Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors

Edward Rothberg

Intel Supercomputer Systems Division
14924 N.W. Greenbrier Parkway
Beaverton, OR 97006

Jaswinder Pal Singh and Anoop Gupta

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

The distribution of resources among processors, memory and caches is a crucial question faced by designers of large-scale parallel machines. If a machine is to solve problems with a certain data set size, should it be built with a large number of processors each with a small amount of memory, or a smaller number of processors each with a large amount of memory? How much cache memory should be provided per processor for cost-effectiveness? And how do these decisions change as larger problems are run on larger machines?

In this paper, we explore the above questions based on the characteristics of five important classes of large-scale parallel scientific applications. We first show that all the applications have a hierarchy of well-defined per-processor working sets, whose size, performance impact and scaling characteristics can help determine how large different levels of a multiprocessor's cache hierarchy should be. Then, we use these working sets together with certain other important characteristics of the applications—such as communication to computation ratios, concurrency, and load balancing behavior—to reflect upon the broader question of the granularity of processing nodes in high-performance multiprocessors.

We find that very small caches whose sizes do not increase with the problem or machine size are adequate for all but two of the application classes. Even in the two exceptions, the working sets scale quite slowly with problem size, and the cache sizes needed for problems that will be run in the foreseeable future are small. We also find that relatively fine-grained machines, with large numbers of processors and quite small amounts of memory per processor, are appropriate for all the applications.

1 Introduction

As larger multiprocessors are built, determining the appropriate distribution of resources among processors, cache and main memory becomes increasingly challenging for a designer. Small-scale, bus-based, shared-memory multiprocessors usually provide relatively large per-processor caches (several hundred Kbytes to a few Mbytes) and tens of Mbytes of physical memory per processor. These decisions make sense for small-scale machines. For example, with a small number of processors, the memory per processor must be large in order for the machine to have enough total memory to perform interesting computations. And large caches make sense for several reasons: (i) multiprogramming and the need to accommodate several applications simultaneously, (ii) the use of a shared bus interconnect and the need to reduce traffic on it, and (iii) the fact that there are only a few caches and a large amount of main memory, so that caches cost only a small fraction of the machine.

On large-scale parallel machines, many of these reasons for large caches and main memories per processor no longer necessarily hold. The desirable amounts of main memory and cache per processor are therefore not obvious. These desirable ratios are also very difficult to determine owing to the wide range of issues involved, including application characteristics, machine usage patterns, hardware cost and performance estimates, and even determining the appropriate metrics to optimize.

In this paper, we focus on one crucial input into the above design decisions: the characteristics of applications that are likely to run on high-performance multiprocessors. By studying relevant application characteristics such as memory usage, working set sizes, communication to computation ratios, concurrency and load balancing, and by examining how these characteristics scale to larger problems and machine sizes, we reflect upon the appropriate amounts of memory and cache per processor for five important classes of scientific applications. These classes are: direct equation solvers, iterative equation solvers, spectral transform methods (represented here by a Fast Fourier Transform), hierarchical N-body methods, and volume visualization (volume rendering) methods.

We divide our treatment of every application into two parts. First, we examine the working sets of the applications, which help in determining how large the levels in the machine's cache hierarchy should be to keep performance losses due to capacity misses low. We find that all the applications have a well-defined *hierarchy of working sets*, such that a cache that is large enough to hold a given working set can yield dramatic performance benefits over a cache that is slightly smaller than that working set. We also find that the working sets of all the applications are bimodally distributed, consisting of a few small working sets and one large one that usually comprises a processor's entire partition of the data set. In most cases, the working set that is critical to good performance is one of the smaller ones. In three of the applications (direct solvers, iterative solvers, and the FFT), this important working set—and hence the cache size needed for good performance—is very small and does not scale with problem or machine size. Even in the other two applications (N-body and volume rendering), the working set is quite small and scales very slowly with problem size, so that small caches will suffice for the foreseeable future. There is one application (the iterative solver) in which a large working set also has an important performance impact; however, accommodating this

working set requires the cache to be essentially as large as the local data set per processor, which is not a realistic design point for the near future.

In the second part of our treatment of an application, we use the information about working set sizes as well as other relevant application characteristics to reflect upon desirable grain sizes for machines. The grain size (or granularity) of a machine can be loosely defined as the amount of main memory and cache per processor on the machine. Using several approximations and simplifying assumptions, we find that all but one of our applications can effectively use large numbers of processors with small amounts of main memory and cache each. The argument for fine-grained machines from an applications perspective is further strengthened when time constraints are incorporated in the scaling model. However, there are reasons why one might not want to actually build machines with small amounts of memory per processor in the near term, and we discuss some of these.

The paper is organized as follows. In the next section, we describe the methodology and framework we use to study the applications. Sections 3 through 7 discuss the individual applications. In Section 8, we discuss our results and some caveats to the argument for fine-grained machines. Finally, Section 9 summarizes the main conclusions of the paper.

2 Methodology and Framework

In this section, we describe the common framework we use to present the results for each application. Sections 2.1 through 2.3 exactly mirror the structure of the computation description, working set size and grain size discussions in each individual application section, and also describe the methodology we use to obtain our results. Section 2.4 states some additional simplifying assumptions that we make.

2.1 Description of Computation

Our discussion of each application begins with a description of the most important steps of the computation. To make our investigations concrete, we also describe a *prototypical problem*. Our prototypical problem for every application is one whose data set is 1 Gbyte and is distributed at 1 Mbyte per node on a 1024 node machine. This is intended to represent a fine-grained machine configuration.

2.2 Working Set Hierarchy

The second subsection for each application identifies the important application working sets. To determine the sizes of these working sets, we simulate a cache-coherent, shared-address-space multiprocessor architecture, with each processor having a single level of cache and an equal fraction of the total main memory. For a given problem size and number of processors, we simulate different cache sizes and look for knees in the resulting performance (or miss rate) versus cache size curve.

To exclude the effects of *conflict* misses, which are influenced by a host of low-level artifacts, we use fully associative caches with an LRU replacement policy. To the extent that conflict misses are important, working set sizes measured this way are aggressive estimates of desirable cache size, and real caches—with low degrees of associativity—will need to be somewhat larger. For the first three applications we consider, the difference between a cache with limited associativity and a fully associative cache is not significant, since the cache conflict problem can easily be avoided. We comment on the use of direct-mapped caches for the other two applications in their respective sections. Finally, to exclude *cold-start* misses where appropriate, we omit the first few time-steps or iterations in those applications that are in reality expected to proceed over many time-steps or iterations. Thus, what we measure are misses due to *inherent communication* and finite cache *capacity*.

The first three applications we consider are well-understood and highly predictable computational kernels. In these cases (direct solvers, iterative solvers, and the FFT), we determine the working set sizes analytically, and use simulation to confirm our estimates for some examples. Since these applications are highly floating-point intensive, the metric we use to describe cache miss rates is number of double-word read misses per double-precision floating-point operation. The other two applications, Barnes-Hut and volume rendering, are full-scale applications that are not as regular, analytically describable, or floating point dominated. In these cases, we use simulation to look for knees in the read miss rate (read misses divided by number of read references) rather than misses per FLOP. We focus on read misses since these are likely to have a much greater impact on performance than write misses, the latencies of which can be easily hidden in these programs.

Scaling: Having determined the working set sizes for the prototypical problem, we then look at how these sizes scale with various application parameters and numbers of processors. We assume for this discussion that machines are made larger by adding processors, each processor bringing with it an amount of cache and memory equal to the cache and memory per processor on the original machine. We first examine how the working sets scale with individual parameters, and then look at how they scale under certain accepted models of scaling problems to run on larger machines. The two scaling models we consider are memory-constrained (MC) and time-constrained (TC) scaling. Given a larger machine, the MC scaling model assumes that a user will scale the problem to fill the available main memory on the machine, regardless of the effect this has on execution time. The TC scaling model, on the other hand, assumes that the user will increase the problem size so that the new problem takes as much time to solve on the new machine as the old problem took on the old machine. For more information about these scaling models, see [9].

2.3 Grain Size

Having understood the working sets, we then examine other application characteristics that affect the desirable granularity of processing nodes. In particular, we study the implications of interprocessor communication costs, load balance, and problem concurrency for node granularity. We begin by looking at the impact of these issues for the prototypical problem, and then we study how this changes with the problem and machine size.

