Migration in Single Chip Multiprocessors

Kelly A. Shaw and William J. Dally
Computer Systems Laboratory, Stanford University
{kashaw, billd}@cva.stanford.edu

Abstract— **Global communication costs in future single-chip multiprocessors will increase linearly with distance. In this paper, we revisit the issues of locality and load balance in order to take advantage of these new costs. We present a technique which simultaneously migrates data and threads based on vectors specifying locality and resource usage. This technique improves performance on applications with distinguishable locality and imbalanced resource usage. 64% of the ideal reduction in execution time was achieved on an application with these traits while no improvement was obtained on a balanced application with little locality.**

I. Introduction

FUTURE technology will produce chips with billions of transistors, enabling large quantities of logic and memory to be placed on a single chip. Growing wire delays, however, make relative communication costs between these resources increasingly expensive. Single chip multiprocessors adapt to this high communication cost by dividing a chip into a grid of independently functioning cells connected by a global network. In the common case (such as a local computation with local operands), the wire length (i.e. communication distance) is the size of the node. For the uncommon case (such as a remote memory reference), communication travels through the global network. The challenge for these chips is to further reduce global communication by decreasing communication distance while still distributing work across the chip.

The issues of locality and load balance on multi-chip multiprocessors have been thoroughly examined [2][4][6]. Single chip multiprocessors, however, differ from these previous systems, prompting a re-examination of these issues. First, the large numbers of processors on a single chip supply an abundance of computational power. Second, the memory per node is small compared to previous per-node memory hierarchies, increasing remote data accesses. Third, and most importantly, the communication cost function differs drastically.

In previous multiprocessor systems, communication costs divided into two types: local or remote. Remote communication costs far exceeded local communication costs, making differences in remote access latencies insignificant. In the network we consider, contention-free communication costs are small but increase linearly with distance. This new communication cost function provides flexibility in the placement of data and threads in relation to one another. Data and threads no longer need to be co-located in order to have low latency accesses; they can be on physically close nodes. Data and threads can therefore be moved to balance resource demands with only slight increases in communication latency. Previous techniques for distributing work and improving locality do not take advantage of this key characteristic.

The authors would like to thank Margaret Martonosi for the many discussions that helped improve this work. This work was supported in part by DARPA under ARPA order E253 and monitored by the U.S. Army under contract DABT63-96-C.0039. Additional support was provided by DARPA under contract MDA904-98-C-A933 and the MARCO Interconnect Focus Research Center. Manuscript submitted: 13 Sept. 2002. Manuscript accepted: 23 Oct. 2002. Final manuscript received: 4 Nov. 2002.

In this paper, we present a new migration technique for data and thread migration which takes advantage of the new communication cost function. The approach characterizes the two competing goals of (i) reducing global communication distance and (ii) distributing resource demands as vectors which are then combined to determine a migration direction. This technique obtains 64% of the ideal reduction in execution time for an application exhibiting resource demand imbalance and locality and achieves no improvement on an application that is balanced and exhibits little locality.

II. Related Work

Previous work in process and data migration has been based on two key assumptions. First, prior work assumes processor utilization affects performance more than usage of other resources, and, second, data locality is defined as local or remote.

Processor load balancing techniques in message passing systems migrate tasks from nodes with high processor load to neighboring nodes with lower processor load [2][3]. Migration decisions in these systems ignore the issue of data locality. Some migration strategies in shared memory multiprocessors, such as central ready queues [7], also focus solely on processor load. Others, however, incorporate the benefits of co-located data into scheduling decisions. These approaches explore the benefits of not migrating threads whose data is resident in the local cache [6][8] and/or local memory [1][4]. Research in data migration focuses on co-locating data with its accessing threads either by migrating or replicating pages in memory [5][9].

III. A Vector Model for Migration

The conflicting goals of improving locality and distributing resource demands can be characterized as vectors. An *attraction vector* is associated with every object (data or thread) and designates the dominant direction of the object's communication. Moving the object along the direction of the attraction vector reduces communication latency. *Repulsion vectors* are associated with specific physical regions of nodes. A repulsion vector specifies the direction objects should be pushed towards to reduce the load on the current node. Combining these two vectors creates a *migration vector* which specifies a direction to move an object along to best satisfy the two competing goals.

