The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors

Chris Holt, Mark Heinrich, Jaswinder Pal Singh, Edward Rothberg, and John Hennessy

Abstract

Distributed shared memory (DSM) machines can be characterized by four parameters, based on a slightly modified version of the *logP* model. The *l* (latency) and *o* (occupancy of the communication controller) parameters are the keys to performance in these machines, and are largely determined by major architectural decisions about the aggressiveness and customization of the node and network. For recent and upcoming machines, the *g* (gap) parameter that measures node-to-network bandwidth does not appear to be a bottleneck. Conventional wisdom is that latency is the dominant factor in determining the performance of a DSM machine. We show, however, that controller occupancy—which causes contention even in highly optimized applications—plays a major role, especially at low latencies. When latency hiding is used, occupancy becomes more critical, even in machines with high latency networks. Scaling the problem size is often used as a technique to overcome limitations in communication latency and bandwidth. We show that in many structured computations occupancy-induced contention is not alleviated by increasing problem size, and that there are important classes of applications for which the performance lost by using higher latency networks or higher occupancy controllers cannot be regained easily, if at all, by scaling the problem size.

1 Introduction

For systems with more than a small number of processors, distributed shared memory (DSM) multiprocessors are converging to a family of architectures that resemble the generic system shown in Figure 1.1. This architecture consists of a number of processing nodes connected by a general interconnection network. Every node contains a processor, its cache subsystem, and a portion of the total main memory on the machine. It also contains a *communication controller*, which is responsible for managing the communication between it and other nodes. The shared address space provides ease of programming, and the distributed memory and interconnect provide the increased bandwidth needed for performance and scalability. Our interest in this paper is in a specific class of DSM machines: those that support communication and coherent replication at the fixed granularity of cache lines.

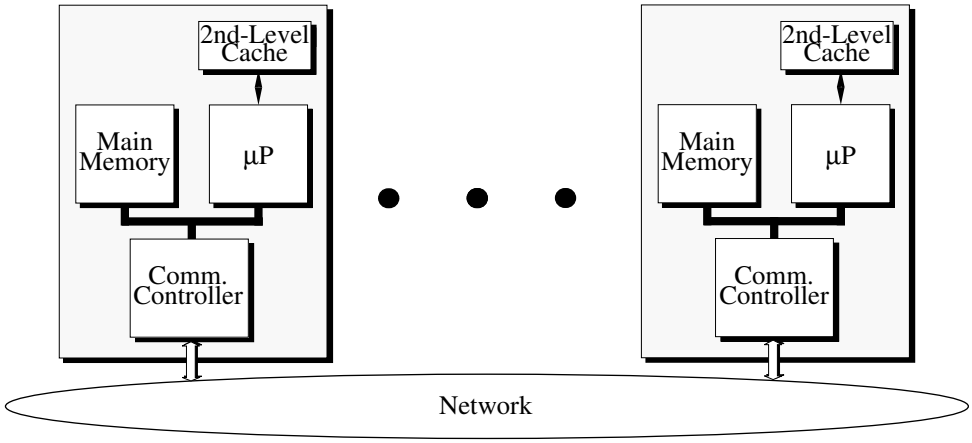

Figure 1.1. Convergent DSM architecture

There are various ways to build cache-coherent DSM machines, arising from differences in desired performance and cost characteristics and in the extent to which one wants to use commodity parts and interfaces rather than build customized hardware. We assume the use of a commodity microprocessor, cache subsystem and main memory in this paper. The major sources of variability are in the network and the communication controller, which together constitute the communication architecture of the multiprocessor.

Candidate networks vary in their latency and bandwidth characteristics as well as in their topologies. They range from low-latency, high-bandwidth MPP networks, all the way to commodity asynchronous transfer mode (ATM) networks. On the controller side, there are two important and related variables. One is the location of the communication controller in the process-

ing node. The communication controller can be located in the cache controller, in the memory subsystem, or on the I/O bus. The closer to the processor we locate the controller, the greater the performance, but the less we can leverage off commodity parts for the processing node. The other design variable is how specialized the controller is for the tasks it performs; for instance, it may be a hardware finite state machine, a customized special-purpose processor that runs protocol code in response to events, or an inexpensive off-the-shelf general-purpose processor.

Because of their differences in design cost, all of these types of systems are interesting. Current and proposed architectures for fine-grained distributed shared memory take different positions on the above tradeoffs. The unanswered question is how the performance characteristics of the network and controller affect how well the machines will run actual parallel programs. That is, as we move from more tightly-coupled and specialized systems to less tightly-coupled and more commodity-based systems, how much effectiveness in parallel performance do we lose over a wide range of computations? This is the question we address in this paper, by studying a range of important computations and interesting communication architectures through a combination of analytical modeling and detailed simulation.

We characterize the communication architectures of DSM multiprocessors by a few key parameters that are similar to those in the *logP* model [CKP+93]. An abstract model is of course simplistic from the perspective of an actual architecture, since it does not capture many of the details of real machines. We shall set up our architectural context and discuss some of the more detailed issues in the next section. Section 3 describes the framework and methodology we use to study the effectiveness of different types of DSM architectures. Section 4 and Section 5 present and analyze our results, and Section 6 concludes the paper.

2 Architectural Context

As in the *logP* model, we abstract the multiprocessor communication architecture of a parallel machine in terms of four parameters. The l parameter in our model stands for the network latency from the moment the message enters the network from a source node to the moment it arrives at its destination node, o is the overhead of sending a message, g is the gap (reciprocal of node to network bandwidth through the network interface), and P is the number of processors. The only difference between our model and the *logP* model is in the o parameter. In the *logP* model, o represents the overhead of a message or communication, which is the time during which the processor is busy initiating or receiving a message and cannot do anything else. In most DSM machines, however, protocol processing is off-loaded to a separate communication controller, and the processor is free to continue doing independent work while the controller is occupied[1]. The o parameter in our DSM model, then, stands for the *occupancy* of the communication controller per protocol action or message; that is, the time for which the controller is tied up with one action and cannot perform another.

We fix the number of processors P at 64 in this paper. The three parameters that characterize the communication architecture—latency, occupancy, and bandwidth or gap—all have complicated aspects to them, and we make certain simplifying assumptions. Let us first discuss each parameter individually, before placing our range of variations of these parameters in the context of realistic machines.

Latency: The latency of a message through the network depends, among other things, on how many hops the message travels in the network topology. For the moderate-scale machines that we consider, the overhead of getting the message from the processor into the network and vice versa usually dominates the topology-related component of the latency seen by the processor. We therefore ignore topology, and assume that the network transit time from one node to another is always the same.

Occupancy: The occupancy that the controller incurs for a request affects performance in two ways. First, it contributes directly to the latency of the current request because the request must pass through the controller. Second, it can contribute indirectly to the latencies of subsequent requests, through contention for the occupied controller. Occupancy may be more difficult to represent as an abstract parameter than network latency for two reasons. First, we have to decide which types of transactions invoke actions on the controller and hence incur controller occupancy. Second, while occupancy in real machines often depends on the type of the transaction, we want to represent it by a single parameter o. We now examine these issues separately.

Clearly, all events related to internode communication and protocol processing incur controller occupancy, including local cache misses that need data from another node, references from the processor that require the communication of state changes to other nodes, and incoming requests and replies from the network containing data and protocol information. The question is whether the controller should also handle local cache misses that do not generate any communication. In this paper, we assume a *bypass path* to local memory for these misses, so that they need not invoke the controller [RLW94, SFL+94]. We also

1. This is not specific to DSM machines; even in message-passing architectures the message handling can be off-loaded to a communication co-processor or controller, as in the Intel Paragon.

assume that the state lookup that determines whether or not a miss needs to invoke the controller is free, and hence does not contribute to the latency of the miss.

In a real machine, particularly one in which the communication controller runs software code sequences for protocol processing, the occupancies of the controller are often different for different types of protocol actions. We make the following assumptions about occupancy. When the communication controller is simply generating a request into the network or receiving a reply from the network it incurs occupancy o. When the communication controller is the home of a network request it incurs occupancy $2o$, because it has to retrieve data and/or manipulate coherence state information [HKO+94]. In this case the memory access occurs in parallel with the operation of the controller. If the state lookup at the home reveals that the requested line is dirty in the home node's cache, the communication controller incurs an extra occupancy C. If the requested line is dirty in a third processor's cache, the home node forwards the request to that processor and the communication controller at that node incurs an occupancy of $2o$+C. The only other time occupancy is incurred is when the communication controller at the home node is servicing a write and must send invalidations to all nodes that are sharing the data. In this case the controller incurs an additional occupancy of one cycle per invalidation that it sends.