Communication Costs: To determine the relative cost of interprocessor communication for each application, we first calculate a computation to communication ratio for the prototypical problem. To provide some feeling for what ratios we would consider sustainable, let us consider relevant parameters on existing and likely future parallel machines. One example is the Intel Paragon machine. Each node in this machine will have four 50-MFLOPS processors, yielding 200 MFLOPS per node. The machine uses a 2-D mesh interconnect with 200-Mbyte-per-second channels. Let us first consider nearest-neighbor communication. In this case, the bandwidth in the Paragon is limited by that of the node-to-router link, which is 200 Mbytes/sec peak. The sustainable ratio, in FLOPs per double-word, is therefore $\frac{200}{200/8} = 8$. For more random communication, sustainable communication volume is determined by the bisection width of the network. For a 32x32 (1024) node Paragon, the number of network links across a bisector is 64. Assuming that half of all random messages cross this bisector, each processor can generate only $64/512$,

or one-eighth as much traffic as in the nearest-neighbor case, yielding a sustainable ratio of 64 FLOPs/word. Similarly, the sustainable ratios on the Thinking Machines CM-5 are about 50 FLOPs per word for nearest-neighbor communication and about 100 for general communication (assuming 128MFLOPS vector nodes, 20Mbyte/sec nearest-neighbor communication bandwidth and 5Mbyte/sec general bandwidth).

As technology progresses, we should see both faster floating point processors and faster communication chips. For this paper, we simply assume that computation to communication ratios of 1-15 FLOPs/word are extremely difficult to sustain, 15-75 are sustainable but not easy, and above 75 are quite easy to sustain. (Of course, all the analytical and experimental data we provide remain valid even if the reader makes different assumptions about sustainability than we do.)

Load Balance and Concurrency: Two other potential sources of difficulty in obtaining high parallel performance are load imbalances and deficiencies in available problem concurrency. We comment on the expected impact of these for the prototypical problem.

Desirable Grain Size: We then attempt to determine what would constitute a desirable processor grain size for the prototypical problem. Our goal is not to make fine distinctions in grain size, but rather only very coarse ones. That is, we are not trying to determine whether the appropriate grain size is 1 Mbyte or 2 Mbyte of main memory per processor, but rather whether it is on the order of 1 Mbyte, 10 Mbytes or 100 Mbytes. To estimate a desirable grain size, we examine the expected parallel performance—based on communication cost, load balance, and concurrency considerations—for two variations of the prototypical problem with very different granularities. The first is a 1 Gbyte problem on 64 processors, resulting in 16 Mbytes of data per processor. The second is the same problem on 16 thousand processors, resulting in 64 Kbytes of data per processor.

Scaling: Finally, we consider how this desirable grain size changes as the problem is scaled.

2.4 Other Assumptions

We make a few additional simplifying assumptions in our analysis. We assume that the processor is based on commodity processor technology and thus is a given; its performance does not change when the number of processors is changed. We also assume that since the machine supports a shared address space, it is optimized for small data exchanges between processors and thus provides inexpensive interprocessor synchronization. Finally, we ignore the impact of contention in various parts of the machine as well as that of locality in the network topology, with the exception of our coarse notion of local versus random communication patterns discussed earlier in this subsection.

3 Direct Methods for Solving Linear Systems

The first application we consider is the LU factorization of large, dense matrices. This important and widely used computation factors a matrix A into the form $A = LU$, where L is lower-triangular and U is upper-triangular. The most common source of large dense LU problems is radar cross-section problems, where people currently solve problems that require several hours on today's largest parallel machines.

While we specifically examine dense LU factorization in this section, our analysis actually applies to a wider set of applications. Applications with very similar structure include dense QR factorization, dense Cholesky factorization, dense eigenvalue methods, and in many respects sparse Cholesky factorization.

3.1 Description of Computation

Dense LU factorization can be performed extremely efficiently if the dense $n \times n$ matrix A is divided into an $N \times N$ array of $B \times B$ blocks, ($n = NB$) [11]. The following pseudo-code, expressed in terms of these blocks, shows the most important steps in the computation.

```
1.  for K = 0 to N do
2.      factor block A_KK
3.      compute values for all blocks
            in column K and row K
4.      for J = K + 1 to N do
5.          for I = K + 1 to N do
6.              A_IJ ← A_IJ − A_IK A_KJ
```

The dominant computation here is Step 6, which is simply a dense matrix multiplication.

The parallel computation corresponding to a single K iteration in the above pseudo-code is shown symbolically in Figure 1. Two details have been shown to be crucial for reducing interprocessor communication volumes and thus obtaining high performance. First, the blocks of the matrix are assigned to processors using a 2-D scatter decomposition [2]. That is, the processors are thought of as a $P \times Q$ grid, and block (I, J) in the matrix is assigned to processor (I mod P, J mod Q). A simple 3×3 processor example is shown in Figure 1. Second, the matrix multiplication in Step 6 above is performed by the processor that owns block $A_{I,J}$. Within one K iteration, a processor thus uses blocks in the appropriate rows of column K (those blocks owned by a processor in the same row of the processor grid) and the appropriate columns of row K to update blocks it owns. The shaded blocks in Figure 1 are the blocks that processor P1 uses in one K iteration.

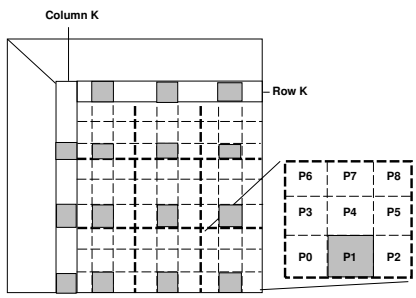

Figure 1: Dense block LU factorization.

Three factors must be traded off in choosing an appropriate block size B. Larger blocks lead to lower cache miss rates. However, larger blocks also increase the fraction of the computation performed in the less parallel portion of the computation (Steps 2 and 3 in the earlier pseudo-code), and can also cause load balancing problems. Relatively small block sizes ($B = 8$ or $B = 16$) can be shown to strike a good balance between these factors.

3.2 Working Set Hierarchy

Our prototypical 1 Gbyte data set on 1024 processors corresponds to a roughly $10,000 \times 10,000$ LU factorization problem.

Since people are currently solving $50,000 \times 50,000$ dense systems arising from radar cross-section applications on 128 processor machines, our choice of a smaller problem on a larger machine is actually somewhat aggressive.

The structure of LU factorization is sufficiently simple that we can derive working set sizes analytically. Figure 2 shows analytical cache miss rates for an $n = 10,000$ matrix, using block sizes of B = 4, 16, and 64, and $P = 1024$ processors. The graph shows double-word cache misses as a fraction of double-precision floating-point operations. The important levels

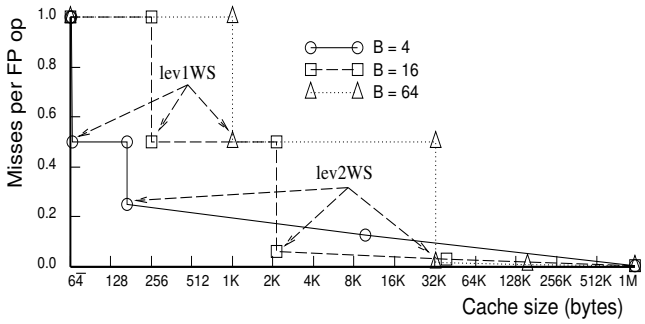

Figure 2: Miss rates for LU factorization, $n = 10,000$, $PE = 1024$.

of the working set hierarchy are as follows. The level 1 working set (lev1WS) consists of two columns of a block, and is roughly 260 bytes for $B = 16$. Once two columns fit, one column can be reused, roughly halving the overall miss rate. The second working set (lev2WS) consists of an entire $B \times B$ block, and is roughly 2200 bytes for $B = 16$. When this working set fits in the cache, the miss rate drops to roughly $1/B$. The other block sizes ($B = 4$ and $B = 64$) naturally lead to different level 1 and level 2 working sets sizes and miss rates.

