Future Applicability of Bus-Based Shared Memory Multiprocessors

C.R.M. Sundaram and Derek L.Eager *

Abstract

Bus-based multiprocessors currently provide a cost-effective approach to achieving modestly parallel execution of large applications. Key (high level) parameters of such systems include the number of processors, the processor speed, the bus bandwidth, and the size of the caches. It is well-known that bus-based systems are not scalable in terms of the number of processors, as that parameter is tightly constrained by limitations of the bus architecture. As technology advances, however, the other key parameters (processor speed, bus bandwidth, cache size) will evolve, quite possibly to differing extents depending on the characteristics of the underlying technologies.

This paper addresses the issue of how quickly bus bandwidth and cache size must scale, as processor speed scales, if bus-based multiprocessors are to be useful platforms for parallel computing in the future. For this purpose, we study the execution characteristics of a number of data parallel numerical and scientific applications, under a time-constrained scaling model, as system parameters are scaled.

Our results indicate three types of applications. For Type I applications, parallel processing on future bus-based machines can be worthwhile only if either bus bandwidth scales as fast as processor speed, or if caches scale fast enough to eventually contain entire application data sets and appropriate affinity scheduling policies are employed, or a physically distributed memory architecture is employed. For Type II applications, parallel processing will be worthwhile regardless of the evolution of the memory system, assuming moderate scalings of bus bandwidth. For Type III applications, bus-based machines will become useless as a platform for parallel computing, unless the bus bandwidth scales as fast as the processor speed.

1 Introduction

Small-scale bus-based multiprocessors have become increasingly common over the past 10 years. Although such systems are perhaps most commonly employed for improved throughput capacity (by allowing multiple jobs to be processed concurrently), they also provide

*The authors are with the Department of Computational Science, University of Saskatchewan, Saskatoon, SK S7N 0W0, Canada.

a low-cost avenue to efficient (albeit modest in scale) shared-memory parallel computing. Concurrent with the increase in popularity of these systems, however, has been dramatic increases in processor speeds, coupled with somewhat smaller improvements in bus bandwidth [9]. A natural question to consider is whether the technologically driven scalings of processor speed, bus bandwidth, and cache size will be such that future systems of this type will be useful for parallel computing. Alternatively, it may be the case that bus bandwidth will become a limiting factor to performance, and that other communication architectures (such as ring networks) will become mandatory even in small-scale parallel computing systems. Note that this study is concerned only with the applicability of the shared bus architecture, and not with the applicability of the shared memory paradigm that it (as well as other architectures) can support.

This paper addresses the question of the future applicability of bus-based multiprocessors, by studying the execution characteristics of a number of data parallel numerical and scientific applications, as the system parameters of processor speed, bus bandwidth, and cache size are scaled (holding the number of processors fixed). We assume that the more powerful systems of the future will be used to run larger problems, or obtain results of improved accuracy, rather than to simply solve the same problems in less time. Thus, a "time-constrained" scaling approach [14] is adopted, in which application parameters are scaled concurrently with the system parameters in such a way that the execution times before and after scaling the application and the machine are equal. Our focus is on "asymptotic" results; specifically, as the machine and the application are scaled, will the application eventually become communication bound (and thus the machine become unsuitable as a parallel computing platform for the application)?

Increasing disparities between processor speed and bus bandwidth can be addressed by a number of hardware and software techniques. Most of these, however, including relaxed memory consistency models[2], prefetching techniques[10], and multiple hardware contexts [1], have no impact on whether an application will become communication bound asymptotically under scaling. Choice of scheduling discipline can, however, impact on the asymptotic results. We consider both "affinity scheduling" techniques (based on static partitioning of data sets, or the techniques in [9], for example) in which computation is scheduled so as to maximize data "locality", and scheduling in which no attention is paid to locality. The assumed main memory architecture also impacts our asymptotic results. In our baseline architecture, we assume physically shared memory, but we also examine the impact of physically distributing memory among the processors.

Our results indicate three types of applications. For Type I applications, parallel processing on future bus-based machines can be worthwhile only if either bus bandwidth scales as fast as processor speed, or if caches scale fast enough to eventually contain entire application data sets and appropriate affinity scheduling policies are employed, or a physically distributed memory architecture is employed. For Type II applications, parallel processing will be worthwhile regardless of the evolution of the memory system, assuming modest scalings of bus bandwidth. For Type III applications, bus-based machines will become useless as a platform for parallel computing, unless the bus bandwidth scales as fast as the processor speed.

The remainder of the paper is organized as follows. In Section 2, previous related work on scaling is described, and various scaling methodologies are outlined, including that employed here. Section 3 presents the applications that are considered in this work, and the manner by which the parameters of each are scaled. Section 4 presents our scaling analysis and the results of that analysis. Finally, in Section 5, conclusions are presented.

2 Scaling

There has been much previous work on scalability [3-8,11-14,17-21], although all of this work focuses on large-scale shared memory multiprocessors and message passing machines, and all but [4] is concerned with scaling the system size (number of processors). In contrast, we consider small-scale machines, and keep the number of processors fixed while scaling the processor speed and other parameters.

Rothberg et al. [12] considers how cache and memory size requirements scale as the machine size (number of processors) and the problem size scale for various applications on distributed shared memory machines. Singh et al. [14] proposes a new methodology for scaling the parameters of scientific applications which simulate physical phenomena, based on the objective of minimizing the overall simulation error. Subramanian et al. [18] analyze the scaling of different types of overhead such as latency, contention, synchronization, and algorithmic overhead, as the machine size is scaled for a fixed problem size. From the overhead growth functions, they analyze the scalability of a few computational kernels on large-scale shared memory machines. Nussbaum and Agarwal [11] define the scalability of an architecture with respect to a specific algorithm for a given problem size as the ratio of the asymptotic speedups achievable on the architecture under consideration to that achievable on an ideal EREW PRAM (Exclusive Read and Exclusive Write Parallel Random Access Machine). Based on this definition, they evaluate the scalability of meshes and hypercubes for a few simple algorithms. Sun and Rover [20] determine the factor by which the problem size must be scaled, if the application is to be executed with a constant average processing rate (in MFLOPS) as the system size is scaled. Gupta and Kumar have carried out a number of scaling studies. In [3], they consider the scalability of Fast Fourier Transform (FFT) on hypercubes and meshes, and analyze by how much the problem size must be scaled, if FFT is to be executed with a constant efficiency as the machine size is scaled. In [5], the same authors study the effects of overhead function and the degree of concurrency on performance measures such as speedup, execution time, and efficiency, and determine the optimal number of processors to be used to minimize the parallel execution time and to achieve a more general criterion of optimality. In [4], they consider the effect of technology dependent factors such as processor speed, and communication speed, on the scalability of parallel shortest path algorithms and matrix multiplication algorithms on hypercube and mesh architectures Finally, in [7], the same authors provide a survey of a number of scalability metrics and discuss their interrelationships.

An important issue in scaling studies is that of determining how (if at all) application parameters are to be varied, as the machine parameters of interest are scaled. From the previous work in this area, a number of different methodologies for scaling application parameters can be identified, including memory-constrained scaling, time-constrained scaling, and constant problem size scaling [14].

In constant problem size scaling, the processing demands of the application are held constant while the machine parameters are scaled. Such a methodology is not appropriate for our work, in which processor speeds, bus bandwidths and cache sizes are being scaled, as attention would be focussed on cases of smaller and smaller application execution times (and thus of less and less interest in terms of the need for parallel processing), as scaling proceeds. It is more appropriate for our work to assume that higher performance machines will be used to run applications with larger processing demands.