Bandwidth or **gap**: Another important parameter related to the communication architecture is the node-to-network bandwidth, which determines how fast data can be transferred through the network interface, i.e. between the communication controller and the network itself. We ignore the gap parameter in the main discussion in the paper, and simply compute the node-to-network bandwidth requirements of the individual applications in Section 4.3. We ignore gap because the node-to-network bandwidth we assume (400MB/s peak, which corresponds to MPP networks on next-generation machines) is large enough to never be a performance bottleneck in our experiments. For coherence messages that do not carry data, the occupancy of the communication controller always dominates our gap limitation. For messages that carry data, 400MB/s node-to-network bandwidth can theoretically become the bottleneck before controller occupancy for the two lowest occupancies we examine. However, we never observed the symptomatic filling of network interface buffers in practice, both because a given processor allows only a limited number of outstanding requests, and since transactions that do not involve data are usually interspersed with data requests.

Given these assumptions about l, o and, g, let us examine the path of a read miss to a line that is allocated on a remote node and is clean at its home. The request travels through the communication controller on the requesting node (o), traverses the network (l), travels through the communication controller at the home where the request is satisfied ($2o$), traverses the network again (l), and finally travels back through the communication controller at the source node (o). Including the fixed external interface delays into and out of each controller (k_{in}, k_{out}) leads to a total round-trip latency as seen by the processor (without any contention) of $k_{in} + o + k_{out} + l + k_{in} + 2o + k_{out} + l + k_{in} + o + k_{out}$ for the miss, or $2l + 4o + 6k$ if we assume $k_{in} = k_{out}$. If the line were dirty at the home node's cache, there would be an extra fixed cost of C at the home for retrieving the data from the cache. For a line that is dirty in the cache of a third processor (not the requestor or the home), the latency would be $k_{in} + o + k_{out} + l + k_{in} + 2o + k_{out} + l + k_{in} + 2o + C + k_{out} + l + k_{in} + o + k_{out}$, or $3l + 6o + C + 8k$.

The network latency l and the controller occupancy o are the variables in the above costs. We study a range of values for each variable (instantiated as L1, O1, and multiples of these), as shown in Table 2.1, covering a variety of interesting architectural alternatives. We have used reasonable estimates to characterize the latencies and occupancies of these architectural alternatives, though of course each alternative could be customized or improved. Our latencies l vary from tightly-coupled, low-latency MPP networks, through high-latency physically distributed MPP networks, all the way to commodity networks com-

Table 2.1. Machine Configurations and their Abbreviations

Controller Occupancy (system cycles)	**Network Latency (system cycles)**			
	25 MPP	50-200 Distributed MPP	400 Aggressive ATM	800 Today's ATM
< 7 Hardwired Controller	L1, O1	L2, L4, L8 O1	L16, O1	L32, O1
14-28 Customized Co-processor	L1, O2-O4	L2, L4, L8 O2-O4	L16, O2-O4	L32, O2-O4
56 General-purpose Co-processor on Memory Bus	L1, O8	L2, L4, L8 O8	L16, O8	L32, O8
112 General-purpose Co-processor on I/O Bus	L1, O16	L2, L4, L8 O16	L16, O16	L32, O16

posed of ATM switches[2]. Small values of occupancy *o* represent communication controllers which are tightly-integrated, hardwired state machines [ACD+91, KSR92, LLG+92]. As *o* increases the controller becomes less hardwired and more general-purpose, from specialized co-processors [KOH+94], through inexpensive off-the-shelf processors on the memory bus, to an off-the-shelf processor located on the I/O bus of the main processor. The cycle counts used in Table 2.1 and the remainder of this paper are in terms of 100 MHz system cycles, i.e. the clock speed of the controller and the rest of the node, not that of the main processor. The entries in Table 2.1 correspond to specific values in our range of latencies and occupancies. The L1, O1 point is our base architecture, with a network latency of 25 system cycles or 250ns, and a controller occupancy of 7 system cycles[3]. The other entries are multiples of these base values.

With this context established, we now present our framework for studying the effects of varying *l* and *o* on system performance over a range of important parallel computations.

3 Framework and Methodology

3.1 How We Approach the Problem

Our goal is to understand the impact of latency and occupancy on the effectiveness of a DSM system, and thus understand how much we lose as we relax the aggressiveness of our architectures along these dimensions. The first question we must resolve is how to evaluate the effectiveness of an architecture for an application when parameters like *l* and *o* are varied. One possibility is simply to examine how the performance of the architecture on a fixed problem size degrades as *l* and/or *o* are increased. This may be the most important question to one who really cares about that particular application and data set. It also provides useful insights, which is why we study it in this paper. However, it is not our main indicator of architectural effectiveness since the results depend greatly on the problem size that is used[4].

The impact of communication performance on overall system performance depends not only on the structure of the communication but also on the ratio of computation to communication. Also, the impact of communication depends on whether communication really is the important performance bottleneck to begin with, or whether the bottleneck is something else like load imbalance. For a given number of processors, both these issues—the computation-to-communication ratio, and what the dominant bottleneck is—usually depend on the problem size that is used. Larger data sets usually improve both the computation-to-communication ratio and the load balance, and hence allow a machine to deliver better parallel performance or speedup relative to a uniprocessor implementation.

Thus, if an application with a given problem size delivers good parallel performance on an aggressive architecture but not on one that uses a commodity controller and network, this does not in itself mean that the less aggressive communication architecture is inappropriate for that application. There may be a problem size for which the less aggressive architecture performs well too, and perhaps another problem size for which it performs almost as well as the aggressive architecture. The question is how large are these problem sizes relative to the base problem size, and are they still realistic or interesting. Thus, we believe that the best way to cast the effectiveness question is: *Given an application, a number of processors, and values for l and o that characterize the network and controller, what is the minimum sized problem that can deliver a desired level of parallel performance?*

The question that remains is the choice of the "desirable" level of performance. Our measure of parallel performance is the parallel efficiency of an execution; that is, the speedup of the parallel execution over a sequential implementation of the application on a uniprocessor, divided by the number of processors used (64). Typically, the larger the efficiency the larger the problem size needed for a given combination of *l* and *o*. Thus, the efficiency level we choose is an important determinant of the constant factors in the expression for the required problem size. Furthermore, it can also affect the growth *rate* of the required problem size with *l* and *o*, if changing the desirable efficiency level changes the relative importance of different performance bottlenecks. For example, for an efficiency level of 30%, the dominant bottleneck to overcome by increasing problem size may be communication, but for a 95% efficiency level it may be load imbalance. The bottlenecks also may not behave in predictable ways as problem size or efficiency level change, particularly for irregular applications. In most cases, however, if the dominant bottleneck does not change, then the chosen level of efficiency will not affect the growth rate of the required problem size but only the starting point. We assume a desired parallel efficiency of 60% in this paper.

---

2. Although our highest value of latency corresponds to the latency of an ATM switch, it does not represent actual ATM networks because the 400 MB/s bandwidth we assume is much higher than current ATM bandwidth.

3. We also experimented with a controller occupancy of 3 cycles, and the results were essentially the same as those for a controller occupancy of 7 cycles.

4. The problem size is determined by many application parameters, including the size of the data set, the number of time-steps, and the accuracy of the computation. These parameters themselves have different impacts on computation-to-communication ratio. We focus on data set size when we refer to problem size, and we specifically address other parameters at the end of the paper.

Given the large communication latencies on DSM machines, it is natural to try to hide these latencies when possible. Latency can be hidden by various techniques, all of which exploit the availability of additional bandwidth and require that the processor allows multiple outstanding references. To hide write latency, we assume that the architecture supports a relaxed consistency model. Read latencies are typically more difficult to hide, and it is not clear how successfully communication latencies for read misses will be hidden in practice. Where applicable, we use two versions of our applications: one that tries to hide read latency with software-controlled prefetches that we insert in the application by hand, and the other that does not.