Clearly, the cache sizes required to hold the lev2WS are much smaller than the caches people are building on parallel machines today, even for relatively large block sizes (B =16 or 32). The resulting miss rates are small enough to yield high performance. Almost all the misses would be serviced from a processor's local memory, provided the matrix blocks are placed in the local memories of their owner processors. Also, the misses are predictable enough to be easily prefetched.

The next working set (lev3WS) includes all blocks in row/column K that affect blocks owned by a particular processor (e.g., the shaded blocks in row/column K of Figure 1). The size of lev3WS is $2NB^2/\sqrt{P} = 2nB/\sqrt{P}$ (roughly 80 Kbytes for $B = 16$). If the lev3WS fits in cache, then the miss rate is further reduced by a factor of 2 to $1/2B$. However, the miss rate is small enough even before the lev3WS is reached, so that the lev3WS is of only minor importance to performance.

The final working set (lev4WS) is the set of all blocks belonging to a processor. If the cache accommodates the lev4WS (of size n^2/p), the miss rate is equal to the communication miss rate.

Scaling: When considering problem or machine size scaling, we note that the most important working set, the lev2WS, depends only on the block size B. It is independent of n and P. In other words, a small amount of cache is sufficient for any problem or machine size.

3.3 Grain Size

Communication Costs: LU factorization of an $n \times n$ matrix performs roughly $2n^3/3$ floating-point operations. Every block in the matrix is communicated to a row or column of $\sqrt{P}$ processors, yielding an overall communication volume of $n^2\sqrt{P}$. The computation to communication ratio is thus $2n/(3\sqrt{P})$, and depends only the grain size (n^2/P). For our prototypical problem, with its 1 Mbyte grain size, this yields a ratio of roughly 200 floating-point operations per floating-point word of communication—a relatively low bandwidth requirement. Also, most of these interprocessor communication costs can be hidden from the processors (using software prefetching, for example).

Load Balance and Concurrency: Another important issue that affects parallel performance is the load balance and available concurrency of the computation. For our 10,000 by 10,000 prototypical dense LU example with $B = 16$, each of the 1024 processors is assigned roughly 380 blocks from the matrix. This is a large enough number of blocks for dense LU factorization that load balancing and concurrency issues do not detract significantly from achieved parallel performance either.

Desirable Grain Size: Clearly, a 1024-processor machine with 1 Mbyte of data per processor would produce good processor utilization. Let us consider whether the grain size can reasonably be reduced to solve the same 1 Gbyte problem. Consider solving the problem on a 16K processor machine with 64 Kbytes of memory per processor. The computation to communication ratio would decrease by a factor of four to 50 operations per communicated datum, more difficult but still quite possible to sustain. The larger effect comes from load imbalance. With $B = 16$, each processor would now be assigned 25 blocks, which would reduce processor performance somewhat. This load balance problem could be improved by reducing the block size, but at a cost of increased cache miss rates. In either case, the higher computation to communication ratio, combined with the performance loss due to either poorer load balance or higher cache miss rates, would reduce per-processor performance. Thus, while a 1 Mbyte grain size is easy to sustain for a 1 Gbyte problem, a 64 Kbyte grain size is not so easy.

Scaling: Let us now see how the desirable grain size changes as larger problems are run. Keeping the grain size fixed at 1 Mbyte per processor allows us to factor a 20,000 by 20,000 matrix on 4096 processors. Compared with the prototypical problem, this problem would require the same amount of cache memory, would produce the same computation to communication ratio, and would generate a very similar computational load balance (since each processor still handles 380 blocks ($B = 16$)). We therefore conclude that the desirable grain size is independent of the problem size.

Keeping the grain size fixed while increasing the number of processors results in memory-constrained (MC) scaling. Since the amount of computation (which scales as n^3) grows much faster than the data set size (which scales as n^2), the parallel execution time grows quite quickly under MC scaling, which may therefore be an unacceptable scaling model for this application. If, on the other hand, a time-constrained scaling model were used, the per-processor data set would shrink with increasing P (of course, the performance of the individual processors would decrease as well). Constraints on execution time therefore provide another argument for finer-grained processing nodes on large-scale machines.

3.4 Summary

To summarize, we have found that dense LU factorization places very modest demands on a parallel machine. A small cache is sufficient to reduce the cache miss rate to nearly negligible levels, even for large problems on large machines. Similarly,

a small amount of per-processor memory (1 Mbyte or less) is sufficient to yield good performance, regardless of n and P.

4 Iterative Methods for Solving Linear Systems

The next class of computations we consider are iterative methods for solving linear systems of equations (or for finding eigenvalues of large sparse matrices). Iterative methods, which begin with a guess at the solution and iteratively attempt to improve this guess, are finding increasing use in solving large systems of equations in parallel. At the heart of these iterative methods is a sparse matrix-vector multiply, typically accompanied by some combination of vector additions and dot products. While we specifically consider the conjugate gradient (CG) method for solving sparse linear systems of equations here, the results should be similar for a range of other iterative methods.

4.1 Description of Computation

Each iteration of the CG method performs a single sparse matrix-vector multiply, 3 vector additions, and 2 dot products. The matrix-vector multiply is the dominant computation. This operation is most easily described by considering the sparse matrix A as a graph $G = (V, E)$, with a vertex $v \in V$ corresponding to each row/column in A and a weighted edge $i, j \in E$ corresponding to each non-zero $A_{i,j}$. The sparse matrix-vector multiply $b \leftarrow Ax$ is performed by associating an x value with each vertex in the graph, and iterating over all vertices. For every vertex i, the value of b_i is computed by summing the products of the weights of the edges (i, j) incident to i with the x values at the adjacent j vertices.

The CG computation is parallelized by partitioning the vertices in the graph representation of the matrix among processors. Consider the case where the graph representation of the sparse matrix is a simple 2-D grid (Figure 3). The example grid is partitioned among 4 processors in the figure. At each CG iteration,

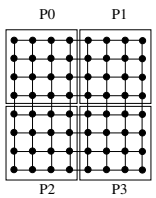

Figure 3: A 2-D grid partitioned among 4 processors.

a processor iterates over the points assigned to it, computing new values for b at its vertices. Interprocessor communication is necessary when a processor handles a vertex that is adjacent to a vertex belonging to another processor (the vertices on the boundaries between processor partitions in Figure 3), since the value at the other end of that edge was presumably changed in the previous iteration.

Our prototypical 1 Gbyte problem on 1024 processors corresponds to a roughly 4000×4000 2-D grid. An important trend in problem domains that use iterative methods is toward 3 dimensional problems. In this case, the prototypical problem corresponds to a $225 \times 225 \times 225$ 3-D regular grid.

4.2 Working Set Hierarchy

A processor sweeps through the entire set of nodes assigned to it in every iteration, touching the data corresponding to every edge incident to these nodes. Thus, unless this entire data set fits in the cache, the computation provides few opportunities to reuse data.

The working set hierarchies for our 2-D and 3-D grid examples on 1024 processors are shown in Figure 4. For the 2-D problem, the lev1WS consists of the x values from three adjacent sub-rows of points assigned to a processor. This lev1WS is quite small, consisting of roughly 5 Kbytes of data in the prototypical 2-D problem. While the impact of this 5 Kbyte working set on miss rate is significant, the miss rate remains high even after this working set fits in the cache. The lev2WS consists

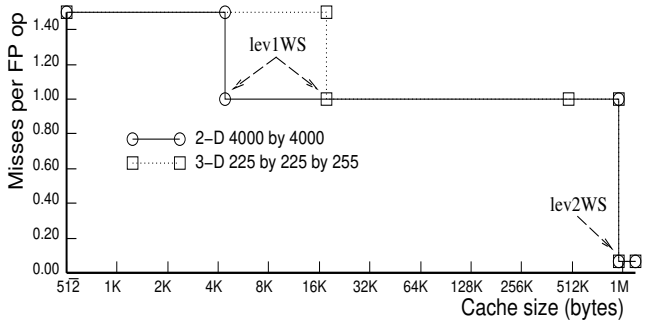

Figure 4: Miss rates for CG, 4000×4000 grid, $P = 1024$.

of the entire set of data owned by a processor. At this point, the miss rate drops to the communication miss rate. However, it is generally unreasonable to expect this set of entries to fit in cache.

