Thread Migration and Communication Minimization in DSM Systems

Kritchalach Thitikamol and Pete Keleher
University of Maryland
(kritchal | keleher)@cs.umd.edu
Key words: DSM, load-balancing, thread migration.

Abstract

Networks of workstations are characterized by dynamic resource capacities. Such environments can only be efficiently exploited by applications that are dynamically re-configurable. This paper explores mechanisms and policies that enable online reconfiguration of shared-memory applications through thread migration.

We describe the design and preliminary performance of a DSM system that performs online re-mappings of threads to nodes based on sharing behavior. Our system obtains complete sharing information through a novel correlation-tracking phase that avoids the thread thrashing that characterizes previous approaches. This information is used to evaluate the communication required by a given thread mapping, and to predict the resulting performance.

1. Introduction

Meta-computer environments can be characterized by distribution, heterogeneity, and changing resource capacities. Meta-computers consist of networks of machines, some of which might be shared memory multiprocessors. Distributed and parallel applications can be run in such environments, but usually not as effectively as on shared memory multiprocessors. Part of the reason is that meta-computers are often non-dedicated, forcing the individual threads of a parallel application to compete with other jobs for resources. Another part of the problem is the individual machines often have different capabilities. Finally, such environments are highly dynamic.

Parallel applications must be dynamically reconfigurable in order to run efficiently in such environments. Reconfigurability can be explicit in the application's structure. However, this approach is unlikely to be portable, and puts a large burden on application developers. A more general approach is for the runtime system to implement reconfiguration transparently to the application. This paper describes such a reconfiguration mechanism in the D-CVM (*Dynamic* Coherent Virtual Machine) [1] distributed shared memory (DSM) system.

D-CVM implements reconfiguration through thread migration. Thread and process migration has long been used as a load-balancing mechanism [2, 3] in parallel and distributed systems. However, DSMs usually have much higher communication requirements than message-passing systems, implying that good thread migration policies in this domain must also account for communication behavior.

Consider a page-based software DSM. If threads on distinct nodes of a system share data on a specific page *x*, sharing traffic can only be eliminated by co-locating both threads on the same node. Rather than just moving pages to threads that request them via network faults, thread migration allows the computation to be moved to the data instead.

Creating a good mapping of threads to nodes requires several distinct steps. First, we must be able to evaluate the load distribution of a given mapping. This generally requires a way of estimating threads' computational needs and nodes' computational capacities. This distribution must take into account both *parallelism*, or how many nodes we are exploiting, and the *balance*, or how uniformly the load is distributed across those nodes. Second, we must be able to evaluate a mapping's communication cost. This problem reduces to identifying the sharing between threads that are located on distinct nodes. Finally, there must be a way to combine these metrics into a single algorithm.

In general, neither parallelism maximization nor communication minimization can proceed in isolation. Assume that work is distributed equally across four threads, p_1 and p_3 on one node and p_2 and p_4 on another. This distribution is clearly "balanced" in the sense that each node has the same amount of work. However, the communication is just as clearly not optimal if each thread communicates with the neighbors implied by the thread ids (i.e. p_2 communicates with p_1 and p_3). A better mapping of threads to nodes would be p_1 and p_2 on the first node, and p_3 and p_4 on the second. By re-mapping threads to nodes we reduce communication by a factor of three without affecting the load balance.

We illustrate these issues by describing D-CVM's thread-mapping mechanism. D-CVM applications consist of a single process and one or more user-level threads on each node of the system. Each thread has a stack and other D-CVM context. All threads share global data uniformly.

We do not require threads to have uniform amounts of work. We also do not assume a dedicated environment. In fact, we expect reconfigurable systems like D-CVM to be most useful in the meta-computer environments discussed above. While our approach aggressively exploits the underlying DSM's mechanisms in order to track threads' sharing behavior, our techniques are not specific to D-CVM's consistency protocols. Moreover, the heuristics that we use to map threads to nodes are relevant to the load-balancing of message-passing applications as well.

Much of this paper describes the specific mechanisms used in D-CVM, but the ultimate goal is to explore the design space of thread migration policies. To this end, we discuss alternatives and tradeoffs at each relevant portion of the paper. The application domain assumed in this paper is that of highly iterative scientific code running on top of software DSMs.

Section 2 describes the hardware and software environment used in our experiments. Section 3 describes the D-CVM mechanisms used to migrate threads across machine boundaries. Section 4 describes D-CVM's approach to maximizing parallelism and minimizing load imbalance. Section 5 describes D-CVM's use of active correlation-tracking to obtain sharing information, and the design and preliminary performance of several different thread-

mapping heuristics. Finally, Section 6 describes related work and Section 7 concludes.

2. Experimental Environment

The DSM target used in this work is a version of D-CVM [4], modified to handle migratory threads. D-CVM is a page-based, user-level DSM that implements multiple-writer lazy release consistency (LRC) [5], which is a derivation of *release consistency* [6]. In release consistency, a processor delays making modifications to shared data visible to other processors until special *acquire* or *release* synchronization accesses occur. The propagation of modifications can thus be postponed until the next synchronization operation takes effect. LRC allows the propagation of modifications to be further postponed until the time of the next acquire. Programs produce the same results with these memory models as with more conventional memory models, provided that all synchronization operations use system-supplied primitives, and that all conflicting shared accesses are ordered by synchronization or program order. In practice, most shared-memory programs require little or no modifications to meet these requirements. From the perspective of the mechanisms discussed in the rest of this paper, the most important attribute of D-CVM is that its protocols tolerate false sharing [7] well.

The majority of our experiments were run on an eight-processor SP-2. Each node is a 66.7 MHz POWER2 processor with 64K first-level caches and 128 MBytes of memory per node. The processors are connected by a 40 MByte/sec switch. The operating system is AIX 4.1.4. D-CVM runs on UDP/IP over the switch. Lock acquires are implemented by sending a request message to the lock manager, which then forwards the request on to the last requester of the same lock. This may take only two messages if the manager is also the last owner of the lock. Simple 2-hop lock acquires take 779 µsecs, while 3-hop lock acquires take 1185 µsecs. Simple page faults across the network require 1576 µsecs. Page fault times are highly dependent on the cost of *mprotect* calls, 15 µsecs, and the cost of handling segmentation violation (segv) signals at the user level, 120 µsecs. Minimal 8-processor barriers cost a minimum of 1176 µsecs.