3.2 Simulation Environment

The results presented in this paper are gathered from a detailed memory simulator that interfaces to the Tango Lite event-driven reference generator [Golds93]. The simulator models contention in detail within the communication controller, between the controller and its external interfaces, at main memory, and for the system bus. The input and output queue sizes in the controller's processor and network interfaces are uniformly set at 16 entries. We assume processor interface delays of 1 cycle inbound and 4 cycles outbound, and network interface delays of 8 cycles inbound and 4 cycles outbound. The total round trip interface delay (k) encountered on a remote clean miss is 29 system cycles. We assume that the latencies through the interfaces remain fixed as controller and network characteristics are varied. We also fix the access time of main memory DRAM at 140 ns (14 system cycles), a fairly aggressive number. Fixing the interface delays and the memory access time is realistic, and allows us to focus on the performance of the communication architecture.

We assume a fast, next-generation main processor that can issue up to three memory references every 100 MHz system cycle. Because the processor controls its own secondary cache, we assume that it takes 15 system cycles for the controller to retrieve state information from that cache when necessary. This is the value of C used in our simulations (see Section 2). The caches are 1 MB in size, two-way set associative, and have a line size of 128 bytes. We also assume that the processor has both prefetch and prefetch exclusive instructions. In our processor model a load miss stalls the processor until the first double-word of data is returned, while prefetch, prefetch exclusive, and store misses will not stall the processor unless there are already references outstanding to four different cache lines. While this upper bound of only four outstanding cache lines can limit the amount of latency that a processor can hide with bandwidth, it is nonetheless more aggressive than the current situation in commodity microprocessors.

3.3 Applications

Table 3.1. Applications and Communication Patterns

Application	Description	Communication Pattern
Barnes	Barnes-Hut hierarchical N-body simulation	irregular, hierarchical
Ocean	Multigrid large scale ocean simulation	nearest neighbor iterative, hierarchical
Water	Molecular dynamics simulation	structured, many-to-many
FFT	Radix $\sqrt{n}$ Six-Step Fast Fourier Transform	regular, all-to-all, blocked matrix transpose
LU	Blocked dense LU decomposition	structured, one-to-many
Radix	Integer radix sort	irregular, all-to-all

The applications we use in our study are summarized in Table 3.1.The programs were chosen because they represent a variety of important scientific computations, including both kernels and complete applications, and they have different communication patterns and requirements. Barnes is representative of the class of hierarchical N-body methods, which are used in the domains of astrophysics, electrostatics, and plasma physics, among others. Ocean is representative of many computational fluid dynamics applications on regular grids. Water is representative of a wide range of computational chemistry applications which compute particle interactions based on a cutoff radius. FFT forms the computational core of a wide variety of applications, including image and signal processing as well as climate modelling. The most common need for large dense LU factorization is in radar cross-section problems; however, for our purposes dense LU factorization is very similar to more widely used sparse matrix factorization techniques (such as blocked Cholesky factorization [Roth93]), and of various other matrix factorization and eigenvalue methods. Finally, Radix is a widely used sorting algorithm. Descriptions of the applications can be found in: Barnes [HS94]; Radix and Ocean [WSH94]; Water [SWG+94]; FFT and LU [RSG93]. The applications are quite highly optimized to improve communication performance, and particularly to reduce spurious hot-spotting or contention effects that adversely impact occupancy. The codes for the applications are taken from the SPLASH-2 application suite [SWG+94], although Radix was modified to use a tree data structure (rather than a linear key chain) to communicate ranks and densities efficiently.

4 Results for a Fixed Problem Size

We have said that the best way to look at how latency and occupancy impact the effectiveness of DSM architectures on a given application is to determine how the problem size necessary to achieve a given level of efficiency changes with latency and occupancy. However, it is also useful to examine how the parallel efficiency of an application changes with l and o for a fixed problem size; namely one that yields the desired level of efficiency on our base architecture. Understanding how latency and occupancy affect the base problem lends important insight into how these parameters of the communication architecture interact, and into how much each contributes to performance degradation. Combined with our understanding of the applications, this also guides our search for larger problem sizes to retain the desired efficiency.

4.1 Intuition: What Should We Expect?

As l and o increase for a given problem size, parallel efficiency clearly should decrease. But can we predict *how* it decreases for the different applications? The time taken by a parallel application can be broken down into two components: local computation, including cache and local memory accesses, and communication. Communication cost can be further broken down into the cost due to latency and the cost added by contention for a resource of limited bandwidth (non-zero occupancy). Assuming perfect load balancing of both computation and communication, and that local caching effects stay the same for uniprocessor and multiprocessor problems, the parallel efficiency is determined by the following formula:

$$\frac{T_{comp}}{T_{comp} + V_{comm} \times (T_L + T_C)} \tag{4.1}$$

where T_{comp} is the uniprocessor computation time, V_{comm} is the volume of communication (number of communication misses incurred on all processors), and T_L and T_C are the average stall times due to latency and contention, respectively, for each communication.

For a fixed problem size and number of processors, both T_{comp} and V_{comm} are constant. T_L varies linearly with l and o. The remaining question is how the contention component, T_C, varies with l and o. If controller occupancy contributes only to latency and has no contention component ($T_C = 0$ for all o in the above formula), then an increase in occupancy is indistinguishable from holding occupancy constant and making a corresponding increase in network latency. We define *communication latency* as the round-trip latency, assuming no contention, for a remote miss that is satisfied by the main memory of the home node (computed as $2l+4o+6k$ in Section 2). On a graph of parallel efficiency versus communication latency, if T_C were 0 then all points in the l, o design space would fall on the same curve. The exact shape of the curve can be gleaned from Eq. 4.1, once the constant factors T_{comp} and V_{comm} are known. Now suppose occupancy causes contention for the controller as well. As o increases, the contention worsens and the parallel efficiency becomes worse than it would have had the occupancies stayed the same but network latency been increased correspondingly. The efficiency versus communication latency curve for a larger occupancy would therefore be below that for a lower occupancy. The impact of the contention component of occupancy is likely to be larger when network latencies are smaller, which means that the curve will start to flatten at smaller latencies. However, as latency increases, the effect of contention should diminish, and eventually the curve should approach the curves for lower occupancies.

To understand the performance impact of l and o, we seek the answers to the following questions: (i) starting from the base architecture, how does increasing network latency degrade performance for the base problem size, both with and without prefetching; and (ii) to what extent does controller occupancy cause contention in addition to contributing latency, both with and without prefetching, and how does this contention affect parallel efficiencies for realistic architectures with different occupancies and latencies. Other interesting questions that we answer in the process are: What is the problem size needed to obtain 60% parallel efficiency on the base architecture, which represents an aggressive next-generation multiprocessor, both with and without prefetching; and what are the node-to-network bandwidth requirements for the base problem size?

4.2 A Case Study

In this section we outline the framework for our fixed problem size results in the context of a single application. As a case study, we choose to present our results for FFT because it is a simple, well-understood program that illustrates the framework we use to analyze all our applications. For both the prefetched and non-prefetched versions of the program, we examine the results of varying latency and occupancy for a fixed problem size, obtained through both analytical modelling and detailed simulation. Deriving analytical models and validating them with data obtained from detailed simulations can lead us to a more fundamental understanding of the effects and interactions of latency and occupancy than simply observing the results obtained from simulation. Many of the important insights emerge in the course of this case study. In Section 4.3 we present results for all our applications, and compare and contrast them with those obtained here.