For the 3-D grid computation, the working sets are quite similar. The major difference is in the lev1WS, which now consists of 2-D cross-sections from the 3-D region assigned to each processor, and thus represents a larger data set than the 1-D subrows from the 2-D grid. In the prototypical problem, the lev1WS grows from 5K to 18K. Note that these numbers are still smaller than the first-level caches found in nearly all modern processors.

Scaling: If we expect the per-processor data set to be larger than the processor cache, then the only working set that can fit in the cache is the lev1WS. Since each processor receives an $n/\sqrt{P} \times n/\sqrt{P}$ portion of the $n \times n$ 2-D grid, the size of the lev1WS is proportional to $n/\sqrt{P}$. The size of this working set therefore remains quite moderate. A problem that requires 16 Mbytes of storage per processor, for example, would have lev1WS sizes of 18 Kbytes and 90 Kbytes for 2-D and 3-D grids, respectively. Furthermore, the size of lev1WS can actually be kept constant through the use of blocking techniques.

The fact that fitting the lev2WS (a processor's entire partition of the grid) in the cache has a substantial impact on the performance of CG brings up an interesting design issue. Particularly under time-constrained scaling, the data set per processor may not be very large on large-scale machines, so that it may make sense to build larger caches and fit the lev2WS in the cache. This amounts to fitting the entire data set in cache memory, so that there is no need for DRAM memory. While this may be an interesting design point for very large-scale machines, we restrict ourselves here to a more conservative model where the per-processor data set is much larger than the cache.

4.3 Grain Size

Communication Costs: The total amount of computation in one CG iteration on an $n \times n$ 2-D grid is roughly $10n^2$ operations. Each processor owns a $n/\sqrt{P} \times n/\sqrt{P}$ grid of points. The $4n/\sqrt{P}$ points along the perimeter must be communicated to neighboring processors in every iteration. The computation

to communication ratio is thus $5n/(2\sqrt{P})$, and once again depends only on the grain size. For the 1 Mbyte grain size of our prototypical problem, the ratio would be roughly 300 FLOPs per word. This high ratio, combined with the fact that the communication latencies can be easily hidden due to the very regular structure of the computation, make a 1 Mbyte per processor grain size quite appropriate for CG on 2-D grid problems.

For a 3-D grid problem, each processor would own a 3-D subgrid that is $n/\sqrt[3]{P}$ on a side. The processor would have to communicate the values on the 6 2-D faces of its subgrid to other processors. The computation to communication ratio would be $7n/(3\sqrt[3]{P})$, yielding a ratio of roughly 50 for the prototypical problem. This ratio is not as easily sustained as the ratio for 2-D problems, but it is still feasible.

Load Balance and Concurrency: The regularity of a grid computation makes load balancing quite simple. The only limitation on concurrency is the global sum that accompanies the two dot product operations. Given our assumptions about the costs of interprocessor communication and processor synchronization, the cost of the fully parallel portion should dwarf the cost of the less parallel global sum in the prototypical problem. Thus, the problem exposes sufficient concurrency for 1024 processors.

We should note that many important problems (e.g., unstructured problems that model complex physical structures) will not be nearly as regular as the 2-D and 3-D grids considered here. This reduced regularity will require more sophisticated strategies for partitioning the problem among a set of processors. This will have three important effects. First, the computational load balance among the processors will certainly not be as good. Second, the computation to communication ratio for problems with the same data set size will most likely be significantly higher. Finally, the partitioning step itself will represent a computational overhead whose cost increases with the number of processors. This partitioning step will generally possess limited parallelism, so the presence of more processors would not necessarily reduce its cost.

We conclude from the above discussion that a 1024 processor machine with 1 Mbyte of memory per processor would be quite appropriate for regular 2-D problems. The appropriate grain size for irregular problems or 3-D problems may be somewhat larger.

Desirable Grain Size: Let us see if we can use a 16K-processor machine with only 16 Kbytes or memory each to solve a 1 Gbyte problem. The computation to communication ratios increase to roughly 75 and 20 for 2-D and 3-D grids, respectively. Thus, the desirable grain size is somewhere between 1 Mbyte an 16 Kbytes for the prototypical CG problem as well.

Scaling: Now consider how the appropriate grain size would change with a scaled problem. The important thing to note here is that the computation to communication ratio for both 2-D and 3-D grid problems depends only on the volume of data on one processor, and is independent of the number of processors. Thus, if a grain size of 1 Mbyte per processor produces sustainable communication volumes on P processors, then it would also produce sustainable volumes on $2P$ processors, given a problem that is twice as large. The one other issue that might be relevant when considering scaling is the cost of the global sum operation in the dot products. While this cost clearly increases with P, the rate of increase ($O(\log P)$) is sufficiently slow that, under our machine model, this cost would not be a significant performance drain for practical P.

4.4 Summary

We therefore conclude that the conjugate gradient method requires a somewhat larger grain size than dense LU factorization. The desirable grain is still quite small, however. A 1 Mbyte per processor data set size appears reasonable.

5 Transform Methods (FFT)

The next computation we consider is the 1D complex fast Fourier transform (FFT). Our analysis in this section also applies to the complex 2D and 3D FFT. These computations form the computational core of a wide variety of applications from the fields of image and signal processing as well as climate modeling.

5.1 Description of Computation

The structure of the FFT computation is captured by the familiar butterfly network. For an $N = 2^M$ point FFT, the computation proceeds in M stages, where in stage s, pairs of data points at a distance of 2^s interact with each other to produce the points at stage $s + 1$.

In a straightforward parallel implementation, each processor handles a contiguous set of points. During the first $\log N - \log P$ stages of the butterfly, processors work locally with no interprocessor communication. In each of the remaining $\log P$ stages, all N points are exchanged between processors,

Unfortunately, the simple, so-called radix-2 FFT computation described above makes very poor use of the memory system. It sweeps through all N points in one stage of the butterfly before moving on the next stage, thus making little use of the processor cache. In the last $\log P$ stages of the butterfly, the processors perform only a single computation step on each communicated point, thus producing a very low computation to communication ratio. Both the cache usage and the computation to communication ratio can be improved dramatically by increasing the *radix* of the computation (see [1] and [12]). Increasing the radix is equivalent to 'unrolling' the butterfly, performing multiple butterfly stages in a single pass through the data. A radix-8 FFT, for example, would combine three butterfly stages into a single stage, where each step in this new stage performs operations on 8 points simultaneously. A radix-r FFT would combine $\log r$ butterfly stages into a single stage, operating on r points simultaneously.

An efficient parallel FFT is therefore structured as follows. To minimize interprocessor communication, the overall computation is performed with as large a radix as possible. This turns out to be radix-D, where D is the number of points assigned to each processor (i.e., $D = N/P$). Thus, the $\log N$ stages of the butterfly are grouped into sets of $\log D$ stages. At each radix-D stage, a processor receives D points from other processors, performs $\log D$ stages of the butterfly on these points, and sends the resulting D data points to the processors that use them in the next stage.

To make good use of the processor cache, the radix-D stages are further subdivided into smaller internal groups. For example, a processor might perform the $\log D$ stages in the radix-D computation three-at-a-time, essentially performing a radix-8 computation within the radix-D computation. We call this smaller radix the internal radix. This further sub-division produces a smaller processor working set than would be present if all $\log D$ stages were performed in a single sweep. We use this more efficient parallel FFT in the results we present below.

5.2 Working Set Hierarchy

The prototypical 1 Gbyte problem corresponds to a 64 million point complex FFT on 1024 processors, yielding 64K points per processor.

Working set hierarchies for radix-2, radix-8, and radix-32 FFT computations on this data set are shown in Figure 5.

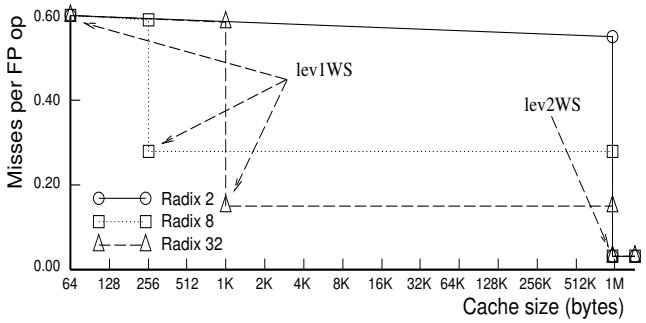

Figure 5: Miss rates for 1D FFT, $n = 64M = 2^{26}$, $PE = 1024$.