The applications used in our study are SOR, a simple nearest-neighbor stencil with a 1024x1024 point grid, FFT, an implementation of a 3-D Fast Fourier Transform solver with 64x64x64 points, and barnes, ocean, and Water-Nsquared (water) from the SPLASH–2 benchmark suite. Barnes was modified by Rajamony [8] to decrease synchronization granularity and solves equations for 16k bodies. Ocean was run with 256x256 molecules and water with 512. In all cases, our system consists of 32 threads distributed across eight nodes. Since our environment is homogenous and all threads perform equal amounts of work, we place four threads on each node.

2.1 Thread Representation

D-CVM's thread mechanism is based on the NewThreads [9] user-level thread library. One of the primary distinctions among thread packages is whether they support kernels- or user-level threads [10]. Kernel-level threads are scheduled and otherwise managed by the kernel directly. One advantage of this arrangement is that the kernel can switch to other threads when the active thread makes a blocking system call. The kernel can also integrate the scheduling of threads into the overall scheduling of processes on the machine. Each kernel-level thread can potentially compete with threads of other processes, while all user-level threads of a single process must compete for resources as a single unit.

The disadvantages of kernel-level threads include poorer performance and a lack of flexibility. User-level threads are usually faster because thread operations do not require kernel calls. They are more flexible because the only limitations are usually those imposed by the hardware. D-CVM is ideally suited for user-level threads because it does not use blocking I/O calls.

3. Thread migration mechanisms

This section discusses D-CVM's thread migration mechanism, together with the implications of alternative design choices. Transparent migration of threads across node boundaries requires that the destination environment be "equivalent" in some sense to the source environment. This equivalence has two parts: the *node environment*, and the *data view*.

3.1 Node Environment

A thread's node environment includes all aspects of the application's runtime environment that are related to the particular node on which the thread is running. These aspects include environmental variables, and resources allocated or read from the operating system. These problems have been studied at length elsewhere [3, 11]. Moreover, these problems are not specific to thread migration systems, but apply to any distributed environment, including PVM, MPI, and any DSM. To the authors' knowledge, no DSM system explicitly addresses this issue in any general way. We therefore follow the standard practice of requiring all system calls to occur during the initialization phase. Since migration is disabled during initialization, all system calls occur before any threads are re-located. None of our applications needed to be modified in order to obey this restriction.

3.2 Data View

Data seen by a thread includes values in registers, the stack, non-shared global data, and the shared data segment(s). Registers are easily copied from one machine to another. We do not have to worry about volatile variables or compiler optimizations since migration only occurs when threads voluntarily yield the processor through a procedure call [12].

Non-shared global data

Non-shared global data refers to heap data and to statically allocated data. D-CVM only ensures consistency of data explicitly allocated through D-CVM calls. Hence, heap and statically allocated data is not consistent across nodes. The result is that threads on a single node see the same copies of non-shared data, whereas threads on different nodes see distinct versions. There are two general approaches to this problem. The first is to ensure consistency of this data by explicitly handing it off to the DSM system. The second is to disallow any non-shared data that is processor-dependent. We chose the second approach because the first requires extensive (and non-portable) link-time manipulation in order to segregate the application's global data from the library and DSM data.

		Stack Size	Source			Destination			
			Send	Other	Total	Reply	Scan	Other	Total
SP2 (switch)	r	1704	1272	14	1286	157	149	5	311
	s	1704	1261	15	1276	116	270	5	391
Alpha (ATM)	r	832	2226	122	2348	180	103	44	327
	s	832	1782	125	1907	181	127	46	354
UltraSp (ether)	r	1280	2475	72	2547	119	185	7	311
	s	1280	2457	72	2529	121	216	8	345

Table 1: Migration Costs (*u*secs)

Stacks

Stacks can easily be copied from one machine to another. The major complication is dealing with pointer values located in the stack. *Self-referential pointers* become inconsistent if thread migration causes a stack's address to change. There are essentially three approaches to dealing with this problem. First, compiler or language support could maintain enough type information for pointers to be reliably identified. Such an approach is inherently specific to a single language or runtime system.

Second, systems using the "scan" approach attempt to dynamically identify pointers in the stack by scanning the stack for possible pointer values [13]. The probability that any data value is misidentified as a pointer is low, but non-zero. Any such misidentifications could contaminate ongoing computations without obvious symptoms, causing erroneous results to be accepted as correct.

Finally, the "reserve" approach requires systems to reserve unique virtual addresses for the stack of each thread in the system. With a unique stack address, a thread can migrate to any node without needing to change the stack's address. This causes consumption of the address space to scale with the number of threads in the entire system. With 32k stacks, 128 threads/node, and a 16-node system, this approach uses only 64 megabytes of the address space. While large, this is certainly usable on current systems. The advent of 64-bit addresses makes possible a probabilistic approach that consists of allocating the stack at a random 64-bit address. With good random number generators, the possibility of two threads being allocated to the same address is exceedingly remote. Any collisions that do occur can easily be detected and dealt with by prohibiting that particular configuration.

Table 1 shows the cost of a single thread migration on D-CVM for the "scan" and "reserve" (labeled 's' and 'r', respectively) migration mechanisms. We present results for three different architectures: the 66.7 MHz Power2 processors over the SP-2's 40-MByte switch, 275-MHz 21164 Alpha processors over a 155 Mbit ATM, and UltraSparcs over a 10MBit Ethernet. UDP was used in all cases. The table gives the size of the migrated thread's stack, and the runtime cost seen by the source and destination nodes. "Send" shows the cost of creating and sending the migration message. "Reply" shows the cost of reading the request and sending the reply message. Note that these communication costs are relatively large, reflecting our use of kernel-based IP primitives. These costs can be amortized by including more than a single thread in a message. Communication costs could be reduced by more than an order of magnitude through use of zero-copy protocols such as BIP [14] running on top of a Myrinet [15]. "Scan" is the cost of scanning the stack for pointer values, updating any pointers found, and copying the stack into its new location. This cost can be considerable, but is still insignificant compared to communication overheads. For the "reserve" approach, the "scan" column just refers to the cost of copying the stack out of the migration message. Both columns labeled "Other" consist of local bookkeeping, such as manipulating local thread queues to reflect incoming or outgoing threads.

