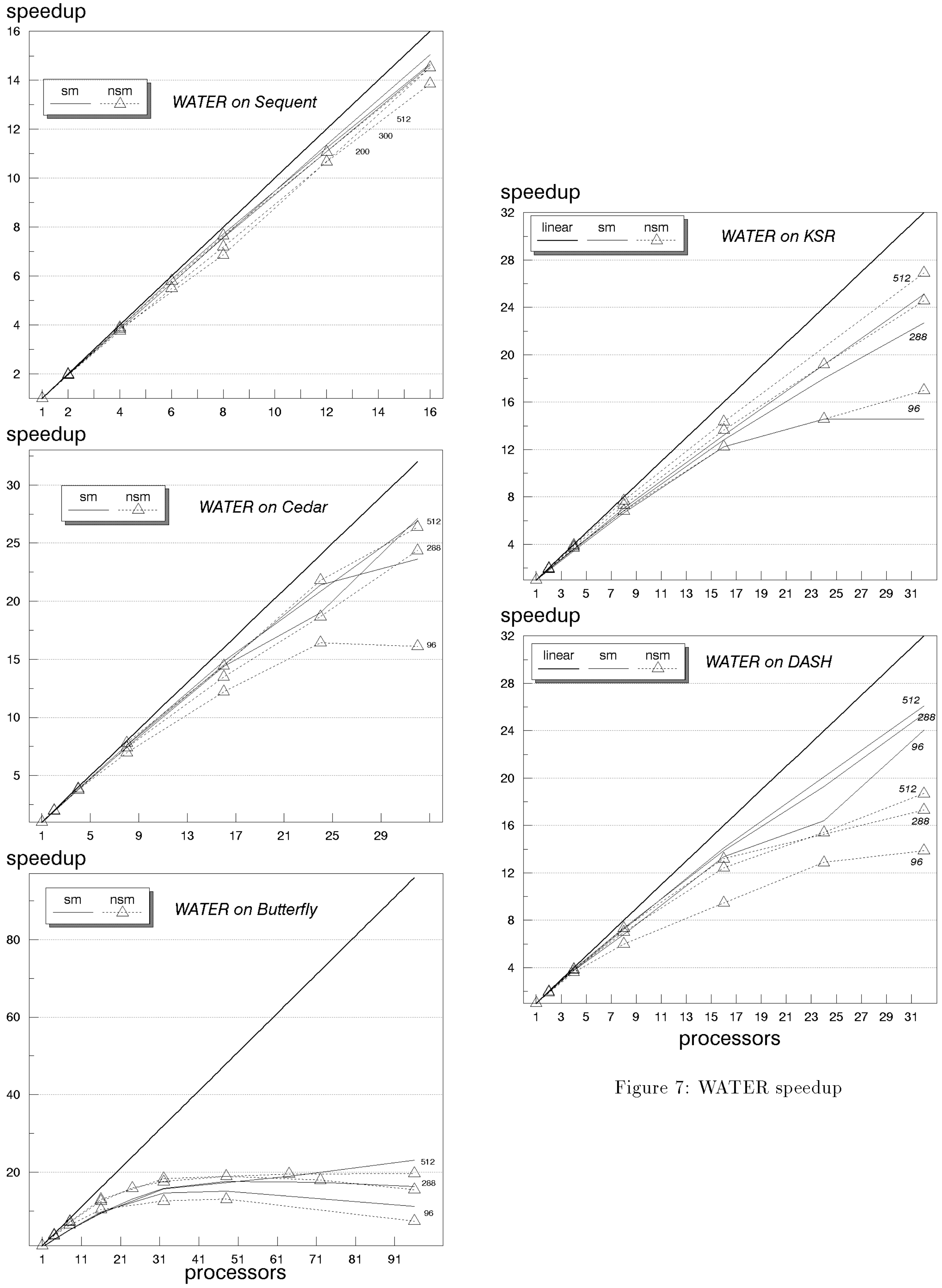

Figure 7: WATER speedup

6 Conclusion

To understand the effect of the shared and nonshared memory programming model on data locality on shared memory computers, we have studied in detail two applications on five widely differing machines and compared these models with respect to performance. We express the performance difference between the two versions as a factor α. Table 4 summarizes the range of the factor α for each application on each machine; note that the nonshared memory program has the better performance in the range $\alpha < 1$.

	LU		WATER	
machine	low	high	low	high
Sequent	1.01341	1.11902	1.00238	1.10290
Cedar	0.69997	0.96779	0.77829	1.46597
Butterfly	0.14009	0.55449	0.67901	1.52111
DASH	0.78565	1.02385	0.98151	1.73333
KSR-1	0.66854	0.87856	0.85714	1.00000

Table 4: Measured α

The results validate our hypothesis in equation (1). For shared memory machines with a non-uniform memory access time, programs written using the nonshare memory model has an advantage since it can more fully exploit the faster local memory. The benefit varies with the machine architecture and depends mainly on the effective gap between the global and local memory access time. It can be significant if the ratio between the global and local memory latency is large, for instance those found in the Butterfly, but it is otherwise small if the ratio is small or nonexistent. Although the KSR and DASH have a deep memory hierarchy with large gaps between the levels, the hardware coherent cache plays a significant role in reducing the *number* of long latency accesses and thus the effective gap in the memory hierarchy. However, the effectiveness of the coherent cache depends critically on the characteristics of the application, and data locality still plays an important role in achieving good speedup.