The first and most important working set (lev1WS) contains the set of data items needed to perform a single step of a stage sweep. If the internal radix is 2, the lev1WS simply consists of the two data points that are at distance 2^s from each other. The miss rate is 0.6 misses per op when the lev1WS with radix 2 fits in the cache. For internal radices of 8 and 32, lev1WS consists of the relevant 8 or 32 data points, and brings the miss rates to roughly 0.25 and 0.15 misses per operation, respectively. These misses can be easily prefetched. Thus, a small cache is sufficient to significantly reduce the miss rate for parallel FFT.

The only other working set in a parallel FFT (lev2WS) is simply the entire data set assigned to a processor.

Scaling: The size of the important, level 1 working set depends only on the internal radix. The choice of internal radix is independent of the problem size and the machine size, and a small radix suffices to keep the capacity miss rate small. Consequently, a small cache (a few Kbytes) is sufficient for any problem size or machine size. The lev2WS depends on N and P, but is not expected to fit in a cache.

5.3 Grain Size

Communication Costs: The computation to communication ratio is most easily estimated by considering the operations that a processor performs in a single stage of the radix-D computation ($D = N/P$). Within a single stage, a processor performs $5D \log D$ operations, and then communicates all $2D$ double-words computed in that stage to other processors. Thus, the overall computation to communication ratio $\frac{5}{2} \log D = \frac{5}{2} \log \frac{N}{P}$, and depends only the grain size $\frac{N}{P}$.

This ratio is unfortunately inexact due to quantization effects. Consider our prototypical problem, with 1024 processors and 64K points per processor. The resulting radix-64K FFT groups the butterfly into sets of $\log 64\text{K}$ or 16 stages. The difficulty is that the whole problem only requires $\log 64M = 26$ stages. The second stage would therefore perform only 10 stages of computation for one communication stage, less than the 16 stages assumed by the model.

The actual computation to communication ratio can be determined by noting that the whole computation performs $5N \log N$ operations, and it communicates the $2N$ words of data twice between processors. For our prototypical problem, $N = 64\text{M}$, yielding a ratio of 33. While this ratio would be sustainable if the communications were between neighbor processors, unfortunately it can be shown that communication in the FFT exhibits little locality for most processor interconnection topologies. The exception is a hypercube topology, which is becoming less and less common in large-scale parallel machines. The ratio of 33 operations per word would thus be difficult to sustain.

Load Balance and Concurrency: A very simple distribution of the FFT computation is quite adequate for load balancing. Furthermore, there is more than enough available concurrency to keep a very large number of processors busy (ignoring processor stalls due to communication).

Desirable Grain Size: We have seen that a 1 Mbyte data set per processor produces a computation to communication ratio that is difficult to sustain. A finer-grain machine would clearly exacerbate the problem. Let us therefore examine how this ratio would change if the same problem were solved on a coarser-grain machine. On a machine with one-sixteenth as many processors ($P = 64$), we find that the computation to communication ratio surprisingly does *not* change. This is an artifact of the quantization of levels discussed earlier: there are still two communication stages in the computation.

Let us now consider just how coarse the machine grain must be to produce a sustainable computation to communication ratio. If we even use the optimistic expression for computation to communication ratio of $\frac{5}{2} \log \frac{N}{P}$ derived earlier, a ratio of R requires the number of data points per processor to be $N/P = 2^{\frac{2}{5}R}$. The exponential growth rate of per-processor memory required to improve computation to communication ratios has been previously noted in [4]. The consequences of this growth rate are quite severe. Increasing the computation to communication ratio from 33 to a more easily sustained ratio of 60, for example, would require the per-processor data set to be increased to roughly 270 Mbytes. A ratio of 100, which may be required by some machines for good performance, would require approximately 18 Terabytes of data per processor. It is clearly unrealistic to try to significantly increase the computation to communication ratio by increasing the node grain size.

Scaling: Since the main factor limiting performance, the computation to communication ratio, depends only on grain size, the "desirable" grain size is essentially independent of the problem size or number of processors. MC scaling therefore produces comparable processor utilization on larger machines.

5.4 Summary

The FFT is a difficult computation for large scale parallel machines. While the FFT is easily blocked for a cache to provide high per-processor performance, the communication volume inherent in the computation is sufficiently high that communication costs will certainly dominate the execution time. While one might conclude that the solution to this high communication volume is to increase the processor grain size, unfortunately the grain size increase that would be required to significantly reduce communication volumes is unrealistically large.

6 Hierarchical N-Body Methods

The classical N-body problem is to simulate the evolution of a system of bodies (e.g. stars in a galaxy) under the forces exerted on each body by the whole system. Typical domains of application include astrophysics, electrostatics and plasma physics, among others. As in many other computational domains, hierarchical solution methods have recently attracted a lot of attention for N-body problems, since they construct efficient algorithms by taking advantage of fundamental insights into the nature of physical processes. The two most prominent hierarchical N-body methods are the Barnes-Hut and Fast Multipole methods. We shall use a three-dimensional galactic Barnes-Hut simulation as our example in this paper [10].

6.1 Description of Computation

The computation in N-body problems proceeds over a number of time-steps. Every time-step computes the forces experienced by all bodies, and uses these forces to update the positions and velocities of the bodies. The force-computation is by far the most time-consuming phase in a time-step, and we focus on it in our analysis (although our measurements include the whole application).

The main data structure used by the Barnes-Hut method is an octree which represents the computational domain. The root of the octree is a cubical space that contains all particles in the system. Internal cells of this tree represent recursively subdivided space cells, and the leaves represent individual bodies. The tree is traversed once per body to compute the net force acting on that body. The force-calculation starts at the root of the tree and conducts the following test recursively for every cell it visits. If the center of mass of the cell is far enough away from the body, the entire subtree under that cell is approximated by a single particle at the center of mass. Otherwise, the cell must be "opened" and each of its subcells visited. A cell is determined to be far enough away if the relationship $\frac{l}{d} < \theta$ is satisfied, where l is the length of a side of the cell, d is the distance of the body from the center of mass of the cell, and θ is a user-defined accuracy parameter (θ is usually between 0.5 and 1.2). In this way, a body traverses deeper down those parts of the tree which represent space that is physically close to it, and groups distant bodies at a hierarchy of length scales. For large problems, higher order moments than the center of mass (for example, quadrupole moments) are used to increase force-computation accuracy without making θ too fine. We assume the use of quadrupole moments in our discussion.

6.2 Working Set Hierarchy

There are three important levels of the working set hierarchy in these methods. These are shown in Figure 6 for a small problem simulating 1024 particles on 4 processors. We start with a smaller problem in this application than the prototypical problem used for other applications because the working sets here are measured through simulation rather than analysis, and because it is impossible to simulate the prototypical problem on our multiprocessor simulator. The small problem, however, exposes all the important characteristics and constant factors, and the scaling trends that we discuss below have been verified by simulating some larger problems and machines.

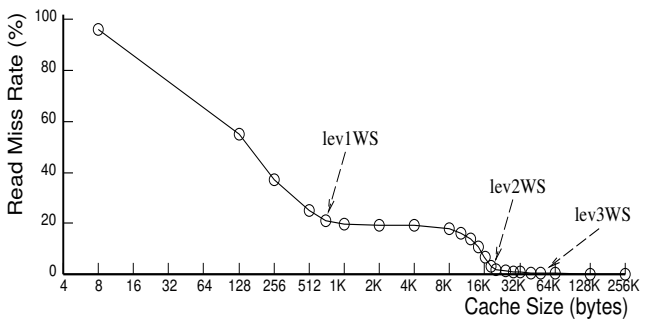

Figure 6: Working Sets for the Barnes-Hut Application: n=1024, theta=1.0, p=4, quadpole moments.

The lev1WS in this application is the amount of temporary storage used to compute an interaction between a particle and another particle/cell. It is only about 0.7 Kbytes in size. Having a cache large enough to hold the lev1WS reduces the miss rate from 100% with no cache to about 20% in most cases we have simulated. While this is a large reduction, the miss rate is still not low enough for effective performance since most of these misses are to nonlocal data, and are not predictable enough to be easily prefetched.