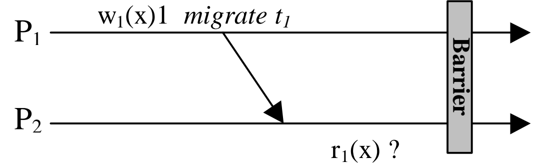

Figure 1: Migration and Consistency

In all cases, the cost of thread migration compares favorably with the cost of fetching a remote page. While the stack sizes are small, we believe them to be typical of scientific codes. Larger stacks could be handled efficiently by demand fetching all but the top pages. The migrations were timed in Water-Nsquared.

Shared data

Finally, each node has a specific perspective on the consistency of shared state, shared by all local threads. Migration of a thread from one node to another requires that the view of the destination be as advanced as that of the source, just as with synchronization. This usually only has to be addressed explicitly by systems that implement relaxed consistency models. As discussed in Section 2, such memory models often delay the *performance* of specific shared accesses in order to reduce overall communication requirement. Figure 1 illustrates the problem. Assume that each of processes P_1 and P_2 contain at least one thread. Thread t_1 of process P_1 migrates to P_2. Before migrating, however, t_1 modifies shared data variable x. If the migration completes before the consistency information arrives, a subsequent read by the same thread at its new location could return an old value. In an LRC protocol like D-CVM's, notice of the modification would not arrive until the subsequent barrier. Hence, the read could return a stale value.

The inconsistency can be addressed by appending consistency information to the messages that migrate the thread. Alternatively, a thread's source processor can *release* to the thread's destination processor before the thread is activated on the new processor. Our system takes the latter approach, moving threads only at predefined synchronization points.

4. Parallelism and load balance

Our overall goals are to maximize parallelism, to minimize load imbalance, and to minimize communication. The combination of these goals is a form of the *multi-way cut* problem, and is NP-hard. While good approximation schemes have been found for the general form of the communication minimization problem [16], our problem is complicated by the fact that we must also address load balancing and parallelism. We therefore decompose our problem into three distinct tasks:

(1) determining the number of nodes that will result in the greatest speedup,
(2) minimizing load imbalance by adjusting the number of threads per node, and
(3) minimizing communication by taking sharing into account when mapping threads to nodes.

Ideally, of course, these tasks should be performed at the same time because they are all interrelated. Since the amount of com-

munication can affect an application's efficiency, the mapping of threads to nodes could affect the number of nodes at which the best performance is achieved. However, addressing all of these issues simultaneously can make the complexity of the required algorithms unmanageable.

4.1 Number of nodes

We determine the number of nodes on which to run a parallel application with the help of an initial guess provided by the user at startup time. As the application continues to execute, the system tracks processor efficiencies by measuring the proportion of time spent waiting on communication and synchronization. These efficiencies are compared to system-wide high-water and low-water thresholds. If the efficiency is below the low-water threshold, we assume the application would be better off running on fewer processors. The converse is true for the high-water mark.

Consider the speedup curve shown in Figure 2. The diagonal line represents linear speedup, so the degree to which the speedup curve diverges from the line represents the inefficiency with which the application is being executed. At point A, the efficiency is high, so we can reliably assume that increasing the number of processors will improve overall speedup. At point B, however, the efficiency is very low, and we can assume that decreasing the number of processors will either increase the speedup, or not hurt it significantly. In either case, we can check the result of changing the number of processors by comparing efficiencies before and after the change.

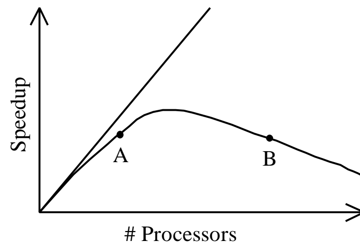

Figure 2: Speedup Curve

Our thresholds were chosen to maximize a single application's speedup, but they may also be chosen to improve overall system throughput. Maximizing speedup at all costs might not be the best choice if the slope of the curve in Figure 2 is a very low positive number. Currently, our system uses 80% for the upper efficiency threshold and 20% for the lower.

This simple heuristic will perform poorly if an application's speedup curve has local minima. For instance, some applications perform poorly unless the number of threads is a power of two. The most general approach to this problem is to provide an API that allows the system to be informed of application-specific scheduling information [17].

4.2 Thread capacity

A truly general load-balancing facility must be able to accommodate both heterogeneous node capacities and threads that perform varying amounts of work. The capacity of a node depends on both the intrinsic capability of the node, as well as the proportion of this capacity being consumed by other jobs. The *residual capacity* of a node is a dynamic measure that can evolve during the course of an application's execution. Thread resource requirements might also vary.

Systems should ideally be able to estimate both node capacities and thread resource requirements at runtime. Unfortunately, obtaining both at the same time is truly a hard problem. A rough estimate of residual capacities can be made if relative thread requirements are known. Conversely, knowledge of residual capacities and fine-grained tracking of CPU usage can be used to estimate the resource requirements of each thread. Furthermore, the problem is greatly simplified in dedicated environments that have homogeneous nodes, or with applications whose threads perform equal amounts of work. We distribute threads uniformly in our experiments because both simplifications apply for our environment and applications.

However, we can develop a more general formulation by letting w_i represent the work to be done by thread i, and R_j represent the residual capacity of node j. We deliberately leave the units of w_i and R_j unspecified, as only their relative magnitudes matter.

$$W_i = \left(\frac{R_i}{\sum_{j=1}^{N} R_j} \right) \sum_{j=1}^{T} w_j \qquad (1)$$

Assuming that there are N nodes and T threads, then the amount of work W_i that should be assigned to node i is:

Note that W_i is some abstraction of work, not necessarily the number of threads. Also, this formulation ignores the influence of communication on the cost of performing each chunk of work. This information is not available if we have not yet mapped threads to nodes. The impact of this general formulation on thread mapping is discussed in Section 5.4.