4.2.1 Results from Analytical Modelling

We first try to model the non-prefetched version of FFT. The communication phase in FFT consists of multiple blocked matrix transposes, where in each transpose a processor reads parts of the columns of a source matrix from other processors, and writes the rows of a locally-owned destination matrix. If we assume no contention, modelling the transpose phase is straightforward. We simply compute the number of cache lines a processor needs to read, multiply it by the communication time, and add the computation time. This model is sufficient for low occupancies, because then occupancy does not cause much contention. The model becomes less accurate for high occupancies, particularly at low latencies, up to 40% off at a network latency of L1 and a controller occupancy of O16 (recall the architectural alternatives in Table 2.1). Figure 4.1(a) shows how this model, called *no contention*, compares to the actual execution time obtained by simulation for two different occupancies, O4 and O16. In this graph, as in all graphs in the paper, we plot parallel efficiency versus *communication latency*, which

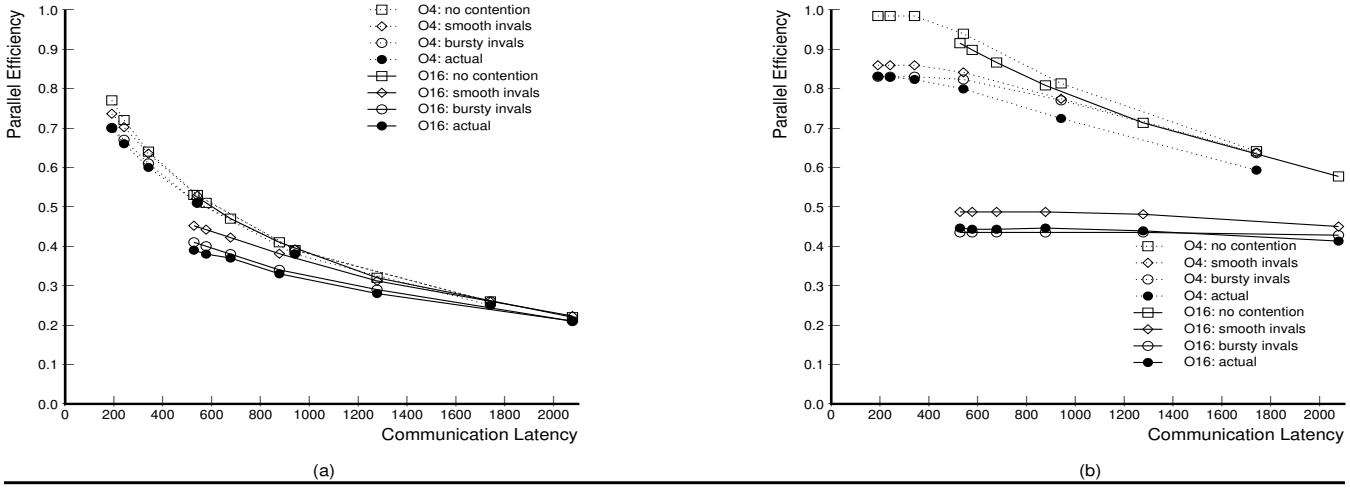

Figure 4.1. FFT modelling results for both the (a) non-prefetched and (b) prefetched versions

was defined in Section 4.1. Communication latency is different from network latency or the *l* parameter, which is the one-way latency of a message in the network itself. The leftmost point on every curve corresponds to the communication latency assuming the base value of *l* and the current value of occupancy *o*.

The actual results for high occupancies and low latencies suggest that occupancy indeed contributes to contention even without prefetching. To model contention, we use a queueing model to determine the average number of requests that are waiting for the communication controller when a request arrives. The simplest queueing model assumes that there is a maximum of one remote read request that the controller has to handle, together with its own processor's read request, because reads are blocking and the processors are kept in synch. We found, however, that a model that only takes read misses into account (not shown in Figure 4.1) does not do much better than the *no contention* model. The model performs poorly because data copying in all but the first transpose phase causes invalidations to processors that previously read those data in the prior transpose phase. The invalidations consume occupancy, and we must include them in the model. If the invalidations are assumed to occur uniformly during the transpose phase, we get the *smooth invals* model, which is still 15% off with the O16 controller at low latencies. It is only when the queueing model is modified to take the burstiness of coherence traffic (resulting from the interactions of multi-word cache lines with the patterns of reading and writing data in the transpose phases) into account, that it matches the simulations well, as shown by the *bursty invals* curve in Figure 4.1(a).

Although it is possible to create a model that accurately predicts the behavior of the FFT, this model proved surprisingly difficult to generate in light of the relatively simple structure of the computation. The prefetched version of the transpose phase is even more complex to model, because processors prefetch data from one processor while they are still communicating with another. Clearly if modelling contention and invalidations is necessary when there are no prefetches, it is even more important when there are, as Figure 4.1(b) shows. The best model we found for the prefetched version sets an upper bound on the number of messages a controller may receive by assuming it handles requests by at most two other processors at any time. Unfortunately, it is not as accurate as the best non-prefetched model for all occupancies.

In both the prefetched and non-prefetched versions it is the occupancy-related contention effects that make accurate modelling difficult (the divergence of the simple models in Figure 4.1 gets worse with increasing occupancy and better with increasing latency). The key property that enables us to model the non-prefetched version of FFT well is that we are able to set a tight limit on the number of messages the controller has to handle in a fixed amount of time. It is difficult to set a similar limit in the other less easily analyzed applications.

4.2.2 Simulation Results for the Base Problem Size

Both to validate our model and to obtain the constant factors in the computation-to-communication ratio, we now look at our simulation results for FFT in more detail.

Without Prefetching: Figure 4.2(a) graphs parallel efficiency vs. communication latency for our base FFT problem size (256K points)[5]. The curves generally have the form predicted in Section 4.2.1. The first interesting result is that larger occupancies lower the curves, indicating that the contention component of occupancy is indeed important, even without prefetching. The curves also begin to flatten as o is increased, which indicates that the controller starts to saturate.

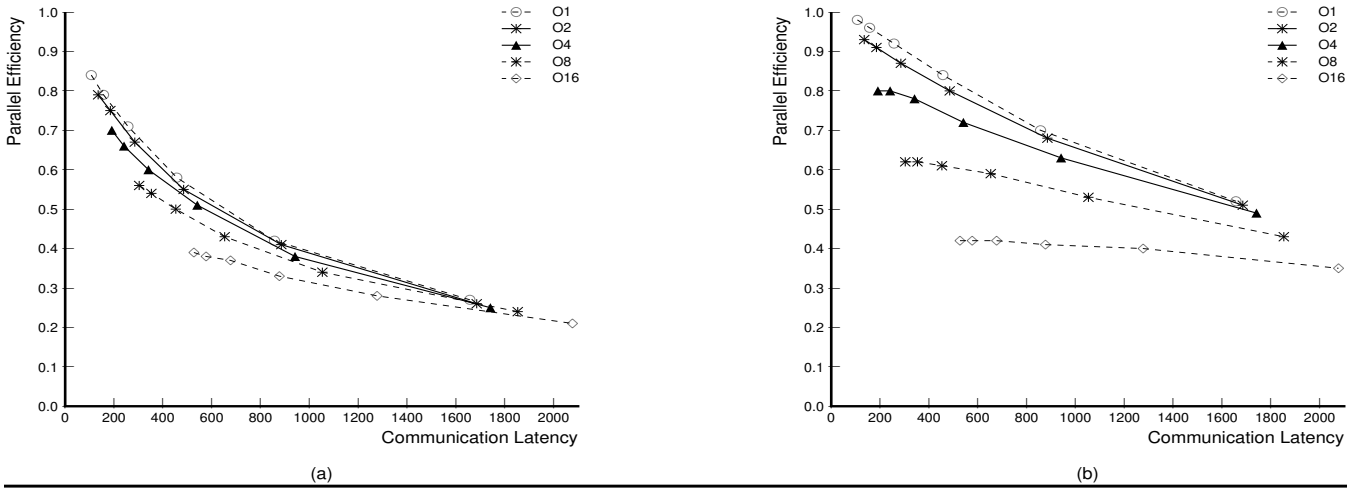

Figure 4.2. FFT base problem size results for both the (a) non-prefetched and (b) prefetched versions

Note that all curves nearly converge at high values of l, implying that at today's ATM latencies controller occupancy does not have a large impact on overall performance for this problem size without prefetching. Conversely, for a range of MPP and distributed MPP network latencies, controller occupancy is a critical determinant of overall performance. What may be surprising are the values of controller occupancy at which the curves begin to diverge at low l. The difference between the O1 and O2 curves for the smallest value of l is small (6%), but the difference from O2 to O4 is greater (13%), and gets progressively larger from O4 to O8 (general-purpose processor on memory bus; 25%) and from O8 to O16 (general-purpose processor on I/O bus; 44%).

With Prefetching: Figure 4.2(b) shows that with prefetching, the curves are no longer concave. In fact, they are almost linear with communication latency and flatten out as o increases. Unlike the non-prefetched case, the curves no longer converge because the contention component of occupancy affects overall performance even at high network latencies. Prefetching improves performance more at low o and moderate l than it does at higher values of o and l. At high l, we cannot hide all the network latency, and beyond a point increases in latency hurt the prefetched case at as quick a rate as the non-prefetched so the curves take on similar shapes. At high o, the controller becomes a bottleneck, as it is unable to match the increased bandwidth needs of prefetching. To support prefetching in DSM machines then, it is crucial to keep the occupancy of the controller low.