A. Where to Migrate?

The choice of migration destinations affects the amount of information collected to calculate attraction and repulsion vectors as well as the time to actually migrate. The simplest scheme allows only single-hop migrations between neighboring nodes. In a two-dimensional chip, objects have the opportunity to move to any of four immediate neighbors: north, south, east, and west.

B. When to Migrate?

Two parameters that strongly affect the benefits and costs of migration are (a) the time to make each migration decision and

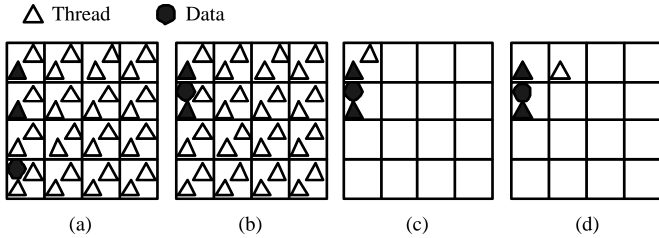

Fig. 1. a-d show the effects of data and thread migration

(b) how frequently resource usage information is exchanged. Longer decision times result in fewer migrations. In this study, we assume each migration decision takes 100 cycles. Frequent state collection prevents problems with stale information but increases network demands. We have chosen to exchange information every 10,000 cycles. Threads are reconsidered for migration after every exchange of state information. Nodes reconsider resident data in a round-robin fashion.

C. *How to Migrate?*

The attraction vector is calculated based on an object's communication patterns. We keep track of every communication in each direction. Opposing directions cancel out, allowing a single counter to be used per dimension. Because communications from and to the same row or column as the object's location discourage migration away from the current location, row and column statistics are collected and then used to reduce the attraction vector's magnitudes.

The repulsion vector is calculated for each region of nodes formed by a node and its four neighbors. Neighbors exchange load information for memory, communication, and processor resources to calculate repulsion vectors. Resources exert pressure away from themselves only if they exceed some usage threshold. For memory and communication, the threshold is 90% of capacity. A processor is considered overloaded if it contains more threads than the region's average. The complete repulsion vector is calculated by adding together the repulsion vectors for each resource and normalizing the result.

Finally, the migration vector is created by combining the attraction and repulsion vectors. An object is migrated if the migration vector's magnitude exceeds the time to migrate. (We have assumed a fixed migration time of 10 cycles for data; future work will incorporate data object size.)

D. *Migration Example: Raytrace*

To illustrate the benefits of simultaneously migrating threads and data, we describe the execution of a raytracing application on a 16 node single chip multiprocessor. A raytracing application generates a picture by sending rays of light through a scene to determine the color of pixels in the final image. Rays intersect with scene geometry, determining pixel color. Calculations performed by each ray are independent from one another and vary in complexity depending on the geometry intersected.

To maximize parallelism, a single thread executes the calculations for a single ray. Figure 1(a) shows an initial configuration for a 4x8 image. Two threads are placed on each node. Because rays generated for neighboring pixels may access the same data, threads are placed based on the associated pixel's coordinates. Data is randomly distributed because associations between threads and data are not known a priori. For illustrative purposes, one piece of data and two threads that access that data are colored black.

In Figure 1(a), moving threads would result in processor load imbalance. Communication latency, however, can be reduced by using locality to migrate data. Migrating the black data north as shown in Figure 1(b) decreases the communication latency for each of the two black threads' accesses; both threads benefit even though the data is co-located with only one thread.

Figure 1(c) shows the application as threads complete, leaving some nodes idle. Thread migration based on resource usage would move one of the remaining threads to a neighboring idle node. Blindly migrating the black thread would cause every black data access to incur an additional hop latency for each direction traversed. By incorporating communication patterns into the migration decision instead, the black thread remains at its current location and the white thread is migrated as shown in Figure 1(d).

As the example shows, the ability to migrate both data and threads supports

- migrating data based on locality,
- migrating threads based on resource usage, and
- migrating threads to reduce communication costs.