5. Thread correlation and mapping heuristics

The final task identified in Section 4 is to map specific threads to nodes. This decision would ideally be made with global information. However, dependence on global state introduces the issue of timeliness into the system, as well as new sources of overhead. More importantly, not all decision processes deal well with incomplete or stale information.

Nonetheless, we use global information for three reasons. First, the amount of state needed to summarize sharing behavior for our heuristics is only $i*n^2$, where i is the size of a short integer and n is the number of threads. This quantity of information can easily be carried in a single message for realistic numbers of threads. Second, sharing information can be piggybacked on top of existing global synchronization operations, e.g. barriers. The only new messages are those that migrate threads between nodes. Finally, thread mapping can be used to adapt to dynamically changing environments. However, the variation that we wish to take advantage of will be at least on the scale of tens of seconds, not milliseconds. The cost of a single decision can therefore be amortized across a relatively large amount of computation.

5.1 Cost evaluations

Our goal in mapping threads to nodes is to minimize communication. Communication occurs in D-CVM for two reasons: synchronization and data fetches. Modulo application non-determinism, communication is minimized by moving communicating threads to the same (or nearby) nodes. Since our system has a uniform remote access cost, communication between a pair of threads can only be reduced or eliminated by co-locating them. Co-locating all threads on the same node would eliminate all communication, but presumably not produce the best overall performance.

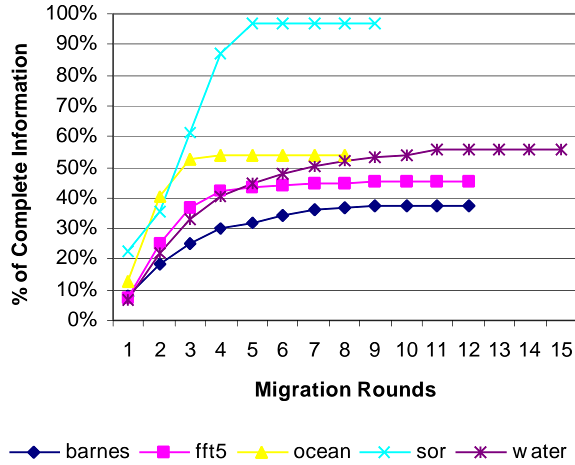

Figure 3: Passive Information-Gathering

Changing the mapping of threads to nodes can have unanticipated effects on performance, such as the influence of co-location on the actions performed by the underlying DSM. However, these secondary effects are likely to scale with communication costs, so we can treat them as one in our system.

A comprehensive cost function that summarizes the desirability of a given mapping of threads to nodes must take parallelism, load balance, and communication requirements into account. However, we assume both parallelism and load balance have been addressed through the steps outlined in Section 4. Hence, the only remaining characteristic to be summarized is communication cost.

Communication cost can be measured in a number of ways. Seemingly obvious metrics include messages counts, and the total amount of communicated data. The problem with these metrics is that they only reflect sharing between nodes, not between the individual threads on each node. For example, a large amount of communication between nodes n_i and n_j is not sufficient to determine which threads on those nodes are sharing resources.

We therefore use information from the underlying DSM protocols to generate metrics that measure sharing between individual threads. Data sharing between threads can be tracked by correlating accesses to shared memory by the threads. Two threads that frequently access the same shared pages can be presumed to share data. We define a *density function* as the access rate of thread i to page p. The correlation of two threads over page p can be computed as the product of the density function of the two threads for page p. The overall correlation of the two threads, then, is the sum of the correlations for each page in the system [18]. Unfortunately, page-based DSMs have no efficient way of deriving density functions because they can not track individual accesses. Instead, accesses are tracked only at the granularity of a page. Systems that capture shared writes through binary rewriting [19] rather than page faults could presumably capture accurate densities. However, this would add overhead to all writes unless function cloning is used.

More generally, the notion of an access *rate* is difficult to capture. Once a page has been mapped locally, subsequent accesses to the page proceed transparently. Hence, we can not track the rate of individual accesses. A rough estimate could be obtained by tracking the average length of time a given page remains invalidated before being revalidated. Unfortunately, this estimate could be greatly affected by intervening events. For instance, 100 *u*secs is a long interval if it contains only local accesses. However, a remote access can take milliseconds. Such events make it unlikely that the rate of page revalidation would accurately reflect the access rate.

We therefore use the number of pages shared across node boundaries as a predictor of the amount of communication that a mapping of threads to nodes will produce. We define the *correlation* of a pair of threads as the total number of pages shared between the threads. We define the *cut cost* of a given mapping of threads to nodes as the sum total of all thread-pair correlations for which the component threads are on distinct nodes. Note that this definition could be extended to deal with non-uniform communication networks by multiplying thread correlations by link-specific coefficients.

5.2 Intra- versus inter-node information

The obvious way to determine the amount of sharing between threads is to track DSM page faults. Tracking page faults can give a rough estimate of system sharing. However, this *passive tracking* approach only identifies sharing between threads of different processors. Since all local threads share the same access rights to each page, multiple local threads can access a page without causing more than a single page fault. No information is gained about sharing between threads located on the same node. Hence, decisions must be made with only partial information, often leading to bad long-term choices. These bad choices are discovered only after the threads have migrated to other processors. Once a thread migrates from a host, the interactions between that thread and those left behind become visible in the form of network page faults. These faults can be used to identify threads that should be moved back to their original position, resulting in the ping-ponging of threads (or *thread thrashing*) across the system.

Figure 3 shows the percentage of complete sharing information gathered by the passive tracking approach as a function of the number of migration rounds. Even at the end of the migrations, the passive tracking only comes close to obtaining complete information for SOR, by far the least complex of our applications. Each round consists of gathering page fault information for an iteration of the application, followed by migrating threads to new locations.

The applications averaged slight more than six rounds of migrations before stabilizing, although Figure 3 shows all rounds in which new information is gained. The term "stabilizing" is used advisedly. Recall that passive correlation tracking only learns about the first local thread to access a page during any synchronization interval. This means that the speed at which information is accumulated is non-deterministic. A configuration might appear optimal for several iterations before the non-deterministic scheduling of threads reveals new information. This happened for water, where migrations occurred eight times, followed by two iterations in which no better configuration was found, followed by one last iteration in which new information caused a final round of migrations to occur.