In memory-constrained scaling, the application is scaled in such a manner that its data set always fills the available memory. Although memory-constrained scaling could be applied in our work (assuming that the system memory size was somehow scaled along with the other machine parameters), it is more naturally applied in studies that scale the number of processors, under the assumption that the total system memory increases proportionally. Memory-constrained scaling also suffers from the disadvantage that, for some applications, execution time scales more than linearly with the data set size. Thus, execution times may become unreasonably large as scaling proceeds.

Finally, in time-constrained scaling, as the machine parameters are scaled the processing demands of the application are correspondingly scaled, so that the application execution time remains fixed. Such a scaling methodology assumes that for any particular application, there is some fixed range of execution times of interest, below which further decreases in execution time offer little benefit to the end user, and above which delays are impractically large. Time-constrained scaling is adopted in this work.

An additional issue that arises in time-constrained scaling is that of precisely how application parameters will be scaled. Consider in particular scientific applications that simulate physical phenomena. Such applications could be scaled either by looking at larger physical systems (e.g., a larger pipe in some fluid-flow application), or by looking at the same physical systems, but with refined accuracy. The latter form of scaling has been argued as being the most practically relevant [14]. This scaling requires not only scaling the data set size (reflecting, for example, refined grids, or inclusion of smaller, previously neglected particles within the physical system), but also scaling the temporal discretization parameter and other application parameters that impact accuracy. In this work, we adopt the scaling methodology proposed by Singh et al. [14] for such applications, in which the various application parameters are scaled so as to ensure that all sources of error decrease at the same rate. For other applications that we consider (simple numerical routines), we simply scale the data set size.

3 Applications

We consider both numerical kernels and complete scientific applications. The numerical kernels considered include Fast Fourier Transform (FFT), Conjugate Gradient, and LU-Factorization. FFT forms the computational core of a wide variety of applications from the fields of image and signal processing. Conjugate Gradient is a representative of the class of iterative methods for solving systems of equations, while LU-factorization is a representative of the class of direct methods. Complete scientific applications are selected from the SPLASH benchmark suite [16], and include Barnes-Hut, Fast Multipole, Ocean, and Water. Each of these applications is described briefly below. (Our descriptions assume use of scheduling techniques designed to maximize locality, although in our analysis, we will also consider the case in which such techniques are not used.)

FFT computes the Fourier Transform of a set of data points. In a parallel implementation based on the Cooley-Tukey serial algorithm [3], there is an outer sequential loop of size $\log n$ (where n is the number of data points), enclosing an inner parallel loop of size n. There are no dependencies among the iterations of the inner parallel loop. In iteration s of the outer sequential loop, data points separated by a distance of $\frac{n}{2^s}$ interact with each other. The data points are partitioned across the processors in such a way that each processor computes the FFT of a contiguous set of data points. With such a partitioning, the last $\log n - \log N$ (where N is the number of processors) outer sequential loop iterations do not involve any communication among the processors and in the first $\log N$ iterations, all the n data points are exchanged among the processors.

In Conjugate Gradient, an initial guess of the solution to a linear system of equations is iteratively improved. In each iteration, the dominant part of the computation is a sparse matrix-vector multiply. In this study, we consider a sparse matrix whose graph representation is simply a two dimensional grid. Each processor handles a square subgrid, exchanging the boundary points with the processors responsible for neighbouring subgrids in every outer sequential loop iteration. Note that the number of iterations required for convergence is relatively insensitive to the size of the system of equations.

In LU-factorization, a given matrix A is decomposed into the form $A = LU$, where L and U are lower and upper triangular matrices, respectively. We consider here blocked LU-factorization, in which the given matrix A (n by n) is divided into a K by K array of B by B blocks ($n = KB$). There is an outer sequential loop of size K, enclosing an inner parallel loop. Each iteration of the inner parallel loop is simply a blocked matrix multiplication. In each iteration of the outer sequential loop, each processor handles the same blocks. In the entire execution, K^2B^2 data points are exchanged among the processors. Thus, the total computation scales as n^3, and the communication scales as n^2.

Both Barnes-Hut and Fast Multipole simulate the evolution of a system of bodies under the influence of gravitational forces. The positions and velocities of the bodies are recorded in an octree representation. Both the applications proceed over numerous time steps. In Barnes-Hut, every time step involves several phases such as building the tree, computing the force acting on every body due to the influence of other bodies, updating the properties of the bodies, partitioning the bodies among the processors for the next time step, and so on. The most time consuming phase is the force calculation phase, which is parallelized over the number of bodies. The bodies are partitioned across the processors in such a way that bodies assigned to the same processor are contiguous in the physical space, and each processor has roughly the same amount of work. Although the partitioning is potentially changed at every time step to reflect the dynamic movement of the bodies, nonetheless each processor can expect to be assigned much the same bodies for consecutive time steps (assuming reasonably sized steps).

As with the other scientific applications, we wish to scale Barnes-Hut so as achieve greater and greater accuracies, under a constant execution time constraint, as machine parameters are scaled. The number of bodies can be scaled so as to reflect the inclusion of smaller, previously neglected bodies within the physical system being modelled. It is also necessary, however, to scale the force accuracy parameter that determines the distance over which interactions with individual bodies are computed. Eventually, this distance will span the entire physical space being modelled, and thus the force calculation will become $O(n^2)$, where n is the number of bodies. Assuming that each processor need fetch the data for each body not in its partition only once during each force calculation phase, the total amount of communication is $O(n)$. (An alternative would be to limit the value of the force accuracy parameter in such a manner so as to limit the force calculation computation to only $O(nlogn)$ time [13]; however, increasing accuracy would then require consideration of successively higher moments, a possibility we do not consider.)

In Fast Multipole, every time-step involves similar phases as in Barnes-Hut, but, the force computation phase is parallelized across cells (each cell containing a large number of bodies), rather than across bodies. The computation time is linear in the number of cells (hence in the number of bodies), unlike the situation in Barnes-Hut. Although the partitioning of cells is changed in every time step to reflect the dynamic movement of the bodies, nonetheless each processor can expect to be assigned much the same bodies for consecutive time steps, as in Barnes-Hut.

The Ocean application studies the role of eddy and boundary currents in influencing large-scale ocean movements. A cuboidal ocean basin is simulated. The application proceeds over numerous time steps. Each time step involves setting up and solving a set of spatial partial differential equations on two dimensional fixed-size grids that represent the horizontal cross sections of the ocean basin. The grids are partitioned into square subdomains that are permanently assigned to a specific processor. Each processor updates only grid points in its own subdomain, but may read boundary points from adjacent subdomains. The updates of grid points and reading of boundary points by adjacent processors occur in different phases separated by synchronization.

The Water application simulates the motion of water molecules in liquid state. The application proceeds over numerous time steps. In each time step, the most time consuming phase involves computing all the pairwise interactions among the water molecules. The molecules are statically partitioned among the processors. The computation is blocked with respect to remote references by performing all the interactions for a molecule fetched from another partition before fetching the next molecule. The computation grows as $O(n^2)$, and the communication grows as $O(n)$, where n is the number of molecules.

All of the above applications have a very similar computational structure. There is an outer sequential loop (over the time steps in the scientific applications) enclosing an inner parallel loop, which is over the data elements of the application. This type of computational structure appears to be common among many other scientific and engineering applications as well [9,16]. Further, the computational cost of the parallel loop contributes the dominant term in the overall execution time complexity of these applications. As the problem size and the machine speed scale, the execution of the inner parallel loop will consume a larger and larger fraction of the total execution time. Thus, as far as the asymptotic results are concerned, other parts of the application cannot have an impact, and we do not consider them in our analysis.