Let us look at the graphs from the perspective of a systems architect. If we had a network with latencies like today's ATM networks, how much does the occupancy of our communication controller affect overall performance? For this problem size, the answer is not much if we do not use latency hiding techniques, but significantly if we do. With prefetching, O1 is 2.32 times better than O16 at latency L1, and 1.49 times better at latency L32. On the other hand, if we could reduce our network latency in half (reach the current goal of ATM networks) how much performance would we gain? The answer here depends on the occupancy of the controller. A machine with a controller occupancy of O1 makes a 56% improvement in parallel efficiency as network latency is decreased from L32 to L16, while a machine with controller occupancy of O16 makes a 33% performance improvement. While the relative gains in performance are quite high, the absolute performance of both of these systems is still low compared to the base architecture.

Now suppose we have a low-latency (L1) MPP network. Beyond a very efficient customized controller on the memory bus (O2), controller occupancy is crucial to performance both with and without prefetching. For low occupancy controllers, going

5. For FFT, the required problem size for 60% efficiency on 64 processors is small enough in our most aggressive architecture that artifacts of its interactions with low-level architectural parameters (such as cache line size) throw it off the growth rate curve. We therefore use an efficiency of 80% for FFT, though the growth rates would have been the same if we had used 60% with different low-level architectural parameters that did not cause these artifacts.

to a higher-latency network also hurts performance significantly, though with prefetching the performance impact is smaller. Once the controller is a general-purpose processor (on the I/O or even memory bus), increasing network latency does not significantly affect performance. Note that starting from a very efficient (L1, O1 or O2) machine, doubling controller occupancy hurts more than doubling network latency. As designers of tightly-coupled machines, if the cost considerations for doubling the two parameters are similar, we might favor keeping occupancy low and sacrificing some network latency.

So we see that for FFT, without latency hiding controller occupancy is critical at low network latencies but not at high latencies, while with prefetching it is critical at all latencies. Since high-latency networks make it all the more important to hide latencies if possible, occupancy is in effect critical at all values of network latency (of course, occupancy would become less critical if the network bandwidth were very low as well). An interesting result is that it is the contention component of controller occupancy, not its latency component, that dominates its contribution to performance degradation, both with and without prefetching. We find that for most of our applications, for controller occupancies above O2—which represents an efficient, customized co-processor—70%-95% of the performance degradation due to increasing occupancy is attributed solely to its contention component for all values of network latency. We will now show that these trends are consistent across many classes of applications, and will discuss how our other applications corroborate the detailed results or differ from them.

4.3 Results

For each of the applications, Table 4.1 first shows the minimum problem size needed to achieve 60% efficiency on the base architecture. The table also shows the per-processor communication bandwidth requirements for that problem size (including all protocol messages). The bandwidth numbers are presented in megabytes divided by total execution time, not just that of the communication phases. Note that the bandwidth numbers in Table 4.1 are moderate, and in all cases are much less than our

Table 4.1. Minimum Problem Sizes and Per-Processor Bandwidth Requirements for the Base Architecture

	Non-prefetched		Prefetched	
Application	**Minimum Problem Size**	**Bandwidth (MB/s)**	**Minimum Problem Size**	**Bandwidth (MB/s)**
Barnes	8192 particles	7.4	N/A	N/A
Ocean	258x258 grid	40.3	258x258 grid	46.4
Water	512 molecules	10.9	N/A	N/A
FFT	64K points	50.4	64K points	50.5
LU	768x768 matrix	19.3	768x768 matrix	21.3
Radix	2M keys, radix 256	83.7	1M keys, radix 256	82.3

node-to-network bandwidth of 400 MB/s (even with burstiness, we find network bandwidth not to be a bottleneck, as mentioned earlier). Bandwidth requirements will generally decrease as problem sizes are increased.

Following the framework developed in Section 4.2, we now present base problem size results for all of our applications. We compare these results with those we have already seen for our FFT case study.

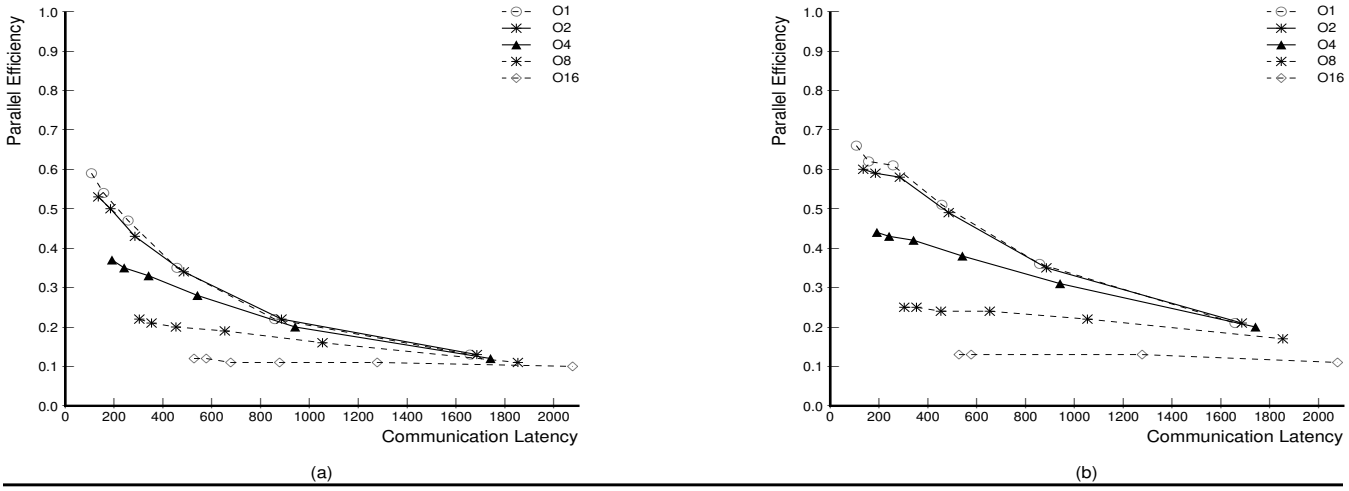

Figure 4.3. Radix results for the base problem size for both the (a) non-prefetched and (b) prefetched versions

Radix: The results for Radix shown in Figure 4.3 are similar to FFT, with a few notable exceptions. Like FFT, without prefetching all the curves almost converge by today's ATM latencies (our rightmost points). While the O1 and O2 curves are

still very close together, the O8 curve is much flatter than it is in FFT, and the O16 curve is almost totally flat. This indicates that in Radix contention matters more than it does in FFT, particularly at low network latencies.

In the prefetched version of Radix, we again see a linearization of all the curves although less than in prefetched FFT (i.e. prefetching is not as successful in Radix as it is in FFT). Two key prefetching trends continue in Radix: prefetching helps much more at lower values of o, and the curves do not converge to a point at L32, indicating that it is still critical to keep occupancy low when prefetching, even with ATM latencies.

LU: The results for LU (not shown) are also very similar to those for FFT. One significant difference for both prefetched and non-prefetched LU is that the performance is less sensitive to both latency and occupancy. The reason is that LU has a high computation-to-communication ratio, and suffers from significant load imbalance for the base problem size, so its performance is less dependent on communication costs.