5.2.1 Active correlation tracking

Thread-thrashing can be avoided if we have information about correlations between local threads before the re-mapping takes place. D-CVM obtains this information through an *active correlation-tracking* phase, which provides complete correlation information for all thread pairs, local and remote. The algorithm uses two data structures: per-page correlation bits, and per-thread access bitmaps:

1) At the start of the tracking phase, all pages are read-protected and the *correlation bit* of each page is set. The pages' previous states are saved in the page structure. The thread scheduler is placed in a special mode that prevents thread-switching from occurring until the next barrier has been reached.
2) At each access fault for a page whose correlation bit is set (a *correlation fault*), the corresponding bit in the per-thread *access bitmap* is set, and the correlation bit is reset. The page is then returned to its original state and the fault handler returns. If the access type would have caused a violation outside the correlation-tracking phase, a second fault occurs and is handled normally.
3) At the next barrier, the system switches to the next thread, sets all correlation bits again, and once again read-protects all pages. This thread is then allowed to proceed in the same manner as the previous thread.
4) The tracking phase ends when all threads reach the next barrier. At the end of the correlation-tracking phase, all correlation bits are reset and untouched pages are returned to their correct states.

After the tracking phase has ended, the per-thread access bitmaps specify exactly which pages each thread accessed during the tracking phase.

Note that we do not distinguish between read and write accesses. The reason is that co-locating two consumers of the same data gives us the same benefit as co-locating a producer-consumer pair. Page faults are avoided in both cases. The sole exception is that we filter out pages that are written only during initialization because read-only pages do not cause page traffic.

The tracking phase has two primary forms of overhead. The most obvious is the cost of the correlation faults. This cost scales with the number of pages accessed locally, and the degree of sharing between the local threads. Given a system with n nodes and p pages, the local threads will usually access at least p/n pages, more if there is a large amount of data sharing between threads. Local sharing increases the number of faults because each shared page incurs more than one page fault. However, the cost of correlation faults on distinct nodes is incurred in parallel.

The second cost results from disabling the thread scheduler during the tracking phase. Turning off the thread scheduler eliminates the latency toleration advantages of per-node multithreading. The performance impact of losing this amount of latency toleration is usually on the order of 10-15% [20], and is only incurred during the active correlation-tracking phase.

We could implement active correlation-tracking without turning off the thread scheduler. However, pages would need to have a correlation bit for each local thread. Furthermore, each thread switch would require the state of all pages to be updated to correspond to the new thread's correlation bits. The impact of these changes would likely overwhelm the advantages of using the thread scheduler.

	Slowdown While Tracking			Correlation Faults	
	rs6000 AIX	Alpha UNIX	Pentium II Linux	/ remote miss	/ shared page
barnes	2%	3%	1%	0.12	6.93
FFT	6%	10%	2%	0.10	1.51
ocean	11%	18%	4%	0.25	9.52
SOR	22%	36%	8%	4.46	1.34
water	1%	2%	0%	0.08	6.55

Table 2: Correlation Tracking Overhead

Table 2 shows the runtime overhead of active correlation tracking for three local platforms. This overhead is calculated from the number of correlation faults and the cost of handling correlation faults on each platform. The platforms shown here are an SP-2 running AIX 4.2, a cluster of 266 MHz Alpha multiprocessors running Digital Unix 4.0, and a cluster of 266 MHz Pentium II's running Linux 2.0.32. The worst overheads are on the Alpha platform, but even here the maximum overhead only reaches 36%. The maximum overhead on Linux is only 8%, showing the value of fast user-level signal-handlers.

The last two columns of Table 2 show the number of correlation faults as a function of the number of remote misses in the default case, and of the number of shared pages. "Correlation faults per shared page" gives an indication of why SOR is such a pathological case for the correlation-tracking mechanism. SOR is a nearest-neighbor application, and therefore only exchanges shared updates for border rows. However, the correlation-tracking mechanism incurs faults on all pages, even the interior pages that are not shared.

The last column shows the number of correlation faults as a function of the number of shared pages. This number gives a rough estimate of the number of threads sharing each page. A '1' would indicate no sharing, larger numbers indicate the degree of sharing between threads. For example, Ocean has a total of 3200 pages. If the data on these pages were distributed evenly across 32 threads, each would be responsible for 100 pages. However, Table 2 indicates that each thread touches *952* pages during each iteration. Nonetheless, Ocean's overhead is only 18% for the Alphas, and 4% for the Pentium II machines.

Nonetheless, the tracking process is too costly to perform often. However, correlation tracking only has to be repeated in response to changes in the environment or the application. The cost can therefore be amortized over the rest of the computation. For example, even SOR's overhead on the Alpha platform might be tolerable if amortized across ten iterations, and would certainly be tolerable if each application performed 100 such iterations.

As noted above, all overhead of the tracking phase is incurred locally, and in parallel across nodes of the system. This implies that the absolute runtime cost of the tracking phase should not increase as the number of nodes is increased. This is in contrast to the passive ping-ponging approach (see Section 5.2), in which increasing system size would probably increase the number of thread migrations. Additionally, the system might also take longer to settle.

The absolute cost of this tracking phase *is* sensitive to the overall amount of sharing in the system. Since sharing means that multiple threads are accessing the same pages, such sharing increases the total number of segmentation violations. Systems with

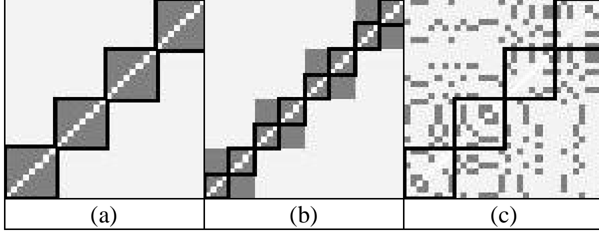

Figure 4: 32-thread FFT, 2^6 x 2^6 x 2^6 - (a) on four nodes, squares indicate thread sharing that does not cause network communication, (b) on eight nodes, as above, (c) randomized thread assignments for four nodes

little or no sharing are therefore insensitive to the number of threads. However, as sharing increases, the number of threads can become significant.