The scaling of application parameters (for the scientific applications, done so as to ensure that all sources of error decrease at the same rate), results in scalings of the following high level application characteristics: the computation per data element (t_C), the number of outer loop iterations I, and the data set size D. Table I summarizes the relationship between these parameters, by indicating the factors by which the first two parameters must change for a given change in the latter. Here, Δt_C, ΔI, and ΔD denote the ratio between the new value and old value of the computation per data element, number of outer loop iterations, and the data set size respectively. These expressions have been derived from results in [14], For the scientific applications, and for the numerical kernels from an understanding of their computational structure.

Application	Δt_C	ΔI
Barnes-Hut	$\sqrt{\Delta D}$	$\left(\Delta D \right)^{0.25}$
Ocean	1.0	ΔD
Fast Multipole	$\log^2_{1.828} \sqrt{\Delta D}$	$\left(\Delta D \right)^{0.25}$
Water	ΔD	1.0
Fast Fourier Transform	1.0	$\log \left(\Delta D \right)$
Conjugate Gradient	1.0	1.0
LU-Factorization	1.0	$\sqrt{\Delta D}$

Table I: Scaling Factors for the Change in Computation per Data Element and Number of Outer Loop Iterations as Functions of the Change in Data Set Size.

4 Analysis and Results

Our principal concern is with how quickly bus bandwidth and cache size must scale, as processor speed scales, if bus-based multiprocessors are to be useful platforms for parallel computing in the future, and with the impact on these results of use of a distributed rather than shared physical memory architecture. Given particular scaling factors for processor speed, bus bandwidth, and cache size, and a particular application, the key question is whether or not that application will eventually become communication bound (in the sense of the time spent on the bus consuming all but a negligible proportion of the execution time). If it does, then parallel processing will eventually yield no benefit, as there is no parallelism in communication on a bus-based architecture, and thus a communication bound application should be expected to run as fast or faster on a suitable single processor system.

In order to be able to address this question, we need to determine how the application parameters must be scaled, given specified scaling factors for the machine parameters of processor speed, bus bandwidth, and cache size. Table I ties all the high-level application parameters together, so that we need only determine how the application data set size must scale. Recalling that our scaling methodology specifies that the application must be scaled so as to yield constant execution time, this one unknown can be determined by equating an expression for application execution time for the original machine, with that for the scaled machine.

Predicting execution time requires predicting the volume of bus traffic that would be generated by the application. In our baseline architecture, main memory is physically shared. When affinity scheduling techniques are employed, the data set of the application is essentially partitioned among the processors (either explicitly or implicitly). In this case, the bus traffic can be divided into three categories: traffic resulting from the first access by each processor to data belonging to its partition ("cold-start" misses), traffic resulting from accesses to data within the processor's partition, but which has been evicted from its cache to make room for other data ("capacity" misses), and traffic resulting from the inherent communication in the application ("coherence" misses, such as the retrieval of boundary elements in Ocean). Although we assume a direct-mapped cache organization, we do not consider "conflict" misses (which result from the limited associativity of the cache), as we make the assumption that each processor's partition of the data set is stored sequentially in memory. (This assumption has no impact on our asymptotic scaling results, however.)

We also consider the case in which affinity scheduling techniques are not employed. It is assumed in this case that the portion of the application data set operated on by each processor, during each iteration of the outer sequential loop of the application, is mostly different from that which it operated on during the last iteration of the outer sequential loop, and thus must be freshly retrieved from memory.

Section 4.1 considers our key question of whether the application of interest will eventually become communication bound (asymptotically) as machine parameters are scaled, in the case of a physically shared memory. As the answer to this question is often "yes", it is natural to wonder as to whether there are architectural modifications that could improve the situation. As noted in the introduction, techniques such as multiple hardware contexts and relaxed memory consistency models can not change the nature of our asymptotic results. One architectural modification that may have an asymptotic impact, however, is to distribute the global memory among the processors (while retaining the basic bus-based communication architecture). Assuming that only accesses to data located in a non-local memory module must use the bus, in this design only the communication inherent to the application (the coherence misses) results in bus traffic. Section 4.2 contains the asymptotic scaling analysis and results for this case. Finally, in Section 4.3, we consider the question of how quickly some of our asymptotic results from Section 4.1 might be expected to take hold.

4.1 Principal Asymptotic Scaling Results

Recall that in each of the applications we consider the most computationally intensive part is a parallel loop that is nested inside an outer sequential loop. It is this part of each application that we attempt to model.

Let t_c denote the original (before scaling the application parameters) computation per data element accessed in the inner parallel loop, t_{miss} denote the original miss cost per data element accessed in the inner parallel loop that is not found in the cache, I denote the original number of outer loop iterations, and D denote the original data set size. We ignore some constant factors in this analysis, as these do not impact our asymptotic results. In particular, we make no distinction between the entire data set size, and the size of that portion of the data set that will be operated on by each processor in each sequential loop iteration (recall that the number of processors is fixed), nor do we consider bus queueing delays (these are upper bounded by a constant factor owing to the fixed number of processors), nor do we make any distinction between I and I-1. Thus, the original application execution time can be written as,

$$\left(t_c + f * t_{miss} \right) * D * I + t_{miss} * D + t_{miss} * \mathrm{Coh}(D) * I$$

The first term is due to computation plus the cost of bringing the portion of the data set operated on by each processor, during each iteration of the outer sequential loop, into the cache (if necessary). Here f is 1 if affinity scheduling is not employed, or if affinity scheduling is employed but the size of the portion of the data set operated on by each processor is more than twice the cache size (in which case, assuming that each processor accesses its data in the same order in each sequential loop iteration, none of its data will be found in its cache). If affinity scheduling is employed, and the size of the portion of the data set operated on by each processor is less than twice the cache size, but more than the cache size, then f is given by $\frac{2*(r-1)}{r}$, where r is the ratio of the size of the data set portion to the cache size. Otherwise, f is 0. The second term reflects cold-start misses. Finally, the third term reflects the coherence misses, where Coh(D) denotes the number of coherence misses per iteration of the outer sequential loop, as a function of data set size.

We suppose that the machine processor speed is now scaled by a factor of p, and express other scalings relative to this. Bus bandwidth is assumed to scale by a factor p^b and cache size by a factor p^c. There is no benefit for the applications considered, to scale the cache size and the bus bandwidth faster than the processor speed. Further, it has been observed that bus bandwidth is likely to scale slower than the processor speed [9,23]. Thus, in the following analysis, we restrict ourselves to the region $b \leq 1, c \leq 1$. With respect to the application parameters, recall that it need only be determined how the data set size scales, as the scalings of the other application parameters can then be determined from Table I. The scaling of the data set size is denoted by the factor p^a, where a is an unknown that must be determined by equating the original execution time to the execution time on the scaled machine.

For Barnes-Hut, from Table I it is seen that if data set size scales by a factor p^a, then $\mathrm{t_C}$ must scale by a factor $p^{\frac{a}{2}}$, and I must scale by a factor of $p^{\frac{a}{4}}$. As noted previously, the total required communication is $\mathrm{O}(n)$, where n is the number of bodies, and thus the coherence misses scale by a factor identical to the scaling of the data set size (p^a). Denoting the value of f on the scaled machine (unknown in the case of affinity scheduling) by f', execution time on the scaled machine is then given by,

$$\left(\frac{\mathrm{t_{C*p^{\frac{a}{2}}}}}{p} + \mathrm{f}' * \frac{\mathrm{t_{miss}}}{p^b} \right) * \mathrm{D} * I * p^{\frac{5a}{4}} + \frac{\mathrm{t_{miss}}}{p^b} * \mathrm{D} * p^a + \frac{\mathrm{t_{miss}}}{p^b} * \mathrm{Coh(D)} * I * p^{\frac{5a}{4}}$$

In the above expression, each term represents the scaled portion of the execution time. For example, the first term represents the scaled portion of the execution time corresponding to the computation involved in the data elements, and the last term represents the scaled portion of the execution time due to fetching the coherence misses.