The lev2WS is the most important working set in the application. It comprises the amount of from the tree needed to compute the force on a single particle. These data include particle positions as well as cell positions and moments. If the partitioning of particles among processors is done appropriately, most of these data will be reused in computing the forces on successive particles. Caches large enough to hold this working set take the miss rate quite close to the inherent communication miss rate obtained with infinite caches (0.2% for this problem). For this small problem, the size of the lev2WS is 20 Kbytes.

Beyond the lev2WS, the miss rate decays much more slowly until the cache size reaches the lev3WS. The size of the lev3WS is roughly the maximum of (i) the amount of data in a processors partition and (ii) the amount of data that a processor needs to compute the forces on all the particles in its partition. Thus, the lev3WS size decreases with increasing number of processors and increases with increasing force computation accuracy (decreasing θ). However, since the lev3WS marks the culmination of a slow decrease in miss rate, and since the capacity miss rate is already very small after the lev2WS is reached, the lev3WS is not important to performance and we do not consider it further.

Scaling: A realistic problem that people run today is one with 64K particles and θ=1.0. When run for 512 time-steps, this problem takes about three days on a single processor of an SGI 4D/240. We use this problem, running on 64 processors, as the starting point for our discussion of scaling. The lev1WS and lev2WS sizes for this problem are 0.7 Kbytes and 32 Kbytes, respectively.

The total data set size increases linearly with the number of particles, and is about 230 bytes per particle when quadrupole moments are used. It is independent of θ and essentially independent of the number of processors.

The lev1WS stays at 0.7 Kbytes independent of the number of particles, the number of processors, and θ. It changes slightly only with the order of moments used, and hence with the nature of an individual interactions.

The size of the important lev2WS is proportional to the number of interactions computed per particle, which is of order $\frac{1}{\theta^2}\ log\ n$ [3]. The lev2WS therefore scales very slowly with the number of particles n, more quickly with the accuracy parameter θ, and is independent of the number of processors p. The constant of proportionality in the above size expression is about 6 Kbytes. How the lev2WS scales with larger problems therefore depends on how n and θ are scaled, as we examine below.

Under memory-constrained (MC) scaling, n would increase linearly with p. If no other parameters are scaled, the size of the lev2WS grows very slowly, going from 32 Kbytes with 64K particles to 40 Kbytes with a million particles (about the largest number of particles that people run on the largest parallel machines today) and to only 60 Kbytes with a billion particles (inconceivable today). Scaling only n, however, is naive. In practice, all of n, θ and the time-step resolution Δt are likely to be scaled simultaneously, in order to scale their contributions to the overall simulation error at the same rate [9]. This leads to the following rule: If n is scaled by a factor of s, Δt must be scaled by a factor of $\frac{1}{\sqrt[4]{s}}$ and θ by a factor of $\frac{1}{\sqrt[8]{s}}$ when quadrupole moments are used. A caveat is that θ is likely to be decreased at this rate only up to a certain extent (θ=0.5 or so), at which point higher order moments such as octopole moments would be used to increase force computation accuracy without

reducing θ much.

The lev2WS grows faster with MC scaling under this realistic parameter scaling rule, since θ—the dominant contributor to the working set size—is also scaled. Even under this model, a billion particle problem (θ=0.6, octopole moments) would have a lev2WS of under 300 Kbytes. However, MC scaling of this sort causes the execution time to grow rapidly, so that MC scaling is in fact unrealistic in practice for this application.

Time-constrained scaling, while asymptotically limited in the amount the problem can be scaled, is more realistic in practice. In this case, the contributions of changing Δt and θ to the execution time don't allow n to scale linearly with p. In fact, n scales slower than $\sqrt{k}$, where k is the factor by which p is scaled. θ therefore scales more like $\frac{1}{\sqrt[16]{k}}$. The result is that both the data set size and the lev2WS (proportional to $\frac{1}{\theta^2}\ log\ n$) still increase in size, but much more slowly than under memory-constrained scaling. For example, starting from our 64K particle problem on 64 processors (θ=1.0), a 1K processor machine under TC scaling would run 256K particles (θ=0.84) rather than the 1 million (θ=0.71) under MC scaling. The lev2WS size in this case is only 25 Kbytes. A million processor machine would run not a billion particles but rather only about 32 million (θ=0.6, octopole moments), and the lev2WS size would be about 140 Kbytes.

The bottom line is that although the important working set for this application is not trivial for large problems, it is still well under 100 Kbytes for the largest problems people can run today, and is likely to stay reasonably small even for problems whose solution is beyond the realm of possibilities today.

6.3 Grain Size

Communication Costs: Modeling the amount of communication in the Barnes-Hut method accurately is very difficult. Using some curve fitting from [7] and some of our own, we find that the communication per processor required to compute forces in a time-step scales as $\frac{n^{1/3}\theta^3}{p^{1/3}} log^{4/3}p$, and that the communication to computation ratio is therefore $\theta \frac{p^{2/3}}{n^{2/3}} \frac{log^{4/3}p}{log\ n}$. Every unit of computation (a particle-particle or particle-cell interaction) is equivalent to about 80 instructions when quadrupole moments are used, and every unit of communication in the above expression is 3 double words of data.

Our prototypical problem for grain size discussions, which uses 1 Mbyte of main memory per processor on a 1024 processor machine (1 Gbyte total), solves a problem with about 4.5 million particles (a very large but feasible computation by today's standards). Let us assume that $\theta = 1.0$. Every processor is responsible for about 4500 particles, and the communication to computation ratio is very small, less than 1 double word per 10,000 processor busy cycles. Since the access patterns of this application are not predictable, communication latencies might not be hidden as effectively as in the regular computations we have discussed so far. However, the communication to computation ratio is very small, and communication does not become a bottleneck until the number of particles per processor becomes very small.

Load Balance and Concurrency: The concurrency in the application scales as the number of particles n, and load imbalance is also not a significant factor until the number of particles per processor (n/p) becomes very small. Given that n is typically large (4.5 million in the prototypical problem), this also means that very large numbers of processors can be used effectively.

Desirable Grain Size: The important per-processor growth rates for this application in terms of n, θ, Δt and p are as follows. The data set size scales as $\frac{n}{p}$, the computation as $\frac{1}{\theta^2}\frac{n\ log\ n}{p\Delta t}$, the working set as $\frac{1}{\theta^2}\ log\ n$, the concurrency as n, the communication as $\frac{n^{1/3}\theta^3}{p^{1/3}\Delta t} log^{4/3}p$, and the communication to computation ratio as $\theta \frac{p^{2/3}}{n^{2/3}} \frac{log^{4/3}p}{log\ n}$.

Clearly, we would get very good speedups on our 1 Gbyte problem on a coarser-grained machine than 1 Mbyte per processor, such as the 64-processor machine with 16 Mbytes of memory per processor. However, solving a 1 Gbyte problem on 64 processors would take a very long time. Let us see what happens when we go to the finer-grained machine instead, solving the same 1 Gbyte problem with 16K processors and 64 Kbytes of memory per processor. Every processor now has about 280 particles. The communication to computation ratio increases to about 1 double word per 1000 instructions, but is clearly very small still. However, particularly given the large number of processors, load balancing may become a problem at this point. The result is that the grain size can probably be pushed to a few hundred kilobytes per processor for a 1 Gbyte problem without compromising parallel performance much.

Scaling: Finally, let us see how the desirable grain size scales with problem size. A memory-constrained scaling model, in which the processor grain size remains constant, provides high processor utilization for this application. The number of particles per processor remains the same, so load balancing is not affected, and the communication to computation ratio either increases extremely slowly (if the accuracy is not scaled as well) or stays constant (if accuracy is scaled) [8]. The cache size needed per processor grows, but is still relatively small, as we have seen. However, such memory-constrained scaling to keep the grain size constant causes the execution time to increase very rapidly. If the goal is to run a problem in the same amount of time, n does not grow nearly as quickly as p. The grain size needed therefore decreases, as does the efficiency of a node (since both communication and synchronization increase relative to computation, and the load balance gets worse).