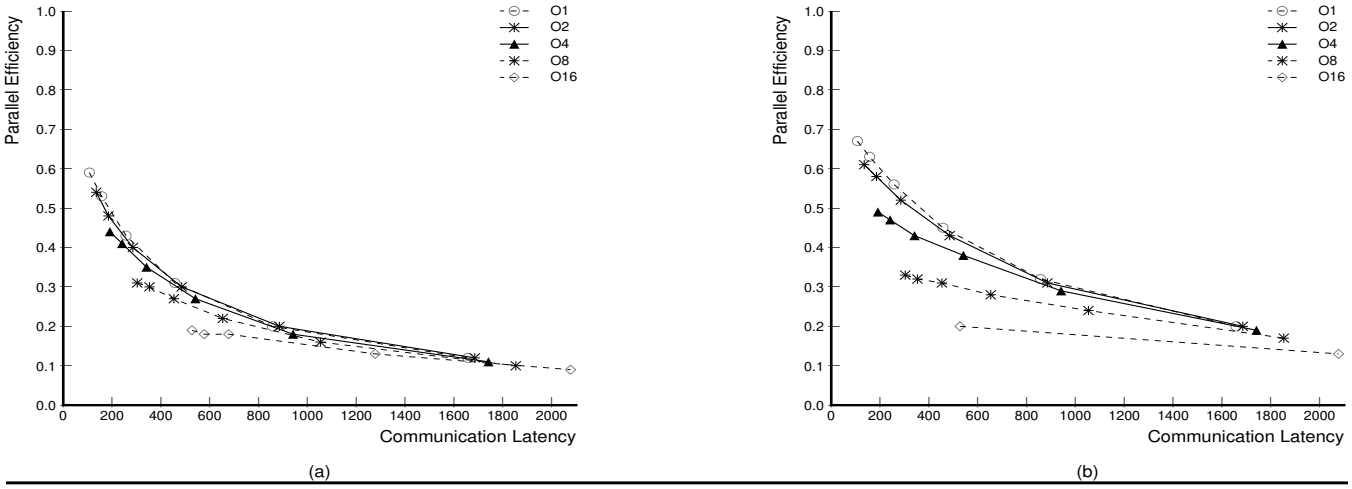

Figure 4.4. Ocean results for the base problem size for both the (a) non-prefetched and (b) prefetched versions

Ocean: Ocean, which performs many iterative nearest-neighbor computations on regular grids, depends more on network latency than any of the previous applications, though it depends substantially on occupancy as well (Figure 4.4). This is especially true at higher controller occupancies. The reason is that unlike the previous applications, Ocean cannot take full advantage of spatial locality when it communicates data, leaving it especially sensitive to changes in network latency. Prefetched Ocean cannot hide enough of the latency, so the prefetched curves are also somewhat concave.

Barnes and Water: Figure 4.5 shows the results for Barnes and Water. Neither application includes prefetching, because the high degree of temporal locality (and irregularity in Barnes) makes it difficult to determine what to prefetch and when. For Barnes, the O1 and O2 curves are identical. The O4 curve is different only for the lowest values of l, and the curves do not begin to diverge until O8. Again, the curves all converge at high network latency. Of all the applications, these two have the least performance variation across the design space. In particular, they are the least occupancy-bound of all the applications.

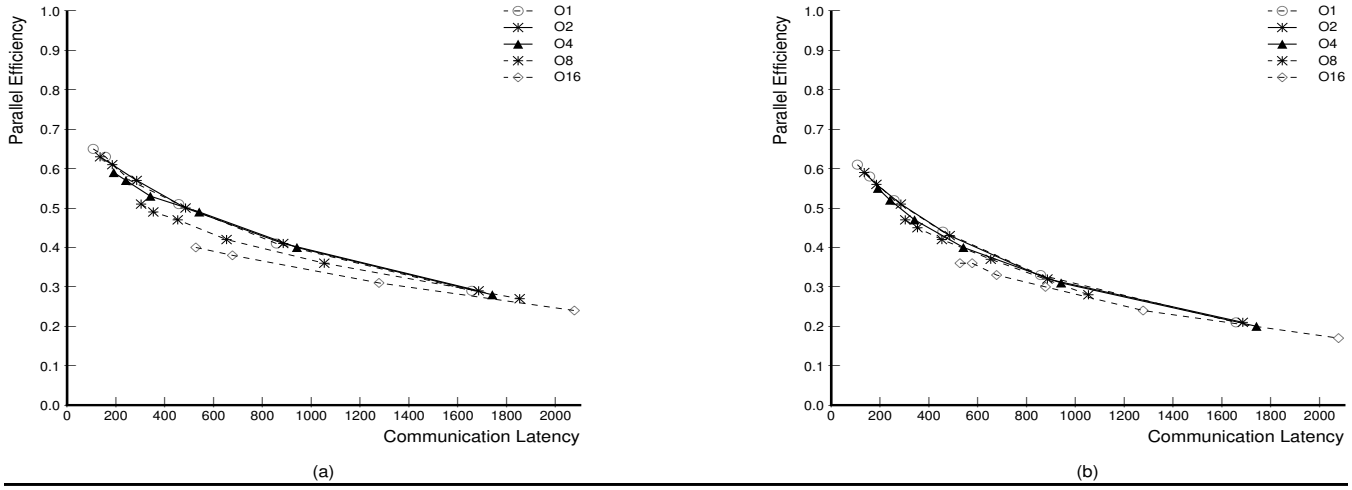

Figure 4.5. Base problem size results for (a) Barnes and (b) Water

As we expected, increasing network latency uniformly decreases overall performance across all the applications. Though there are some differences in the applications, particularly Barnes and Water, the surprising result is that consistently across all applications the contention component of controller occupancy has a significant performance effect, particularly at low values of network latency. In addition, the point at which the curves begin to flatten occurs at relatively small values of occupancy, typically either O4 or O8, and by O16 (general-purpose processor on the I/O bus) the curves are almost flat. Prefetching is often very effective at improving performance, but requires low occupancy controllers. It is also significant that even when prefetching, all of these applications have very low node-to-network bandwidth requirements.

5 Results for Increasing Problem Size

In the previous section we saw how parallel efficiency changed as we varied both network latency and controller occupancy for the fixed, base problem size. In designing DSM machines with high latencies or occupancies, we hope that these machines can be made to run at high parallel efficiencies simply by running somewhat larger problems. The question is how the problem size must be scaled to maintain 60% efficiency at higher values of latency and occupancy, and at what point do these problem sizes become unrealistic. In the results we present, we speak of problem size as being the size of the application's data set. However, we should keep in mind that in most real scientific applications the execution time grows more rapidly than the data set size for a variety of reasons [SHG93], and we shall comment on this fact as well.

To determine the problem size needed to attain a given efficiency, we need to know not only how the amount of computation and communication (T_{comp} and V_{comm}) scale with problem size (recall Eq. 4.1), but how the latency and contention costs of the average communication (T_L and T_C) scale as well. Clearly, T_L does not vary with problem size. The hope when using a high-occupancy controller is that the contention component T_C decreases as problem size increases. The question is whether this is true, or whether T_C increases or is independent of problem size. In Section 5.1 we once again examine the problem through a case study of FFT. Section 5.2 presents the results of increasing problem size for all of our applications.

5.1 Case Study

Consider for example FFT. The growth rate of computation T_{comp} with the number of points n (the problem size) is $O(n \log n)$. The growth rate of communication V_{comm} is $O(n)$. To maintain a fixed efficiency as the average communication cost T_L+T_C increases by a factor of s, we must increase the problem size at a rate that keeps the ratio of computation time to communication time constant. If T_C is not a function of problem size, then this simply means we must keep the computation to communication ratio ($\log n$) constant, which requires an increase in n by an exponential factor of 2^s.

Through simulation we gathered efficiency results for non-prefetched and prefetched FFT at two problem sizes: our base problem size and one that is four times larger. We look at the cross product of O1, O8, O16 and L1, L4, L16, which represent what we believe are realistic machine configurations (see Table 2.1). Increasing the problem by a factor of four did not increase the efficiency much, either with or without prefetching.

An important result is that at all occupancies, the average communication time remains constant in both the non-prefetched and prefetched versions of FFT. This means that the effects of contention do not decrease with an increase in problem size. Although this seems counter-intuitive given that the overall computation-to-communication ratio increases, there is a clear explanation as to why it happens. In many structured applications, communication is isolated in different phases from local computation. This separation has been incorporated in existing models of structured parallel programming such as the Bulk Synchronous Processing (BSP) model [Valiant90]. As a result, although the overall computation-to-communication ratio over the whole application increases with problem size, *within* the communication phases the ratio remains constant as problem size grows. Since contention depends on the rate or burstiness of communication, and that rate is independent of problem size, it follows that contention (T_C) is independent of problem size as well. Thus, FFT indeed requires an exponential increase in problem size to overcome the effects of increased latency or occupancy. Higher occupancies cause more contention, increasing the value of the exponent substantially.