Determination of whether the application is asymptotically communication or computation bound, rests on which term in the above expression dominates, under the constraint that the overall execution time must match the original execution time. Collecting terms and simplifying, we obtain,

$$\text{Computation time on the scaled machine} = \left(\mathrm{t_C} * p^{\frac{7a}{4}-1} * \mathrm{D} * \mathrm{I} \right)$$

$$\text{Communication time on the scaled machine} = \begin{pmatrix} \mathrm{f}' * \mathrm{t_{miss}} * p^{\frac{5a}{4}-b} * \mathrm{D} * \mathrm{I} + \\ \mathrm{t_{miss}} * p^{a-b} * \mathrm{D} + \\ \mathrm{t_{miss}} * p^{\frac{5a}{4}-b} * \mathrm{Coh(D)} * \mathrm{I} \end{pmatrix}$$

It is now necessary to consider $p \to \infty$, and to find the value of a that ensures that the execution time matches the original execution time. (In this asymptotic analysis, all that actually matters is that the execution time must neither shrink to zero, nor grow unbounded.) Most often, at that value of a either the computation or the communication time will dominate, and the other will be negligibly small. The details of this analysis are described below for Barnes-Hut. Similar expressions as those above, for the other applications, are provided in the appendix, and form the basis of analogous analyses.

Define a^* as $\lim_{p\to\infty} a$. For Barnes-Hut, note that a^* must be at most $\frac{4}{7}$, as otherwise, the computation term would grow with p, and the overall execution time could not match the original execution time. Similarly, a^* must be at most $\frac{4b}{5}$, as otherwise, the communication term would grow with p. Note, however, that a^* must take on one of these values, as otherwise (if it were less than both), the overall execution time would decrease with p. Thus, we have, regardless of whether or not affinity scheduling techniques are employed,

$$a^* = \min\left(\tfrac{4}{7}, \tfrac{4b}{5} \right)$$

Which term is the minimum in this expression determines whether Barnes-Hut will be asymptotically computation bound or communication bound. Table II presents the results for Barnes-Hut, and gives corresponding results for the case in which affinity scheduling techniques are not employed, for the other applications as well. Note that all these results are independent of how cache size scales.

Application	Computation Bound	Communication Bound
Barnes-Hut	b $> \frac{5}{7}$	b $< \frac{5}{7}$
Ocean	b $>$ 1.0	b $<$ 1.0
Fast Multipole	b $>$ 1.0	b $<$ 1.0
Water	b $>$ 0.5	b $<$ 0.5
Fast Fourier Transform	b $>$ 1.0	b $<$ 1.0
Conjugate Gradient	b $>$ 1.0	b $<$ 1.0
LU-Factorization	b $>$ 1.0	b $<$ 1.0

Table II: Computation Bound and Communication Bound Intervals with no affinity scheduling.

Results for other applications frequently depend on whether or not affinity scheduling techniques are used. In these cases, to obtain results for affinity scheduling, it is necessary to derive the value of f' from the relative scalings of the cache and data set sizes. This is also necessary for an application like Barnes-Hut, if one wishes to determine which category of bus traffic is responsible for most of the execution time, in cases where the application becomes communication bound, and affinity scheduling techniques are being employed.

Figure 1 shows the results of such an analysis for Barnes-Hut. As determined previously, if $b > \frac{5}{7}$, then the application becomes computation bound. This case corresponds to regions B and C in the figure. Note that in region B, the data set eventually fits in the processor caches (as c is greater than a), and f' must become 0. In region C, in contrast, the data set size becomes much bigger than the cache size, and thus f' must become 1.

In regions A and D, Barnes-Hut is asymptotically communication bound. The only issue is whether the bus time spent servicing coherence misses, or that spent servicing capacity misses as well as coherence misses, dominates the execution time. (The bus traffic due to coherence misses scales at least as fast as that due to capacity misses for this application, while the bus traffic due to cold start misses is strictly less than that due to coherence misses.) Consider region D, in which $c < \frac{4b}{5}$. Since we know that in regions A and D, a^* must equal $\frac{4b}{5}$, in region D the data set size is scaling faster than the cache size, and f' must become 1. Thus, in region D, capacity misses are scaling as fast as coherence misses. In region A, on the other hand, the data set size is scaling more slowly than the cache size, and f' must become 0, implying that in this region, coherence misses dominate.

Figures 2-7 show corresponding results, for the case where affinity scheduling techniques are employed, for Fast Multipole, Ocean, Water, Conjugate Gradient, FFT, and LU-Factorization. The expressions for the computation and the communication terms are given in the Appendix. In regions B and C (should they exist) of each figure, the application is asymptotically computation bound. In region B, the data set eventually fits within the processor caches, while in region C, it does not. In regions A,D, and E (should they exist) the application is asymptotically communication bound. The application data set eventually fits within the processor caches in region A, and the time spent on coherence misses and/or cold-start misses dominates the execution time. In region D, the data set never fits within the caches, and the time spent on capacity misses (and in some cases also coherence misses and/or cold-start misses) dominates the execution time. In region E (which was not present in the Barnes-Hut application), the data set size asymptotically tracks the cache size, but always marginally exceeds it, and the time spent on capacity misses dominates the execution time. Within this region, $a^* = c$, and f' is a function of p with limit 0, but with value between 0 and 1 for finite p.

Figures 1-7 indicate three basic types of applications, which we term Types I, II, and III. Applications of Type I (Ocean, Fast Multipole, and LU-Factorization) are characterized by the presence of the B region, but not the C region. These applications can be executed efficiently on future bus-based machines only if either bus bandwidth scales as fast as processor speed, or if caches scale fast enough to eventually contain entire application data sets and appropriate affinity scheduling policies are employed. The extent of required cache size growth varies depending on which application is considered, ranging from a factor of $p^{\frac{1}{2}}$ for Ocean (i.e., as the square root of the factor by which processor speed scales), to $p^{\frac{4}{5}}$ for Fast Multipole.

Applications of Type II (Barnes-Hut and Water) have both a B and a C region. For these applications, affinity scheduling techniques are not worthwhile in terms of the basic issue of whether the application will become computation bound or communication bound. (Of course, if the application is communication bound, then affinity scheduling techniques would allow larger problems or greater accuracies to be obtained for the same execution time, in those cases in which capacity misses would otherwise dominate the bus traffic, and in which the data set would be capable of fitting in the caches. But, if the application is communication bound, then there is no benefit to parallel processing on a bas-based architecture in any case.) These applications can be executed efficiently on future bus-based machines as long as the bus bandwidth scales by particular factors (for Water, by the square root of the factor p by which processor speed scales, and for Barnes-Hut, by a factor of $p^{\frac{5}{7}}$).

For Type III applications (FFT and Conjugate Gradient), both B and C regions are absent. For these applications, bus-based machines will become useless as a platform for parallel computing unless the bus bandwidth scales as fast as the processor speed. Poor scaling performance of FFT on distributed shared memory and message passing machines has been noted in [3] and [12]. Gupta and Kumar [3] note that FFT can be executed on hypercube architectures with constant efficiency as the number of processors is scaled only if the problem size is increased very rapidly. Rothberg et al. [12] note that FFT demands an exponential growth of problem size as the machine size is scaled to maintain a constant computation-communication ratio.