5.3 Correlation maps

At the end of the tracking phase, each node has complete access bitmaps for each local thread, but only incomplete information about remote threads. The local bitmaps are sufficient to determine a local thread's affinity for an entire remote node. Systems with a high degree of multi-threading might find this useful in allowing nodes to unilaterally send threads elsewhere. Systems like D-CVM, on the other hand, generally use rather coarse-grained threads. Hence, thread exports usually need to be balanced by an equal number of thread imports. Good decisions about which thread(s) should be imported usually require global information.

D-CVM enables a global re-mapping process by appending all access bitmaps to the next barrier arrival message. Once all processes have arrived, the master computes thread correlations by counting the number of pages accessed in common by each pair of threads. The set of all correlations can be depicted visually in a two-dimensional *correlation map*, which shows increasing thread pair correlation by darker shades of gray.

Figure 4 shows 32-thread correlation maps for FFT. Figure 4 (a) shows a mapping of eight threads to each of four nodes. We have added node boundaries in the form of black outlines, i.e. the box in the lower left shows that threads 1-8 are on the same node. The map shows a well-defined structure in which all of the dark areas are concentrated along the diagonal, and contained within the outlines that represent individual nodes. Hence, we infer that this mapping would eliminate most communication. Any dark areas outside the node outlines imply network communication because they represent sharing by threads located on distinct nodes.

By contrast, Figure 4 (b) shows that a mapping of four threads to each of eight nodes captures only half of the dark areas. The implication is that a four-by-four mapping would have much less communication than an eight-by-two mapping. What is not clear from the map is to what extent this communication advantage would translate into a performance advantage. Hence, our current heuristics would have no way of identifying the four-by-four mapping as a good one. Nonetheless, Table 3 shows that this is, indeed, the case. The four-node configuration has fewer than half the number of remote misses, messages, and overall bandwidth requirements of the eight-node configuration. These advantages translate into an overall running time that is 8% faster

	Time (secs)	Bandwidth (bytes)	Msgs	Misses
4x8	2.39	7691340	1904	973
8x4	2.61	15644028	4052	2267

Table 3: FFT Configurations

for four nodes than for eight nodes. Perhaps, more importantly, the four-node configuration consumes only half the resources of the eight-node configuration.

Finally, Figure 4 (c) shows a correlation map resulting from a random mapping of threads to four nodes. Note that the majority of the dark areas are outside node boundaries, and communication behavior can be expected to be worse than for the mapping of Figure 4 (b).

There are essentially two empirical approaches to estimating the relative importance of communication and parallelism. The direct approach is to actually run the application in different configurations, and to search for the fastest configuration [21]. Unlike the original system in which this technique was used, reconfiguration in D-CVM (i.e., thread migration) is relatively inexpensive. Hence, this approach might be practical for long-lived, computationally expensive applications.

The second approach is to measure component communication costs and to attempt to relate them to overall running time. For example, assume that the system detects that page faults consume 50% of the overall running time during execution of an eight-by-two configuration. A good heuristic might be able to combine this information with the correlation map shown in Figure 4 and deduce that the four-by-four configuration is worth trying.

5.4 Thread-mapping heuristics

Given correlation maps and the thread capacity of each node, we can now attempt to map threads to nodes in a way that minimizes communication. This problem is related to the classic bin-packing and weighted-cut problems. However, our problem is essentially the packing of a number of differently sized bins, such that the weight of the items being packed depends upon which bin is being considered. One final complication is that we also want to minimize the number of threads that have to be migrated.

We tested both leader-based and leader-less versions of each of the heuristics discussed below. Leader-based heuristics attempt to minimize thread migrations by constraining the "leader" of each node to remain on the same node in the new configuration. Leaders are those threads with the lowest total communication requirements. If the current configuration is good, the leaders will tend to prevent other threads from migrating as well. Leader-based algorithms work best in cases where highly correlated threads are already co-located on the same node.

A mapping of a thread to a node is not considered if the addition of the new thread would overwhelm the node's thread capacity. Unfortunately, non-uniform capacities or requirements might cause fragmentation of available capacity, preventing all threads from being assigned under the above policy. We assign any leftover threads to those nodes that have the greatest remaining capacity, regardless of sharing behavior.

AscEdge

Our first heuristic, AscEdge, uses the standard approach of "weighted cut" heuristics in attempting to ensure that non-

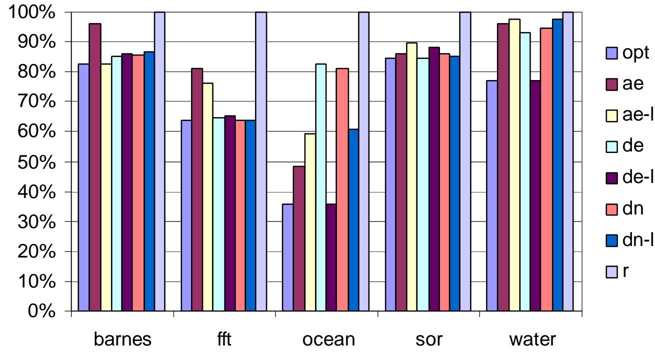

Figure 5: Normalized 8-processor execution times

communicating threads are not co-located. AscEdge treats threads and the sharing between them as nodes and edges of a weighted graph, respectively. We map threads to processors by sorting edges according to weight (correlation between the threads) in ascending order, breaking ties by choosing on the edge with the lowest-numbered node (thread). The endpoints of each edge are put onto distinct nodes, if possible. Each thread is put onto the node with which the thread has the highest aggregate correlation (through the threads currently on the node). Nodes are preferred in numerical order in the case of ties. One potential problem is that even the highest-cost edges, which are processed last, might be placed on distinct nodes, causing large amounts of communication.

DesEdge

DesEdge is similar to AscEdge, except that the edges are processed in the reverse order, and threads are placed into the same processor, if possible. This variation handles the edges with high communication costs explicitly, rather than implicitly as with AscEdge.

DesNode

DesNode, our final heuristic, works directly with threads. Threads are sorted by aggregate communication requirements. In terms of the above graph, the *weight* of a thread (a node of the graph) is the cost of communication across a cut that separates the thread from all other threads. Threads are sorted in descending order, and are mapped to nodes with which they have the highest aggregate correlation.