This insight, that T_C like T_L is independent of problem size, allows us to predict the required problem sizes for FFT as l and o change, as long as we know how T_L and T_C change with l and o. Since FFT communicates through reads, we can use the average remote read miss time from the simulation results or our model in Section 4.2.1 to estimate $T_L + T_C$. The simulations also provide us with the constants for the computation time. Optimistically assuming perfect load balancing, Table 5.1 shows the minimum problem size needed to reach 60% efficiency for the nine selected combinations of l and o. Of these, we were able to simulate the required problem size for an occupancy of O1 and latencies of L1 and L4. The other numbers listed in Table 5.1 are predicted values, although the trends and contention effects have been validated.

Without Prefetching: It is clear from the table that increasing latency causes an exponential increase in the required problem size. The contention component of occupancy also has a big impact on the required problem size for FFT, even without prefetching. For example, if the O8 controller had the same contention component T_C as the O1 controller, but the communica-

Table 5.1. Minimum Problem Size Required for 60% Parallel Efficiency for both Non-Prefetched and Prefetched FFT

Controller Occupancy	Network Latency					
	Non-Prefetched			Prefetched		
	L1	L4	L16	L1	L4	L16
O1	2^{16}	2^{18}	2^{44}	2^{16}	2^{16}	2^{16}
O8	2^{22}	2^{28}	2^{58}	2^{18}	2^{18}	2^{20}
O16	2^{40}	2^{46}	2^{76}	2^{30}	2^{30}	2^{32}

tion latency corresponded to O8, the problem sizes for O8 in Table 5.1 would have been from 64 times smaller at L1 to 16 times smaller at L16. For O16, the problem sizes would have been 4096 times smaller at L1 and 1024 times smaller at L16.

With Prefetching: For the same l and o, the minimum problem size needed is much smaller than for the non-prefetched version, and depends much less on latency. However, once the latency becomes too large to be hidden, the growth rate is exponential in the amount that cannot be hidden. Contention still plays a critical role in determining the required problem size. With the same contention component of the O1 controller, the O8 controller would only need problem sizes 4 to 16 times smaller than those listed in Table 5.1. The O16 controller is far worse off: It would need a problem 16384 to 65536 times smaller.

For both versions of FFT, the problem size needed to achieve the desired efficiency at high controller occupancies is unreasonably large. The same is true of the non-prefetched version at high network latencies. Compromising the aggressiveness of a communication architecture, then, makes it be extremely difficult to achieve high parallel efficiencies for FFT.

5.2 Results for Other Applications

Radix: Radix, both with and without prefetching, is a more difficult application to retain good efficiency on than FFT, because the computation-to-communication ratio over the *entire* application is constant. This means that unless contention decreases, increasing the problem size should not increase the efficiency. We find that as the problem size increases, contention actually becomes worse, and then levels off fairly quickly. Contention worsens because the dominant communication in Radix is through writes (in a permutation), and these writes are bursty. As the problem size approaches the total amount of cache memory in the machine, it becomes increasingly likely that a given write will cause a cache line to be written back to the home, which is usually remote. This in effect doubles the number of messages that the communication controller at the home has to handle in the same amount of time. The situation does not get any better past this point, and in fact the irregularity of the communication causes some nodes to become hot-spots as problem size increases. Without prefetching, only the base architecture reached 60% efficiency. Even with prefetching, only the O1 and O2 controllers manage to reach 60% efficiency with the fastest network, and only the O1 controller can sustain 60% efficiency when a distributed MPP network is used.

LU: LU scales much better than either Radix or FFT. One reason is that the computation-to-communication ratio in LU grows linearly in the problem size ($O(n^3)$ computation versus $O(n^2)$ communication). LU therefore requires much smaller increases in problem size to reduce relative communication costs. The other is that the main bottleneck for LU on an L1, O1 machine is load imbalance and not communication. Increasing the problem size improves load balance quickly as well.

Table 5.2. Minimum Problem Size Required for 60% Parallel Efficiency for both Non-Prefetched and Prefetched LU

Controller Occupancy	Network Latency					
	Non-Prefetched			Prefetched		
	L1	L4	L16	L1	L4	L16
O1	800^2 (1.0x)	1250^2 (2.4x)	2900^2 (13x)	700^2 (1.0x)	800^2 (1.3x)	1200^2 (2.9x)
O8	1350^2 (2.8x)	1800^2 (5.1x)	3450^2 (19x)	850^2 (1.5x)	950^2 (1.8x)	1350^2 (3.7x)
O16	2000^2 (6.3x)	2400^2 (9.0x)	4100^2 (26x)	1000^2 (2.0x)	1100^2 (2.5x)	1500^2 (4.6x)

Like FFT and Radix, LU also communicates data in structured phases that have a constant computation-to-communication ratio. Consequently, contention does not decrease with increasing problem size, allowing us to predict the required problem size for machines with larger latencies and occupancies. Table 5.2 summarizes the results. For each entry, the value in parentheses is the ratio of the required data set size to that for an L1, O1 machine. Note that the computation *time* for LU scales a factor of n faster than the data set. This means that the time required for the "desirable" LU to complete grows more quickly than the table indicates. The time on an L16, O16 machine would be 133 times that on an L1, O1 machine without prefetching and 9 times with prefetching, even though the data set size required is only 26 times and 4.4 times larger, respectively.

Ocean: Ocean, which uses nearest-neighbor iterative computations including multigrid, also has a computation-to-communication ratio that scales linearly with problem size and a better load balance than LU. As Figure 5.1 shows, both the non-

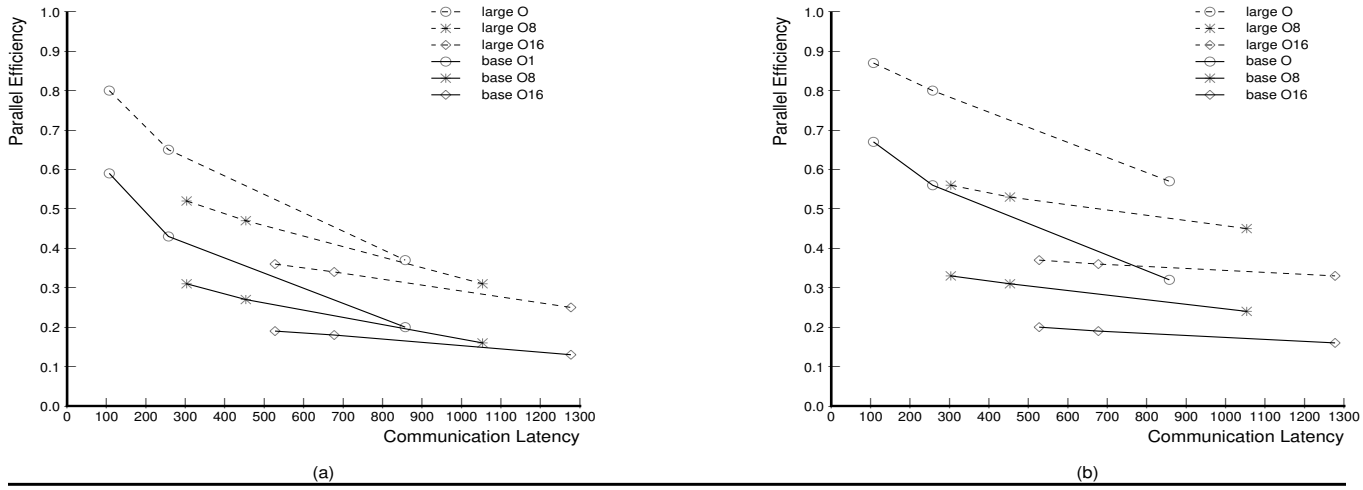

Figure 5.1. Efficiency in Ocean at two different problem sizes for both the (a) non-prefetched and (b) prefetched versions

prefetched and prefetched versions of Ocean scale much better than the previous applications. An important observation is that although even the higher occupancy curves increase substantially in efficiency with larger problem sizes, they still do not assume the shape of the lower occupancy curves. Once again, this is because Ocean also has structured communication, so contention does not decrease with increasing problem size. Table 5.3 shows the problem sizes required for 60% efficiency.