Note that the above results are based on the specific solution algorithms used in the applications considered. A different solution algorithm will change the asymptotic results for a given application, only if the other solution algorithm has a different order of computation or communication time complexity. If the complexities differ in only constant factors, then the asymptotic results will not change. For example, a different implementation of FFT

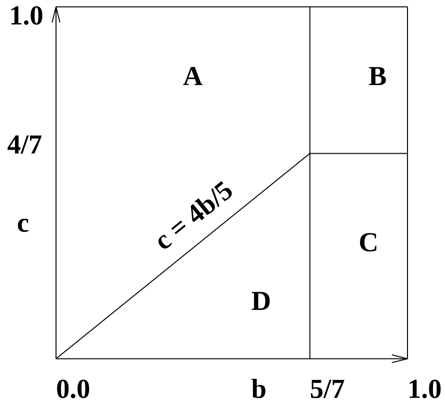

Figure 1: Barnes-Hut
(B,C: Computation Bound Regions,
A,D: Communication Bound Regions)

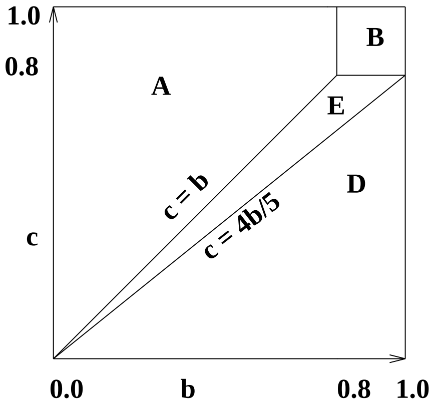

Figure 2: Fast Multipole
(B: Computation Bound Region,
A,D,E: Communication Bound Regions)

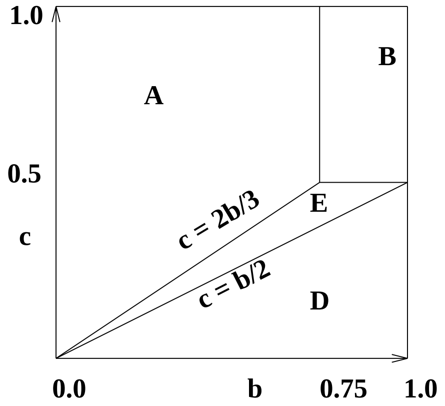

Figure 3: Ocean
(B: Computation Bound Region,
A,D,E: Communication Bound Regions)

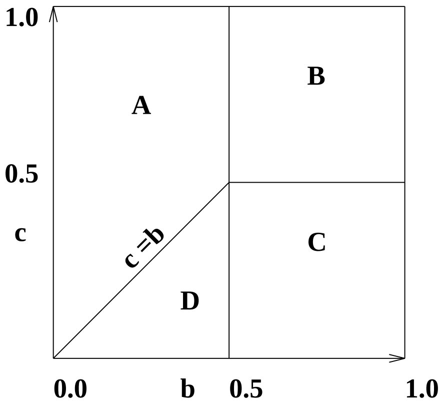

Figure 4: Water
(B,C: Computation Bound Regions,
A,D: Communication Bound Regions)

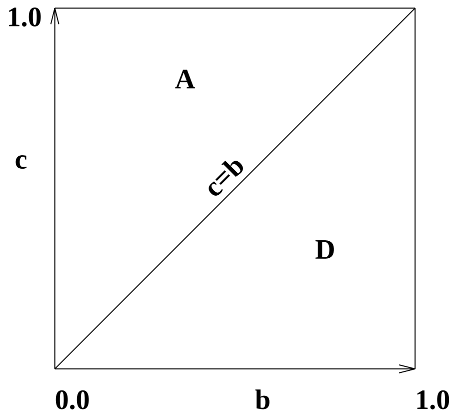

Figure 5: Conjugate Gradient
(A,D: Communication Bound Regions)

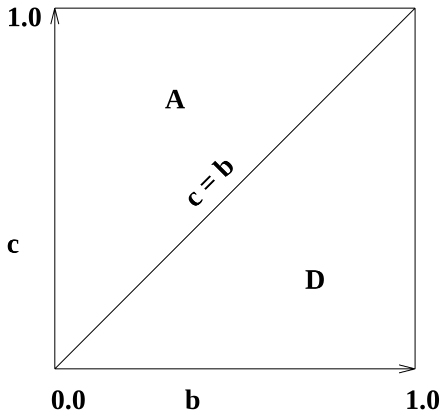

Figure 6: Fast Fourier Transform
(A,D: Communication Bound Regions)

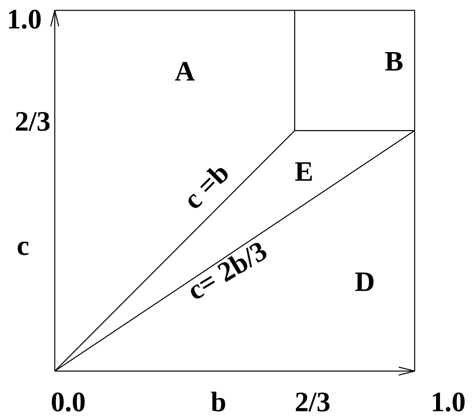

Figure 7: LU-Factorization
(B: Computation Bound Region,
A,D,E: Communication Bound Regions)

based on transposing of matrices [3], does not change the asymptotic results. Whereas, Barnes-Hut and Fast Multipole (two different solution algorithms for n-body problems), have different asymptotic results, because the computation and communication complexities of the solution algorithms have different order.

The classification of an application into one of our three types can be correlated with the application characteristics given in Table I. For applications of Type III, neither the number of outer sequential loop iterations, nor the computation per data element, scales significantly with the data set size. Applications of Type II are characterized by a computation per data element that does scale significantly with the data set size. Finally, applications of Type I are characterized by a computation per data element that does not scale significantly with the data set size, but a number of outer sequential loop iterations that does scale significantly.

4.2 Asymptotic Scaling Results for an Architecture with Distributed Memory Modules

In this section, we consider a modified architecture in which the global shared memory has been distributed among the processors, while retaining the basic bus-based communication architecture. (I/O capabilities may be similarly distributed.) Only accesses to data located in a non-local memory module must use the bus, and thus only communication inherent to the application (coherence misses) results in bus traffic. We assume that the communication bandwidth between the processor and its own local memory module can be scaled up commensurate with the scaling of processor speed. Such an architecture results in minimal demands being placed on the bus. If an application still becomes communication bound even on this architecture, then it seems certain that alternative communication architectures are required.

Applications can potentially exploit the modified architecture only if affinity scheduling techniques are employed. In this case, results may be derived simply by removing the terms reflecting capacity and cold start misses from the communication time expressions previously derived. Table III shows the resulting thresholds on how the bus bandwidth must scale (in terms of the exponent b in the factor p^b, where p is the processor speed scaling factor), for each application to asymptotically become communication or computation bound.

Application	Computation Bound	Communication Bound
Barnes-Hut	$b > \frac{5}{7}$	$b < \frac{5}{7}$
Ocean	$b > \frac{3}{4}$	$b < \frac{3}{4}$
Fast Multipole	$b > \frac{3}{5}$	$b < \frac{3}{5}$
Fast Fourier Transform	$b > 1.0$	$b < 1.0$
Conjugate Gradient	$b > \frac{1}{2}$	$b < \frac{1}{2}$
Water	$b > \frac{1}{2}$	$b < \frac{1}{2}$
LU-Factorization	$b > \frac{2}{3}$	$b < \frac{2}{3}$

Table III: Computation Bound and Communication Bound Intervals

By comparing the results of Table III to those results in Figures 1-7 corresponding to high cache size scaling factors (where the data sets ultimately fit within the cache) it can be seen that physically distributing main memory yields all the benefits of high cache size scaling factors (given the assumptions of our analysis), and, in addition further benefits are achieved for Fast Multipole and Conjugate Gradient. (In the case of Conjugate Gradient, note that the application becomes of Type II on this architecture, rather than of Type III as on the original architecture.) For these applications, once capacity misses are removed (through affinity scheduling techniques and having sufficient local cache), cold start misses dominate. In the modified architecture, this traffic is removed from the bus.