5.5 Heuristic performance

Our evaluation consists of two parts. First, we evaluate the accuracy of our heuristics by comparing the cut costs of the configurations generated by each heuristic with that of the optimal configuration. Second, we study the value of the cut cost as a predictor of communication requirements and overall performance. None of the untuned heuristics took longer than 1.5 milliseconds for any of our applications.

Table 4 shows the cut costs and communication that result from running each of the heuristics. The first five columns give cut costs. The second set of five columns gives the amount of data communicated per iteration, and the last five columns give the number of messages sent each iteration. The heuristics AscEdge, DesEdge, and DesNode are abbreviated 'ae', 'de', and 'dn', respectively. Leader-based variants are identified by '-l' suffixes. Additionally, we also show the communication costs of the optimal configuration ('opt'), and of a random configuration ('r').

Although there is a large amount of variation across the different applications and heuristics, the configurations generated by de-l are optimal for all but barnes, where the de-l configuration has a cut cost less than 1% higher than optimal. De-l minimizes the effects of fragmentation by handling the costliest edges first. Additionally, the leader-based approaches help to ensure that nodes are filled at the same pace. The problem with filling nodes at different paces is that highly-correlated threads might not both fit on the same node.

Cut costs match up quite well with the amount of data and the number of messages sent. However, the differences in cut costs are exaggerated in the byte and message totals. This implies that the pages handled better by some of the heuristics cause relatively more communication than pages handled equally well by all of the heuristics.

Figure 5 shows the execution times resulting from the use of each of the heuristics, normalized to the execution time of the random heuristic. Overall performance matches up well with the communication requirements shown in Table 4. Original speedups are 5.0, 4.5, 1.5, 7.1, and 5.0 for Barnes, FFT, Ocean, SOR, and water, respectively. The "default" mapping of threads to nodes closely approximates the optimal performance in this environment. However, the performance of default mappings in heterogeneous or non-dedicated environments could easily be closer to the performance of the random heuristic.

5.6 Synchronization behavior

So far, we have discussed only data sharing. However, threads also communicate in order to synchronize. Hence, a general-purpose cost function would seem to require taking both into account when assessing the viability of a candidate configuration. We have found, however, that the specific mapping of threads to nodes tends to affect synchronization behavior less than sharing behavior. While rigorously characterizing application synchronization behavior is beyond the scope of this paper, we can unscientifically divide synchronization operations into three different categories. First, many operations use global barrier synchronization. The cost of this synchronization is often significant to the application's overall performance. However, barrier cost is largely independent of any particular thread configuration, provided that load is balance. The reason is that all threads must participate, regardless of their location.

Many pair-wise lock synchronizations can be categorized as either reductions or work queue operations. In the former case, locks are used to arbitrate access to global sums or bounds. The need for such access is either uniform, i.e., all threads need to contribute to a given sum, or entirely unpredictable, as when locks are used to guard access to a global minimum. Moreover, the particular order that threads gain access to these synchronization variables is often non-deterministic. Hence, past access behavior is unlikely to be a good predictor of future accesses. Finally, work queue synchronization behaves similarly to reduction operations in that all threads access the work queue at least occasionally, and the relative access order is highly dynamic.

As the above discussion shows, synchronization behavior is not a viable candidate for use in determining thread configurations. While there is certainly a large class of applications for

	Cut Cost					Total KBytes Communicated					Data Request Messages				
	barnes	FFT	ocean	sor	water	barnes	FFT	ocean	SOR	water	barnes	FFT	ocean	SOR	water
opt	99458	2240	9792	28	11230	21122	17096	49876	867	8165	15011	3583	25727	196	2441
ae	102121	2960	11041	32	11902	26885	29703	82810	982	13899	24649	6450	34965	224	4199
ae-l	99458	2960	10918	36	11826	21122	31748	77736	1098	13467	15011	6737	33557	252	4064
de	100918	2240	12751	28	11651	23450	17096	144028	867	11683	16873	3583	51818	196	3547
de-l	100416	2240	9792	28	11230	23347	19021	49876	982	8176	17494	3812	25727	224	2442
dn	101276	2240	13592	32	11985	24726	17190	162050	982	14555	18182	3583	61330	224	4106
dn-l	100416	2240	11304	28	11835	23348	17191	103845	867	13559	17502	3587	41049	196	4097
r	103737	3440	14681	108	12091	27632	51428	121948	2943	14979	22022	10974	40689	700	4158

Table 4: Impact of thread mapping on communication statistics

which synchronization is more important than data sharing, the performance of such applications is better addressed through latency-hiding techniques.

6. Related Work

Thread migration has also been studied in the Millipede [22] and PARSEC [18] DSMs. Both systems implement thread migration in the context of sequential consistency rather than a relaxed consistency model. This makes comparisons with our system difficult, as sequentially-consistent systems suffer from both false and true sharing. Relaxed consistency models hide false sharing effectively without recourse to multi-threading [7]. Thread-scheduling algorithms on modern systems, therefore, only address performance problems due to true sharing.

Both systems implement forms of passive correlation scheduling, in which remote page faults are used to gain information about data sharing between threads. As discussed in Section 5.1, this technique fails to provide information about the affinity between local threads, and can cause thread thrashing.

In addition to correlation scheduling, PARSEC also implements a "suspension scheduling" algorithm that temporarily suspends threads involved in page thrashing. Suspension scheduling effectively deals with the same performance problems as the delta mechanism, which is only needed in single-writer protocols. Hence, suspension scheduling is unlikely to be of use with more modern underlying consistency mechanism. This is crucial in evaluating the performance results in this paper, as two of the three applications speed up only through suspension scheduling. The performance of the remaining application, water-nsquared from SPLASH-2 [23], improves by approximately 17%. However, the paper gives no absolute performance information for this application, and in fact does not specify how many processors are used.

MOSIX [24] is a distributed operating system that automatically migrates processes for load balancing and for the avoidance of virtual memory thrashing. The system does not support DSM and does not take sharing or communication into account when determining migration targets. However, the system ensures that a process's entire environment migrates transparently with the user. A data structure called a *deputy* is left on the migrating process's old node in order to provide forwarding addresses for interaction with the old node's environment.