6.4 Summary

Our results show that fine-grained machines, with well under 1 Mbyte of memory and a couple of hundred kilobytes of cache, can be very effective for this application. A couple of issues, however, may inhibit going to a very fine grain. First, for large problems, the amount of fully associative cache needed will be as large as or larger than the local memory per node. The use of realistic—set-associative or direct-mapped—caches would further increase the required cache size, resulting in an expensive design point that may not be appropriate for other kinds of computations. Preliminary results with direct-mapped caches for small problems show that the knees in the miss rate versus cache size curves are not as well-defined as with fully associative caches, and that the direct-mapped cache size required to hold the important working set is about three times as large as the corresponding fully associative cache size. Set-associative caches and data restructuring might reduce this factor of three. While we have not simulated large problems with direct-mapped caches, there is little reason to believe that the factor increase in required cache size will be much different as the problem scales.

The second issue is that although the force-calculation phase can be parallelized very efficiently on large numbers of processors, some other phases—such as building the octree and computing the moments of cells—do not yield quite as good speedups due to larger amounts of synchronization and contention that they encounter. These phases consume a small fraction of the execution time on moderately parallel machines (at

least up to 512 processors for large problems), but may become significant for very fine-grained machines with very large numbers of processors.

7 Volume Rendering

Our next application is from the field of scientific visualization. Volume visualization techniques are of key importance in the analysis and understanding of multidimensional sampled data. This application, which renders volumes using optimized ray tracing techniques, uses a parallel version of the fastest known sequential algorithm for volume rendering [6].

7.1 Description of Computation

The volume to be rendered is represented by a cube of voxels (or volume elements). The outermost loop of the computation is over a series of frames or images. Successive frames correspond to changing angles between the viewer and the volume being rendered. For each frame, rays are cast from the viewing position into the volume data through every pixel in the image plane corresponding to that frame. The voxel data are resampled at evenly spaced locations along each ray by trilinearly interpolating the values of surrounding voxels. Rays are not reflected at all, but pass straight through the volume unless they encounter too much opacity and are terminated early. Finally, ray samples are composited to produce an image or frame. The goal of the application is to render individual frames in real time (30 frames/second), so that an interactive user can view the volume from arbitrarily changing positions efficiently.

7.2 Working Set Hierarchy

There are three important levels of the working set hierarchy in this application. Data reuse is afforded across sample points along a ray (lev1WS), across successive rays (lev2WS), and perhaps across successive frames (lev3WS). These working sets are shown in Figure 7 for a 256x256x113 voxel data set of a human head. While smaller than our prototypical 1 Gbyte problem (the data set is about 30 Mbytes) for reasons of simulation feasibility, this data set is a very realistic real-time challenge for today's parallel machines.

The voxel data in this application are read-only. An octree data structure is used to find the first interesting (non-transparent) voxel in a ray's path efficiently, as well as to determine whether the neighboring voxels around a sample point are interesting. The lev1WS consists of the voxel and octree data that are reused across neighboring sample points along a ray. This working set is very small: about 0.4 Kbytes. A cache that accommodates it reduces the read miss rate to about 15%, which is still too large to be acceptable, particularly since the misses are potentially to nonlocal data and the access patterns are not regular enough to be easily prefetched.

The lev2WS is the most important working set. It measures the fraction of the data used in computing a ray that is typically reused by the next ray. This reuse owes itself to the partitioning scheme, which assigns every processor a contiguous rectangular subblock of pixels in the image plane. Successive rays cast by a processor therefore pass through adjacent pixels and tend to reference many of the same voxels in the volume. The lev2WS is about 16 Kbytes for this data set, and a cache that accommodates this working set reduces the read miss rate to about 2%.

After the lev2WS is reached, the miss rate diminishes more slowly until the lev3WS is reached. The size of the lev3WS depends on how quickly the angle between the viewing position and the data set is changed between successive frames. If the change is gradual, as in our simulations, a given processor references many of the same voxels in successive frames; otherwise, the overlap may be negligible. Thus, the lev3WS size can vary from the voxels referenced by a processor in one frame to almost the entire voxel data set. For our data set and simulations, the lev3WS is about 700 Kbytes, and a cache that accommodates it brings the miss rate down to the communication miss rate of 0.1%. The lev3WS is therefore large, but is not very important to performance and we do not consider it further.

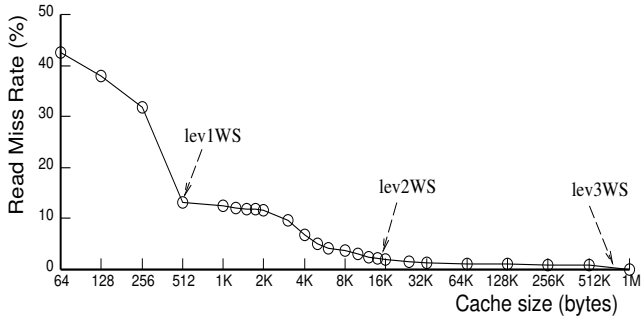

Figure 7: Working Sets for the Volume Rendering Application: 256x256x113 head, p=4.

Scaling: With n voxels along a single dimension, the data set for the volume rendering application is roughly $4n^3$ bytes. The two important parameters that might be scaled in this application are n and the number of processors p. The lev1WS size is independent of either of these. The lev2WS is also independent of p, but grows proportionally to n, corresponding to the number of voxels sampled along a ray. The size of the lev2WS is roughly (4000 + 110*n) bytes. Note that n here is only the cube root of the data set size.

Since the execution time grows at the same rate as the data set size (n^3), time-constrained scaling is essentially the same as memory-constrained for this application. Thus, the important working set grows as only the cube root of the number of processors under either scaling model, with a very small constant factor of only 110 bytes. Even for a very large, 1024x1024x1024 problem, far from renderable in real time on even the largest machines today, the lev2WS is only 116 Kbytes large. For a while, also, the push in using larger machines is going to be to render relatively small data sets in real time, rather than to render bigger data sets. Finally, as data sets get larger, the octree will probably be used to skip transparent voxels along a ray even after the first nonempty voxel is found, which may reduce the size of the lev2WS. Thus, the important working set of this application is likely to remain relatively small (under 100 Kbytes or so) for a while to come.

7.3 Grain Size

Communication Costs: The most important and heavily referenced data structure, the voxel data set, is accessed in a read-only fashion. Thus, if the entire voxel data set were replicated in the local memory of every processing node, there would be essentially no communication during rendering (except the small amount of communication generated by the ray-stealing performed to ensure load balancing toward the end of the rendering phase [6]). However, such replication would imply either unreasonable amounts of local memory per processor or that large data sets cannot be run. In our shared address space implementation, the data set is not replicated at all in main memory but only to some extent in the caches. Because of this, communication is generated when accessing voxel data, since voxel data get replaced in the caches.

If the cache provided is significantly smaller than the lev3WS, as is very likely, we can assume that almost all of the voxel data

that a processor accesses during a frame are not in its cache at the beginning of that frame. Since the viewing angle changes, the most reasonable data distribution across local memories is an interleaved or random one to minimize contention. Thus, the first accesses to voxel data in a frame have no more than a random chance of being satisfied in local memory, and are likely to generate communication. Two bytes of data are read per voxel, so that the total volume of communication in a frame is somewhat larger than $2n^3$ bytes (since processors overlap to some extent in the voxels they access). Since a frame involves more than $300n^3$ instructions, the computation to communication ratio is very large, close to 600 instructions per word of communicated data, independent of n or p (see the limitations of this analysis below). If caches yield reuse across frames, the computation to communication ratio will be even larger.

Our prototypical problem amounts to a 600x600x600 voxel problem on a 1024-processor machine, with every processor being responsible for about 1000 rays. Since the computation to communication ratio is independent of n or p, it is 600 instructions per word in this case as well.

Load Balance and Concurrency: After a processor has processed its statically assigned rays, it steals rays from other processors if it is idle. Stealing introduces additional synchronization and communication, and is the main source of performance loss if the number of rays stolen by a processor is large compared to the number initially assigned to it. In the prototypical problem, every processor is assigned 1000 rays, so that the amount of stealing is not significant.

Desirable Grain Size: The important per-processor growth rates for this application in terms of n and p are as follows. The data set size scales as $\frac{n^3}{p}$, the computation as $\frac{n^3}{p}$, the important working set as n, the communication as $\frac{n^3}{p}$, and the communication to computation ratio stays roughly fixed. The concurrency in the application is equal to the number of rays, which grows as n^2: There is one ray per pixel, and there are n^2 pixels in the 2-d image plane projected from the data set.