4.3 Transient Results

As shown in Section 4.1, efficient execution of Type I applications on future bus-based, physically shared memory machines is likely only if caches are large enough to contain entire application data sets. It is apparent, however, that on current bus-based machines, application data sets for reasonably-sized problems are much too large to fit in processor caches [12,13]. A natural question to consider is that of how long (roughly) it will take for our asymptotic results to take hold; specifically, for the data sets for reasonably-sized problems to fit in the caches of the future machines, assuming cache size scaling rates that imply that this would eventually occur. (Note that we do not claim that such cache size scaling rates are, or are not, likely). We address this question in this section through a "transient" (as contrasted to asymptotic) analysis of application performance, using the communication time and computation time expressions derived previously. Results are shown only for Ocean; results for the other Type I applications that we consider are qualitatively similar in nature.

We first consider how long it will be (roughly) until the size of each processor cache is one half that of each processor's portion of the application data set. After this point, a portion of the data set starts to be retained in the caches, assuming appropriate affinity scheduling policies. Let P denote the scaling factor for processor speed at which this occurs. Then, since cache size will have scaled by P^c, and denoting by k the factor by which cache size is initially smaller than that of the processor caches, we must have,

$$\Delta \mathrm{D} = \frac{P^c}{\frac{k}{2}} = P^{c - \log_P \frac{k}{2}}$$

Using Table I to determine the corresponding scalings of the computation per data element and the number of outer sequential loop iterations for Ocean, and noting that at the point under consideration we must have $\mathrm{f}' = 1$, we obtain the following expression for the execution time of Ocean after scaling,

$$\left(\mathrm{t_c} * P^{2(c - \log_P \frac{k}{2}) - 1} + \mathrm{t_{miss}} * P^{2(c - \log_P \frac{k}{2}) - b} + \mathrm{t_{miss}} * P^{(c - \log_P \frac{k}{2}) - b} + \mathrm{t_{miss}} * P^{\frac{3(c - \log_P \frac{k}{2})}{2} - b} \right)$$

Note that this expression is approximate; it does not, for example, take bus contention into account. Nonetheless, it is accurate to within small constant factors, which is sufficient considering the nature of the results obtained.

Similarly, we can obtain an expression for the execution time of Ocean after scaling to the point where the size of each processor cache is equal to (rather than one half) that of each processor's portion of the application data set. Noting that we have $\mathrm{f}' = 0$ at this point, assuming that appropriate affinity scheduling techniques are employed, we obtain,

$$\left(\mathrm{t_c} * P^{2(c - \log_P k) - 1} + \mathrm{t_{miss}} * P^{(c - \log_P k) - b} + \mathrm{t_{miss}} * P^{\frac{3(c - \log_P k)}{2} - b} \right)$$

The above expressions can be used to numerically obtain the value of P for each case, for various values of b and c, once one chooses an initial set of machine and application parameters. Specifically, values for $\mathrm{t_c}$, $\mathrm{t_{miss}}$, k, and the initial application execution time are required. The architectural and application parameters given in Table IV (chosen to be reasonably consistent with current technology, but not necessarily reflective of any particular implementation of Ocean) are employed to determine the first two of these parameters. The initial application execution time is left as a free variable, which, together with the parameters in Table IV, determines k.

Note that the parameters in Table IV imply that the application is initially computation bound. From Table IV, 4000 cycles of computation are done on a data element compared to the 960 cycles of communication needed to bring it into a cache. So, initially, the computation time dominates the execution time, although the portion of the application data set owned by each processor is initially about 3.75 times bigger than the cache size even for an application with just a half an hour execution time.

Parameter	Value
Processor Speed	180 MHZ
Cache Size	1 MB
Cache Line Size	16 Bytes
To fetch a Cache Line from Main Memory	60 cycles
Size of a Data Element	256 Bytes
Amount of Computation per Data Element	4000 cycles
Number of Outer Sequential Loop Iterations	2000

Table IV: Architectural and Application Parameters.

Results obtained from our transient analysis are shown in Figures 8 and 9, for various values of b and c chosen to ensure that asymptotically, the application remains computation bound. We have inverted these graphs, in that the processor scaling factor P (which is computed from the above analysis) is on the X-axis, and the application execution time (which is an input parameter to the analysis) is on the Y-axis. Noting that processor speed is increasing exponentially, and that P is shown on a log scale, constant-sized steps along the X-axis correspond to time intervals of constant size. (Given any particular assumption about the rate at which processor speeds are increasing, the correspondence with time can be made precise.) Thus, the graphs show how, as time advances, larger and larger applications will eventually be such that their data sets fit within the processor caches.

In general, the rate at which this happens is quite slow for small cache size scaling rates (e.g., $c = 0.6$), but is quite rapid for large cache scaling rates (e.g., $c = 0.8$). For example, even for a large application execution time of 2.5 hours, the application data set will eventually fit in the caches, as the machine and the application scale up, in well under 20 years (assuming that processor speed doubles every two years), for $c = 1.0$. On the other hand, for $c = 0.6$, it would take on the order of 100 years (an unreasonably long period in terms of the relevance of the analysis) for the cache size to catch up with the data set size, for the same size application and under the same assumption as to the rate of processor speed growth. The rate at which bus speed is scaling has little impact on how long it takes the data set to fit entirely within the caches (Figure 9), but does play a considerable role in determining when the data set begins to partially fit (Figure 8).

Finally, Figures 10 and 11 indicate in part the sensitivity of the results from our transient analysis to the assumed application characteristics given in Table IV. Specifically, these graphs show how the assumed number of cycles of computation per data element impacts the values of P computed from the transient analysis. In all cases, our asymptotic results take hold faster (the computed values of P are smaller) for larger amounts of computation per data element. Note that for large c, however, the variability in P is much less than for small c.

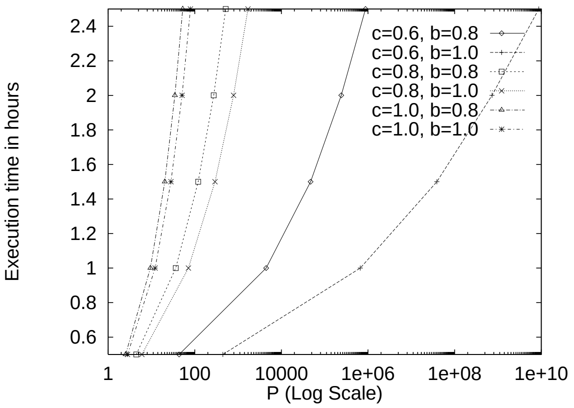

Figure 8: **Application Execution Time for which the Data Set Size will equal twice the Cache Size as a Function of the Processor Speed Scaling Factor**

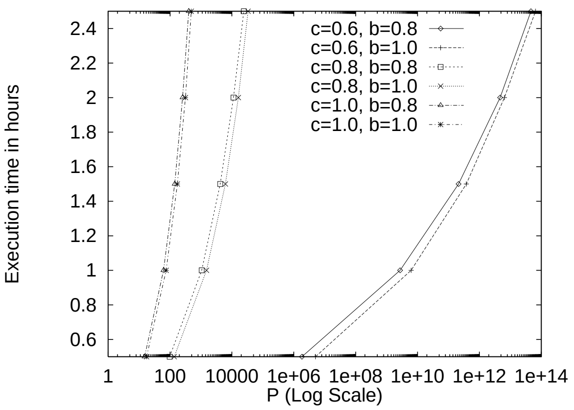

Figure 9: **Application Execution Time for which the Data Set Size will equal the Cache Size as a Function of the Processor Speed Scaling Factor**