Load balancing can also be accomplished implicitly through compile-time data placement [25]. Such techniques have the advantage of not incurring any runtime overhead at all. However, they are generally applicable for a smaller set of applications than runtime techniques. Furthermore, they assume dedicated, homogeneous environments. By contrast, runtime techniques can adapt to changing environments and application sharing patterns.

Several studies [21, 26] have detailed ways in which the runtime system can empirically determine the number of processors that allows the best speedup for a given application. A common characteristic of these systems is that the runtime system systematically tries different numbers of processors and uses a hill-climbing approach to converge to a local maximum. This approach has two drawbacks. First, the local maximum might not be a global maximum. Second, the search procedure can be very expensive, as the cost of migrating entire Unix processes across nodes is non-trivial. Our approach resembles a very slow hill-climbing algorithm (which assumes a good initial guess from the user), but can migrate work at a fine granularity.

7. Conclusions

This paper makes three contributions. First, we describe the design and implementation of the active correlation-tracking mechanism. Active correlation tracking captures complete sharing behavior *without* network communication or thread thrashing. We use this information to create correlation maps, which summarize sharing information among all threads in the system.

Second, we define the *correlation* of a pair of threads as the number of pages shared by the threads. The *cut cost* of a thread mapping is the sum total of the correlations of all thread pairs that are split across two nodes. We show that a mapping's cut cost works quite well as a predictor of communication requirements and overall performance. This is quite important, as the more intuitive notion of rate is either difficult to capture, and easily distorted by implementation details. The utility of the cut cost metric is not specific to our system. However, it is likely to be less valid for protocols and programming models that do not tolerate false sharing well.

We also evaluate several heuristics for finding thread mappings with low cut costs. The best for our applications was the leader-based descending-edge heuristic, which produced cut costs that averaged only 0.3% higher than optimal. The heuristic works by assigning the heaviest edges first, and by using fixed leaders to ensure that nodes are filled evenly.

Of course, this study only scratches the surface of mapping algorithms. We studied only a single system configuration, 32 threads distributed across eight nodes, and only a single set of thread capacities, all the same. This study could be extended to look at the ability of the heuristics to cope with differing thread capacities, and tradeoffs between communication costs and the number of thread migrations. Additionally, local decision-making is likely to be better suited to highly dynamic environments than is global decision-making.

Finally, the holy grail would be to integrate communication minimization, load imbalance minimization, and parallelism maximization into a single heuristic. The best approach to this problem is likely to combine aggressive monitoring of local effi-ciencies with sophisticated heuristics. We are continuing to work on this problem.

8. References

[1] P. Keleher, "The Relative Importance of Concurrent Writers and Weak Consistency Models," in *Proceedings of the 16th International Conference on Distributed Computing Systems*, 1996.

[2] F. Douglis and J. Ousterhout, "Process Migration in the Sprite Operating System," in *Proceedings of the 7th International Conference on Distributed Computing Systems*, September 1987.

[3] M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations," in *International Conference on Distributed Computing Systems*, 1988.

[4] P. Keleher, "CVM Manual," University of Maryland CS-TR-3545, October 1995.

[5] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[7] C. Amza, A. L. Cox, K. Rajamani, and W. Zwaenepoel, "Tradeoffs between False Sharing and Aggregation in Software Distributed Shared Memory," in *Proceedings of the Principles and Practice of Parallel Programming*, 1997.

[8] R. Rajamony and A. L. Cox, "Performance Debugging Shared Memory Parallel Programs Using Run-Time Dependency Analysis," in *Proceedings of the Sigmetrics'97 Conference on the Measurement and Modeling of Computer Systems*, June 1997.

[9] D. Keppel, "Tools and Techniques for Building Fast Portable Threads Packages," University of Washington Department of Computer Science and Engineering UWCSE 93-05-06, May, 1993 1993.

[10] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos, "First-Class User-Level Threads," in *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[11] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch, "The Sprite Network Operating System," *IEEE Computer*, vol. 21, pp. 23--36, February 1988.

[12] B. Steensgaard and E. Jul, "Object and Native Code Thread Mobility Among Heterogeneous Computers," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[13] E. Mascarenhas and V. Rego, "Ariadne: Architecture of a Portable Threads System Supporting Thread Migration," *Software - Practice and Experience*, vol. 26, pp. 327-356, March 1996.

[14] L. Prylli and B. Tourancheau, "BIP: a new protocol designed for high performance networking on myrinet," in *Workshop PC-NOW, IPPS/SPDP98*, 1998.

[15] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su, "Myrinet: A Gigabit-per-second Local Area Network," *IEEE Micro*, vol. 15, pp. 29-36, 1995.

[16] E. Dahlhaus, D. S. Johnson, C. H. Papdimitriou, P. D. Seymour, and M. Yannakakis, "The Complexity of Multiterminal Cuts," *SIAM Journal on Computing*, vol. 23, pp. 864-894, 1994.

[17] F. Berman and R. Wolski, "Scheduling from the Perspective of the Application," in *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, 1996.

[18] Y. Sudo, S. Suzuki, and S. Shibayama, "Distributed-Thread Scheduling Methods for Reducing Page-Thrashing," in *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing*, 1997.

[19] D. Scales and K. Gharachorloo, "Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory," in *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[20] K. Thitikamol and P. Keleher, "Multi-Threading and Remote Latency in Software DSMs," in *The 17th International Conference on Distributed Computing Systems*, May 1997.

[21] T. Nguyen, R. Vaswani, and J. Zahorjan, "Maximizing Speedup through Self-Tuning of Processor Allocation," in *Proceedings of 10th International Parallel Processing Symposium*, April 1996.

[22] A. Itzkovitz, A. Schuster, and L. Wolfovich, "Thread Migration and its Applications in Distributed Shared Memory Systems," Technion IIT LPCR #9603, July 1996.

[23] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[24] A. B. a. O. La'adan, "The MOSIX Multicomputer Operating System for High Performance Cluster Computing," *Journal of Future Generation Computer Systems*, April, 1998.

[25] J. Anderson and M. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines," in *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, June 1993.

[26] K. K. Yue and D. J. Lilja, "An Effective Processor Allocation Strategy for Multiprogrammed Shared-Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, December 1997.