Running the 600x600x600 voxel data set on a coarser-grained machine than 1 Mbytes per processor (e.g. 64 processors with 16 Mbytes per processor) is obviously not a problem from the viewpoint of processor efficiency. However, a 64 processor machine would clearly not be able to render this data set in real time. Let us see what happens when we solve the same 600x600x600 voxel problem on a finer-grained machine, with 16K processors and 64 Kbytes of memory per processor. The communication to computation ratio (ignoring task stealing) is still about 600 instructions per word. However, every processor now processes roughly $\frac{(600\sqrt{3})^2}{16384}$ or 66 rays, likely to be too few for good load balancing without excessive stealing. As in Barnes-Hut, a grain size of a few hundred kilobytes is therefore likely to be adequate for good parallel performance on the 1 Gbyte data set.

Scaling: Finally, we examine how the desirable grain size changes as larger problems are run. If the data set size is increased by a factor of k, keeping the memory per processor or grain size fixed (and therefore scaling p by a factor of k) will cause every processor to process a smaller number of rays (decreasing by a factor of $k^{1/3}$, since the size of a ray grows by a factor of $k^{1/3}$). This is not a problem until the number of rays per processor becomes very small, in which case increased synchronization and communication due to task stealing detract from performance. To maintain the same number of rays per processor and hence roughly the same processor efficiency, the amount of memory per processor (the grain size) must increase by a factor of $k^{1/3}$ when the data set size is increased by k (the working set size per processor also grows as $k^{1/3}$). That is, the number of processors increases by a factor of $k^{2/3}$ rather than k. However, the execution time grows as $k^{1/3}$ as well in this case, which is not desirable from the viewpoint of real-time rendering.

Fortunately, the number of rays needed per processor to retain high processor efficiencies is small. And we mentioned earlier that the data set sizes are not likely to get too much larger in the near future, since the goal today is still to get moderately sized data sets rendered in real time. Thus, the memory needed per processor for this application is small and likely to remain so for some time to come.

7.4 Summary

Our general conclusion is that fine-grained machines (under 1 Mbyte of memory per processor) are likely to perform very well on this application.

8 Discussion

We begin our discussion by bringing together the results for the various applications. Table 1 shows the growth rates for the most important application characteristics, including data set sizes, total operations performed on these data, available concurrency, communication volumes, and the sizes of the most important working sets. Table 2 then shows the implications of these data, including the sizes of the important working sets for our prototypical 1 Gbyte, 1024 processor problem, expressed as a function of total data size (DS), and the desirable amounts of per-processor memory. Table 2 also shows growth rates for both as the problem is scaled. (We note that for the FFT the 'desirable' grain size of 1 Mbyte is not really all that desirable, but that enormous increases would be required to improve the situation.)

Our results show that reasonably fine-grained parallel machines, with memory of 1 Mbyte per processor or less, can be effective for the application classes studied here. However, we now briefly discuss some pragmatic reasons, both hardware and software, why coarser-grained machines are likely to continue being built in the near term.

On the hardware side, one reason is the fact that memory chips have large capacity but currently provide very narrow interfaces (1–8 bits wide). Thus, building the high-bandwidth memory systems that are needed by high-performance processors requires the use of multiple memory chips in parallel, resulting in substantial amounts of total memory per node. Another reason is that the distributed-address-space programming model that is common in today's large-scale parallel machines severely limits the ability of a processor to efficiently access memory that is not local to it. Such a model also makes fine-grained parallel computation less attractive because of the large fixed costs associated with exchanging data between processors. A final hardware reason is the relative costs of processors and memory. It makes little sense, for example, to place $50 worth of memory on a $1000 node. A machine with 4 times as much memory would not cost significantly more and would be much more versatile. Many of these reasons may disappear, however, due to continually improving technology and integration levels. Within a decade, we are likely to see chips with more than 100 million transistors each [5]. This will allow processors, caches, and memory to reside on the same chip. Decisions about how to partition the transistors on a chip among processor, cache, and memory will then involve entirely different tradeoffs. The data presented in this paper show that fine-grain machines should be seriously considered, since applications can use them effectively.

Table 1: Important application growth rates.

Application	Data	Ops	Concurrency	Communication	Important Working Set
LU	n^2	n^3	n^2	$n^2\sqrt{P}$	const.
CG	n^2	n^2	n^2	$n\sqrt{P}$	const.
FFT	n	$n \log n$	n	$n \log P$	const.
Barnes-Hut	n	$\frac{1}{\theta^2} n \ log \ n$	n	$n^{1/3}\theta^3 p^{2/3} log^{4/3} p$	$\frac{1}{\theta^2} \log n$
Volume Rendering	n^3	n^3	n^2	n^3	n

Table 2: Summary of important application parameters (DS is total data set size).

	Cache		Memory	
Application	Growth Rate	Size for 1G prob on 1K P	Growth Rate	Desirable grain size
LU	const.	8K	const.	$<$ 1M
CG	const.	5K	const.	1M
FFT	const.	4K	const.	1M
Barnes-Hut	$\log DS$	45K	const.	$<$ 1M
Volume Rendering	$\sqrt[3]{DS}$	70K	$\sqrt[3]{DS}$	$<$ 1M

On the software side, reasons for coarser grain nodes include support for a sophisticated node operating system, support for multiprogramming, and the flexibility to run applications with limited parallelism more effectively. However, these capabilities are not necessarily as important on large-scale machines as they are on small ones.

In summary, the grain size issue is a complex one. In this paper, we have taken an applications-oriented view; the other issues must also be taken into account to reach more definitive conclusions about how to actually build large-scale parallel machines. We are currently exploring these other tradeoffs. Overall, it may turn out that designs that split the cost equally between processors and memory will be the most competitive, in that they will be within a small constant factor of the optimal design for any given application.

9 Concluding Remarks

We have presented an application-driven study of issues relevant to determining the appropriate distribution of resources among processors, cache, and main memory for large-scale multiprocessors. We first showed that all of the application classes we studied have a hierarchy of working sets, each of whose size, performance impact and scaling properties we identified. Our conclusion is that relatively small (in some cases trivially small) caches suffice for all the applications. One reason for this is the bimodality in the working sets of applications: The working sets are either very small, so that small caches suffice, or too large to be expected to fit in caches. Fortunately, the small working sets have the most impact on performance.

Next, we examined certain other important characteristics of the computations—communication, computation, data requirements, concurrency, and load balancing behavior—to reflect upon desirable grain sizes for machines to support these computations effectively. We found that relatively fine-grained machines, with large numbers of processors and small amounts of cache and memory per processor, are appropriate for all of the applications.

References

[1] David H. Bailey. FFTs in External or Hierarchical Memories. *Journal of Supercomputing*, 4:23–25, 1990.

[2] Geoffrey Fox et al. *Solving Problems on Concurrent Processors, Volume I: General Techniques and Regular Problems*. Prentice Hall, 1988.

[3] Lars Hernquist. Hierarchical N-body methods. *Computer Physics Communications*, 48:107–115, 1988.

[4] H.T. Kung. Memory requirements for balanced computer architectures. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, 1986.

[5] Gordon Moore. VLSI: Some fundamental challenges. *IEEE Spectrum*, pages 30–37, April 1979.

[6] Jason Nieh and Marc Levoy. Volume rendering on scalable shared-memory MIMD architectures. In *Proceedings of the Boston Workshop on Volume Visualization*, October 1992.

[7] John K. Salmon. *Parallel Hierarchical N-body Methods*. PhD thesis, California Institute of Technology, December 1990.

[8] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Implications of hierarchical N-body techniques for multiprocessor architecture. Technical Report CSL-TR-92-506, Stanford University, 1992.

[9] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *IEEE Computer*, 26(7), July 1993. To appear. Also Stanford Univeristy Tech. Report no. CSL-TR-92-541, 1992.

[10] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John L. Hennessy. Load balancing and data locality in hierarchical N-body methods. *Journal of Parallel and Distributed Computing*. To appear. Prelim. version available as Stanford Univeristy Tech. Report no. CSL-TR-92-505, Jan. 1992.

[11] R. van de Geijn. Massively parallel LINPACK benchmark on the Intel Touchstone Delta and iPSC/860 systems. Technical Report CS-91-28, University of Texas at Austin, August 1991.

[12] Charles van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM, 1992.