Table 5.3. Minimum Problem Size Required for 60% Parallel Efficiency for both Non-Prefetched and Prefetched Ocean

Controller Occupancy	Network Latency					
	Non-Prefetched			Prefetched		
	L1	L4	L16	L1	L4	L16
O1	258^2 (1.0x)	514^2 (4.0x)	1282^2 (25x)	258^2 (1.0x)	386^2 (2.2x)	642^2 (6.2x)
O8	642^2 (6.2x)	898^2 (12x)	1666^2 (42x)	642^2 (6.2x)	642^2 (6.2x)	770^2 (8.9x)
O16	1282^2 (25x)	1410^2 (30x)	2050^2 (63x)	1026^2 (16x)	1026^2 (16x)	1154^2 (20x)

Unlike LU, in Ocean both the data set size and the execution time nominally grow as $O(n^2)$ in the grid dimension. However, the implications of latency and occupancy for execution time are nonetheless more severe than for data set size. This is because increasing data set size also requires scaling other parameters (such as the accuracy used in the multigrid solver and the number of times-steps), which increase execution time further. In fact, the numbers for data set size in Table 5.3 are themselves optimistic, since a larger number of grid points causes more time to be spent in the multigrid equation solver, which has the highest communication to computation ratio and the worst load imbalance in the application.

Finally, the effect of contention on required problem size is much less for Ocean than it is for FFT. For example, for both versions of Ocean the required problem size for the O8 and O16 controllers would have been at most 4 times smaller if they had the same T_C as an O1 controller. Like FFT, the effect of contention is greater in the prefetched version of the code.

Barnes: Unlike the previous applications, Barnes does not have separate phases of communication and computation (though there are more structured versions of the application, written for message passing machines, that do [Salmon90]). As problem size increases, more computation is done between communications, so contention decreases. Since the computation-to-communication ratio also depends on the distribution of particles, predicting the problem size required for 60% efficiency is difficult. However, similar hierarchical N-body applications have an expected ratio that is linear in the problem size [Katz89]. This suggests that scaling hierarchical N-body applications to retain a desired efficiency should be relatively easy, if communication is the primary bottleneck. Unfortunately, the bottleneck is typically load imbalance, and it is difficult to predict how that improves with problem size since there are different computational phases with different levels of imbalance. Doubling the problem size for Barnes improved performance somewhat at higher occupancies, but not much. However, for the sizes of problems that are run on machines today, we expect that all of the machine configurations we study should perform quite well.

Water: Table 5.4 summarizes the minimum problem sizes required for Water. Water also has a computation-to-communication ratio that scales linearly with data set size. The effect of contention on required problem size is less in Water then it is in all the other applications. In fact, at ATM latencies the O8 controller achieves 60% efficiency at the same problem size whether or not it has its inherent value of T_C or it has the T_C of an O1 controller. Contention is only slightly more important at lower

network latencies. Thus, Water and Barnes are examples of occupancy-related contention not always being critical. However, note that the execution time for Water grows as the square of the data set size shown in Table 5.4.

Table 5.4. Minimum Problem Size Required for 60% Parallel Efficiency for Water

Controller Occupancy	Network Latency		
	L1	L4	L16
O1	512 (1.0x)	896 (1.8x)	1792 (3.5x)
O8	1152 (2.3x)	1536 (3.0x)	3072 (6.0x)
O16	3072 (6.0x)	3072 (6.0x)	6144 (12.0x)

Overall, significant increases in problem size are necessary for the lower-performance networks and controllers to achieve the desired efficiency, although the amount of increase varies depending on the specific type of application. There are many important classes of applications (transform methods, sorting) for which the efficiency lost by a less aggressive architecture—in latency or occupancy—is extremely difficult or impossible to regain by increasing problem size. In most of the applications, contention owing to the occupancy of the controller played an important role in determining the required growth in problem size, and the amount of contention was not reduced by increasing problem size. Finally, the implications of increasing latency and occupancy for execution time, which may be most important, are often more severe than those for data set size.

6 Conclusions

DSM machines can be characterized in terms of four fundamental parameters: network latency, controller occupancy, node-to-network bandwidth, and the number of processors. Using a fixed number of processors (64), and a high bandwidth interconnect, we evaluated the performance impact of latency and occupancy over a range of representative scientific applications. Our results showed that it is possible to achieve good parallel efficiency on machines with low-occupancy, hardwired or special-purpose communication controllers and low-latency MPP networks. As expected, network latency impacted overall performance, but its importance diminished with high-occupancy controllers, or when applications employed latency hiding mechanisms. In addition, the bandwidth requirements for the applications we studied were low in comparison with the node-to-network bandwidth of current MPP networks.

Our main result, however, is that the occupancy of the communication controller is critical to good performance in DSM machines. For machines with tightly-coupled MPP networks we found that controller occupancy has a large performance impact regardless of whether or not applications incorporated latency hiding techniques. For machines with loosely-coupled networks, we showed that without latency hiding, occupancy did not matter to overall performance. But with latency hiding, controller occupancy once again became a performance bottleneck. Since machines with high-latency networks will need to incorporate latency hiding whenever possible to obtain good performance, these results show that it is important to use low-occupancy communication controllers at any network latency.

Moreover, it was not the latency component of the higher occupancy controllers that caused performance degradation, but rather the contention component, even without latency hiding. This contention component proved difficult to model analytically, especially for applications that included latency hiding. We also found that several important classes of applications communicate in "bulk synchronous" phases where the computation-to-communication ratio is constant. As a result, increasing the problem size did not alleviate contention.

Finally, our results showed that for many classes of applications, it is extremely difficult for architectures with higher values of network latency or controller occupancy to achieve high parallel efficiency. That is, the problem size needed to maintain the desired efficiency quickly becomes unreasonable. There are applications that attain the desired efficiency with reasonable data set sizes, although for many of these applications the execution time scales much faster than the required data set size.

The tendency among DSM designers has been to focus on latency and network bandwidth as the important performance issues in the communication architecture. Our results demonstrate that the occupancy of the communication controller is just as important to overall performance, if not more so. Designers should therefore pay careful attention to controller occupancy when making decisions about using commodity parts in their communication architectures.

Acknowledgments

This work was supported by ARPA contract N00039-91-C-0138. We would like to thank Steven Woo for his work on the SPLASH-2 application suite. We would also like to thank Anoop Gupta, Jeffrey Kuskin, and David Ofelt for their comments on early drafts of this paper.

References

[ACD+91] Anant Agarwal et al. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. MIT/LCS Memo TM-454, Massachusetts Institute of Technology, 1991.

[CKP+93] David Culler et al. LogP: Toward a realistic model of parallel computation. In *Proceedings of the Principles and Practice of Parallel Processing*, pages 1-12, 1993.

[Golds93] Stephen Goldschmidt. Simulation of Multiprocessors: Accuracy and Performance. Ph.D. Thesis, Stanford University, June 1993.

[HKO+94] Mark Heinrich et al. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274-285, San Jose, CA, October 1994.

[HS94] Chris Holt and Jaswinder Pal Singh. Hierarchical N-Body Methods on Shared Address Space Multiprocessors. *SIAM Conference on Parallel Processing for Scientific Computing*, February 1995, to appear.

[Katz89] Jacob Katzenelson. Computational Structure of the N-body Problem. *SIAM Journal of Scientific and Statistical Computing*, pages 787-815, July 1989.

[KOH+94] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302-313, Chicago, IL, April 1994.

[KSR92] Kendall Square Research. KSR1 Technical Summary. Waltham, MA, 1992.

[LLG+92] Daniel Lenoski et al. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63-79, March 1992.

[RLW94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 325-336, Chicago, IL, April 1994.

[Roth93] Edward Rothberg. Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization. Ph.D. Thesis, Stanford University, January 1993.

[RSG93] Edward Rothberg, Jaswinder Pal Singh and Anoop Gupta. Working Sets, Cache Sizes, and Node Granularity for Large-Scale Multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 14-25, San Diego, CA, 1993.

[Salmon90] John K. Salmon. Parallel Hierarchical N-body Methods. Ph.D. Thesis, California Institute of Technology, December 1990.

[SFL+94] Ioannis Schoinas et al. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297-306, San Jose, CA, October 1994.

[SHG93] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Scaling parallel programs for multiprocessors: methodology and examples. *IEEE Computer*, July 1993.

[SWG+94] Jaswinder Pal Singh et al. The SPLASH-2 Suite of Parallel Applications, Technical Report to appear, Stanford University.

[Valiant90] Leslie G. Valiant. A Bridging Model for Parallel Computation. In *Communications of the ACM*, 33(8):103-111, August 1990.

[WSH94] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 219-229, San Jose, CA, October 1994.