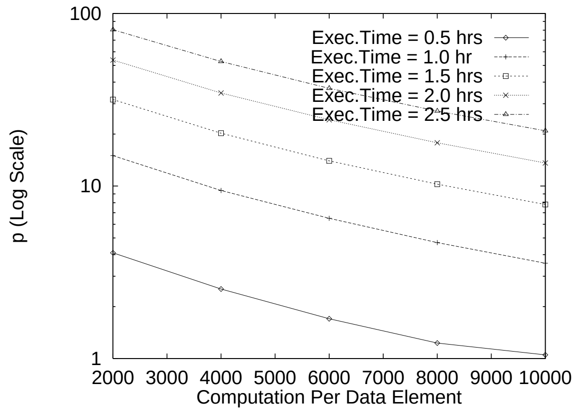

Figure 10: **Processor Speed Scaling Factor at which the Data Set Size Equals twice the Cache Size as a Function of the Computation Per Data Element (c = 1.0, b = 0.8)**

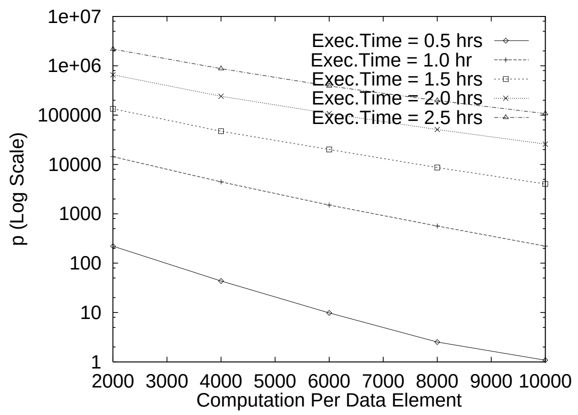

Figure 11: **Processor Speed Scaling Factor at which the Data Set Size Equals twice the Cache Size as a Function of the Computation Per Data Element (c = 0.6, b = 0.8)**

5 Conclusions

This paper has addressed the question of the future applicability of bus-based multiprocessors, by studying the execution characteristics of a number of data parallel numerical and scientific applications, as the system parameters of processor speed, bus bandwidth, and cache size are scaled (holding the number of processors fixed). We assume that the faster systems of the future will be used to do more, rather than to do the same thing in less time, and thus we adopted a "time-constrained" scaling approach, in which application parameters were scaled concurrently with the system parameters in such a manner that the application execution time was held constant.

Our results indicate three types of applications. For Type I applications, parallel processing on future bus-based machines can be worthwhile only if either bus bandwidth scales as fast as processor speed, or if caches scale fast enough to eventually contain entire application data sets and appropriate affinity scheduling policies are employed, or a physically distributed memory architecture is employed. For Type II applications, parallel processing will be worthwhile regardless of the evolution of the memory system, assuming modest scalings of bus bandwidth. For Type III applications, bus-based machines will become useless as a platform for parallel computing, unless the bus bandwidth scales as fast as the processor speed.

Even for applications of Types I and II, however, requirements for bus bandwidth growth were often substantial. The minimum required scaling of bus bandwidth, among the applications considered, was by the square root of the factor by which processor speed scales. For applications such as Fast Multipole, for which the required scaling of bus bandwidth was found to be considerably greater, the required bus bandwidth growth may be unachievable.

Finally, a transient analysis was carried out to determine how quickly the asymptotic results predicted for the Type I applications, for a physically shared memory architecture could be expected to take hold. In particular, we addressed the question of when one might expect the data set of a Type I application of reasonable size to fit within the processor caches, in cases in which our asymptotic results predict that this would eventually occur. Our results indicate that if cache size scales quickly (e.g., as fast as processor speed), then this should be in the foreseeable future; otherwise, it may not be.

Current work is examining a wider array of applications, including applications with dependency patterns among the iterations of the parallel loops. We are also considering a broader class of systems, considering in particular communication performance in more detail, under varying assumptions as to the communication architecture.

Appendix A

In the following, we give the computation time and the communication time on the scaled machine, for all the applications other than the Barnes-Hut application.

In FFT, as described in Appendix A, all the n data elements are exchanged among the processors in the first $\log_2 N$ iterations of the outer sequential loop (where N is the number of processors), while in the remaining $\log_2 n - \log_2 N$ iterations, no communication occurs. Thus, the coherence misses scale with the problem size as $\log_2 N * n$. Together with the sizes of the outer sequential and inner parallel loop, as given in Appendix A, this implies that the computation and communication times are given by,

$$\text{Computation time on the scaled machine} = \left(\ \mathrm{t_C} * p^{a-1+\log_p a+\log_p \log_2 p} * \mathrm{D} * \mathrm{I}\ \right)$$

$$\text{Communication time on the scaled machine} = \left(\begin{array}{l} \mathrm{f}' * \mathrm{t_{miss}} * p^{a-b+\log_p a+\log_p \log_2 p} * \mathrm{D} * \mathrm{I}+ \\ \mathrm{t_{miss}} * p^{a-b} * \mathrm{D}+ \\ \mathrm{t_{miss}} * p^{a-b} * \mathrm{Coh(D)} * \mathrm{I} \end{array} \right)$$

The exponents $\log_p a$ and $\log_p \log_2 p$ come from the fact that the number of outer sequential loop iterations scales as $\log_2 n$, where $n$scales as p^a. The contribution of these exponents towards the execution time tends to zero as p increases by very large values.

For Conjugate Gradient, the computation structure is very similar to that of Ocean. The amount of communication scales as $\sqrt{N}$, where N is the problem size. Thus, the computation and the communication terms are given by,

$$\text{Computation time on the scaled machine} = \left(\ \mathrm{t_C} * p^{a-1} * \mathrm{D} * \mathrm{I}\ \right)$$

$$\text{Communication time on the scaled machine} = \left(\begin{array}{l} \mathrm{f}' * \mathrm{t_{miss}} * p^{a-b} * \mathrm{D} * \mathrm{I}+ \\ \mathrm{t_{miss}} * p^{a-b} * \mathrm{D}+ \\ \mathrm{t_{miss}} * p^{\frac{a}{2}-b} * \mathrm{Coh(D)} * \mathrm{I} \end{array} \right)$$

For LU-Factorization, in the i'th iteration of the outer sequential loop, $2 * (K - i) * B^2$ coherence misses occur, where n is the size of the matrix to be factored, and $n = KB$, where B is the block size. So, the total number of coherence misses in the entire execution scales linearly with respect to the problem size. Thus, the computation and the communication terms are given by,

$$\text{Computation time on the scaled machine} = \left(\ \mathrm{t_C} * p^{\frac{3a}{2}-1} * \mathrm{D} * \mathrm{I}\ \right)$$

$$\text{Communication time on the scaled machine} = \left(\begin{array}{l} \mathrm{f}' * \mathrm{t_{miss}} * p^{\frac{3a}{2}-b} * \mathrm{D} * \mathrm{I}+ \\ \mathrm{t_{miss}} * p^{a-b} * \mathrm{D}+ \\ \mathrm{t_{miss}} * p^{a-b} * \mathrm{Coh(D)} * \mathrm{I} \end{array} \right)$$

The exponent $\frac{3a}{2}$ comes from the fact that the number of outer sequential loop iterations (K) scales as $\sqrt{D}$.