Although we have approached the problem from the point of view of comparing two programming models, we can also view these same results from a purely shared memory perspective: exploiting data locality is critical for performance and can be done through the machine architecture (e.g. cache) and the compiler (e.g. data placement); programming with the nonshared memory model provides an attractive approach for effectively extracting data locality in the program.

The limiting factor for a nonshared memory program is the growing communication overhead relative to the decreasing workload as the number of processors increases. One may argue that this behavior is an artifact of the manner in which the speedup is measured rather than a characteristic of the nonshared model. This argument is supported by the fact that the speedup improves uniformly with larger problem sizes; in this case, a more accurate method for measuring speedup would use the largest problem size possible for each number of processors [13]. However, the memory requirement as a function of the problem size may limit scaling up the problem size to match the number of processors: the system memory at best increases linearly with the number of processors, but the memory requirement of LU, for instance, is $O(n^2)$. Furthermore, this argument only serves the interest of obtaining the best speedup possible. In practice, a real problem specifies a certain data size and the goal is to obtain the optimal performance for this data size. Therefore, a more pragmatic conclusion is that there is a need for optimizing the communication; possible techniques include reducing the frequency and the overhead per message, and overlapping the communication with computation.

Finally, we have not considered architectures such as the Tera machine, which employ multithreading to overlap the memory access latency with the computation. In such machines, data locality can be rendered unimportant *if* there exists sufficient parallelism in the application to hide the memory latency completely. However, we can speculate that if such machine supports variable message sizes (instead of only single or double words), the nonshared model can greatly facilitate the compiler effort: since the total number of cycles to access data will be significantly reduced, the compiler can schedule a smaller number of threads to hide the latency.

Acknowledgements

We wish to thank Calvin Lin for many helpful discussions. We also express our gratitude to Eugene Brooks at the Lawrence Livermore Lab, David Padua at the University of Illinois, and the DASH group at Stanford University for granting access to the Butterfly, the Cedar and the DASH machine.

References

[1] R. Anderson and L. Snyder, "A Comparison of Shared and Nonshared Memory Models of Parallel Computation," *Proceedings of IEEE,* (1991), 79(4):480-487.

[2] C. Ashcraft, "A Taxonomy of Distributed Dense LU Factorization Methods," Engineering Computing and Analysis Technical Report ECA-TR-161, (March 1991).

[3] G. Byrd and B. Delagi, "A Performance Comparison of Shared Variables versus Message Passing," *The Third International Conference on Supercomputing,* (1988) Vol. 1, pp. 1-7.

[4] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua, "Experience in the Automatic Parallelization of Four Perfect Benchmark Programs," *The Fourth Workshop on Languages and Compilers for Parallel Computing,* (August 1991) pp. g1-g19.

[5] High Performance Fortran Forum, "HPF Language Specification version 1.0," (1993).

[6] A. Karp, "Programming for Parallelism," *Computer*, (May 1987) pp. 43-56.

[7] Kendall Square Research, "KSR Technical Summary", (1992).

[8] T. LeBlanc, "Shared-Memory versus Message-Passing in a Tightly-Coupled Multiprocessor: A Case Study," *International Conference on Parallel Processing,* (1986) pp. 463-466.

[9] D. Lenoski, et al., "The DASH Prototype: Logic Overhead and Performance," *IEEE Transactions on Parallel and Distributed Systems,* (January 1993) Vol 4, No 1, pp. 41-61.

[10] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Transactions on Computer Systems,* (November 1989) Vol 7, No 4, pp. 463-466.

[11] C. Lin and L. Snyder, "A Comparison of Programming Models for Shared Memory Multiprocessors," *Proceedings of the International Conference on Parallel Processing,* (1990), Penn State Vol. II, pp. 163-170.

[12] F. Andre and T. Priol, "Programming Distributed Memory Parallel Computers without Explicit Message Passing," *SHPCC,* (1992), pp. 90-97.

[13] Y. Robert, *The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm,* Halsted Press (1990).

[14] J. Singh, W. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared Memory," report draft, Department of Computer Science, Stanford University, (1991).

[15] L. Snyder, "Type Architecture, Shared Memory and the Corollary of Modest Potential," *Annual Review of Computer Science*, (1986), Vol. 1, Annual Review, Inc., pp. 289-318.

[16] S. Stark, and A. Beris, "LU Decomposition Optimized For A Parallel Computer With A Hierarchical Distributed Memory," *The 1991 MPCI Yearly Report: The Attack of the Killer Micros,* (March 1991), pp. 127-132.