For Fast Multipole, we note from [13] that the amount of communication per processor in a single execution of the outer sequential loop s cales as $\log_{1.828} n * \sqrt{n}$, where n is the number of bodies. The computation and communication terms are given by,

$$\text{Computation time on the scaled machine} = \left(\mathrm{t_C} * p^{\frac{5a}{4} - 1 + \log_p \frac{a^2}{4} + \left(\log_p \log_{1.828} p \right)} * \mathrm{D} * \mathrm{I} \right)$$

$$\text{Communication time on the scaled machine} = \left(\begin{array}{l} \mathrm{f}' * \mathrm{t_{miss}} * p^{\frac{5a}{4} - b} * \mathrm{D} * \mathrm{I} + \\ \mathrm{t_{miss}} * p^{a-b} * \mathrm{D} + \\ \mathrm{t_{miss}} * p^{\frac{3a}{4} - b + \log_p a + \left(\log_p \log_{1.828} p \right)} * \mathrm{Coh(D)} * \mathrm{I} \end{array} \right)$$

The exponent terms of $\frac{5a}{4}$ come from the fact the number of outer sequential loop iterations scales as $\mathrm{D}^{\frac{1}{4}}$, while the terms of $\log_p \frac{a^2}{4}$ and $\left(\log_p \log_{1.828} p \right)^2$ appear fromthe fact that the computation per data element scales as $\left(\log_{1.828} \sqrt{\Delta \mathrm{D}} \right)^2$. The contributions by the log terms in the exponents shown tend to zero as p increases by large values.

For Ocean, the boundary elements in the subgrid assigned to each processor are exchanged in every iteration of the outer sequential loop. Thus, the number of coherence misses scale as $\sqrt{N}$, where N is the problem size. Thus, the computation and communication terms are given by,

$$\text{Computation time on the scaled machine} = \left(\mathrm{t_C} * p^{2a-1} * \mathrm{D} * \mathrm{I} \right)$$

$$\text{Communication time on the scaled machine} = \left(\begin{array}{l} \mathrm{f}' * \mathrm{t_{miss}} * p^{2a-b} * \mathrm{D} * \mathrm{I} + \\ \mathrm{t_{miss}} * p^{a-b} * \mathrm{D} + \\ \mathrm{t_{miss}} * p^{\frac{3a}{2} - b} * \mathrm{Coh(D)} * \mathrm{I} \end{array} \right)$$

The exponent $2a$ appears from the fact that the number of outer sequential loop iterations scales linearly as the data set size.

For Water, from [16] we note that the amount of communication in a single execution of the outer sequential loop scales as $\mathrm{O}(n)$, where n is the number of water molecules. Thus, the computation and the communication terms are given by,

$$\text{Computation time on the scaled machine} = \left(\mathrm{t_C} * p^{2a-1} * \mathrm{D} * \mathrm{I} \right)$$

$$\text{Communication time on the scaled machine} = \left(\begin{array}{l} \mathrm{f}' * \mathrm{t_{miss}} * p^{a-b} * \mathrm{D} * \mathrm{I} + \\ \mathrm{t_{miss}} * p^{a-b} * \mathrm{D} + \\ \mathrm{t_{miss}} * p^{a-b} * \mathrm{Coh(D)} * \mathrm{I} \end{array} \right)$$

The exponent $2a$ appears from the fact that the computation per data element scales linearly as the data set size.

References

[1] Agarwal, A., "Performance Tradeoffs in Multithreaded Multiprocessors", IEEE Transactions on Parallel and Distributed Systems, Vol.3, No.5 (September 1992), pp. 525-539.

[2] Gharacharloo, K., Gupta, A., and Hennessy, J., "Performance Evaluation of Memory Consistency Models for Shared Memory Multiprocessors", Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, CA, April 1991, pp. 245-257.

[3] Gupta, A., and Kumar, V., "On Scalability of FFT on Parallel Computers", IEEE Transactions on Parallel and Distributed Systems, Vol.4. No.8 (August 1993), pp. 922-932.

[4] Gupta, A., and Kumar, V., "The Scalability of Matrix Multiplication Algorithms on Parallel Computers", Proceedings of the International Conference on Parallel Processing, Austin, TX, August 1993, pp.115-119.

[5] Gupta, A., and Kumar, V., "Performance Properties of Large Scale Parallel Systems", Journal of Parallel and Distributed Computing, Vol.19, No.3 (November 1993), pp. 234-244.

[6] Gupta, A., Kumar, V., and Sameh, A.H., "Performance and Scalability of Pre-Conditioned Conjugate Gradient Methods on Parallel Computers", To appear in IEEE Transactions on Parallel and Distributed Systems, 1995.

[7] Kumar, V., and Gupta, A., "Analyzing Scalability of Parallel Algorithms and Architectures", Journal of Parallel and Distributed Computing, Vol.22, No.2. (September 1994), pp. 379-391.

[8] Kung, H.T., "Memory Requirements for Balanced Computer Architectures", Proceedings of the 1986 IEEE Symposium on Computer Architecture, Tokyo, Japan, June 1986, pp. 49-54.

[9] Markatos, E.P., and LeBlanc, T.J., "Using Processor Affinity in Loop Scheduling on Shared Memory Multiprocessors", IEEE Transactions on Parallel and Distributed Systems, Vol.5, No.4 (April 1994), pp. 379-400.

[10] Mowry, T., and Gupta, A., "Tolerating Latency through Software Controlled Prefetching in Shared Memory Multiprocessors", Journal of Parallel and Distributed Computing, Vol.12, No.2 (June 1991), pp. 87-106.

[11] Nussbaum, D., and Agarwal, A., "Scalability of Parallel Machines", Communications of the ACM, Vol.34, No.3 (March 1991), pp. 57-61.

[12] Rothberg, E., Singh, J.P., and Gupta, A., "Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors", Proceedings of the 20th International Symposium on Computer Architecture, San Diego, CA, May 1993, pp. 14-25.

[13] Singh, J.P., "Parallel Hierarchical N-Body Methods and their Implications for Multiprocessors", Ph.D. Thesis, Computer Systems Laboratory, Stanford University, February 1993.

[14] Singh, J.P., Hennessy, J.L., and Gupta, A., "Scaling Parallel Programs for Multiprocessors: Methodology and Examples", IEEE Computer, Vol.26, No.7 (July 1993), pp. 42-50.

[15] Singh, J.P., Joe, T., Gupta, A., and Hennessy, J.L., "An Empirical Comparison of the Kendall Square Research KSR-1 and Stanford DASH Multiprocessors", Proceedings of Supercomputing'93, November 1993, Portland, OR, pp. 214-225.

[16] Singh, J.P., Weber, W.D., and Gupta, A., "SPLASH: Stanford Parallel Applications for Shared-Memory", Computer Architecture News, Vol.20, No.1 (March 1992), pp. 5-44.

[17] Singh, V., Agha, G., and Tomlinson, C., "Scalability of Parallel Sorting on Mesh Multicomputers", International Journal of Parallel Programming, Vol.20, No.2. (March 1991), pp. 115-128.

[18] Subramanian, A., Singla, A., Ramachandran, U., and Venkateswaran, H., "An Approach to Scalability Study of Shared Memory Parallel Systems", Proceedings of the 1994 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, Nashville, TN, May 1994, pp. 171-180.

[19] Sun, X., and Ni, L.M., "Scalable Problems and Memory-Bounded Speedup", Journal of Parallel and Distributed Computing, Vol.19, No.1 (September 1993), pp. 27-37.

[20] Sun, X., and Rover, D.T., "Scalability of Parallel Algorithm-Machine Combinations", IEEE Transactions on Parallel and Distributed Systems, Vol.5, No.6 (June 1994), pp. 599-613.

[21] Zhang, X., Yan, Y., and He, K., "Latency Metric: An Experimental Method for Measuring and Evaluating Parallel Program and Architecture Scalability", Journal of Parallel and Distributed Computing, Vol.22, No.3 (September 1994), pp. 392-410.

[22] Lewis, G.T., "Where is Computing Headed? ", IEEE Computer, Vol.27, No.8 (August 1994), pp. 59-63.

[23] Boxer, A., "Where Buses Cannot Go ", IEEE Spectrum, (February 1995), pp. 41-45.