Although not shown in this example, data migration can also reduce contention for memory and communication resources. For example, data can be migrated away from nodes with high communication demands. Performing both thread and data migration can therefore achieve better performance than either form of migration alone.

IV. System Overview

A. *Chip Architecture*

In 2005, a single chip will hold 64 simple processors, 64 MB of DRAM, and a global network. We envision a baseline architecture where resources are organized into an 8x8 grid of nodes, each comprised of a 64-bit single-issue, in-order processor, 1 MB of DRAM, and a network interface to a mesh network.

In our simulations, processors execute one instruction per cycle, contain a single hardware thread context, and issue non-blocking memory requests. The combined chip memory acts as a shared address space accessed through the mesh network. Only application-defined, global, read-only data may exist on multiple nodes at any given time. The memory organization acts similar to a cache where each 64B line has an associated tag and state bits. While accesses to memory are modeled, directories and their associated communication are not simulated.

The global network has a linearly increasing cost function with respect to distance. In the absence of contention, communication latency is measured in terms of the number of routers traversed, where the time to traverse a router is equal to a processor cycle. Physical channels are 8B wide (1 flit) and are multiplexed between four virtual channels. Routers route up to five flits per cycle and can buffer four flits per virtual channel. Off-chip communication is assumed to take fifty cycles regardless of where the request originates.

B. *Applications*

Table I shows the computation and communication demands of the two applications studied, excluding beginning and ending serial code. The first application, raytrace, was introduced in Section III-D; it generates a 128x128 scene from the game Quake.

The second application, Barnes-Hut, performs an n-body simulation of galaxies over time. For each timestep, the effect of all other bodies on a given body's velocity and position is calculated. Distant groups of bodies can be treated as single entities, thereby reducing the total work. A single thread performs each body's computation. Bodies are distributed randomly and threads execute at the location of their associated bodies. 1024 bodies are simulated for one timestep.

TABLE I

APPLICATION CHARACTERISTICS FOR PARALLEL CODE SECTION

	raytrace	barneshut
Threads	16,384	2,048
Instr	136,912,117	194,774,175
Stack Ref	55,409,651	127,344,352
Non-Stack Ref	6,342,147	12,817,321
Accessed Objects	9,745	3,124

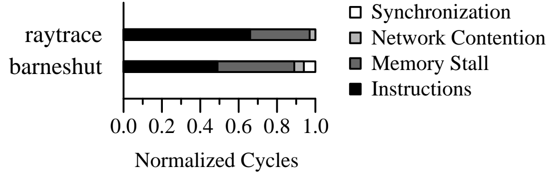

Fig. 2. Breakdown of baseline application performance

Figure 2 shows the breakdown of cumulative execution time. Memory stall time represents the improvements possible from either hiding or reducing communication distance. The network contention delays depict possible benefits from reorganizing the placement of threads and data to avoid network hotspots.

V. DATA MIGRATION

Figure 2 suggests that both applications can benefit from reducing communication latencies. Data migration accomplishes this by reducing communication distance and network contention. It also redistributes memory demands, but that has little effect on the applications studied here due to their low memory demands. Because these applications experience little network contention with their initial placement, we focus on the benefits accrued from reducing communication distance.

Figure 3 shows the execution time benefits from migrating data. For comparison, an idealized execution time that assumes all accesses are local, called all-local, is also shown. Data migration based on locality reduces raytrace's execution time by 26% compared to all-local's 42% reduction. Although all-local is able to halve the execution time for barneshut, data migration based on locality increases execution time by 8%.

Data migration improves raytrace's performance by reducing communication distance. Data is gradually moved towards accessing threads, decreasing average communication distance as shown in Figure 4(a). Figure 4(b) shows that data migration reduces barneshut's average communication distance slightly.

Data migration can only reduce communication distance if data has non-zero attraction vectors and if data can be moved

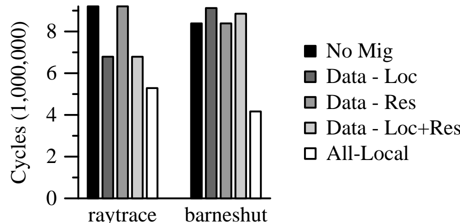

Fig. 3. Effects of data migration on overall execution time

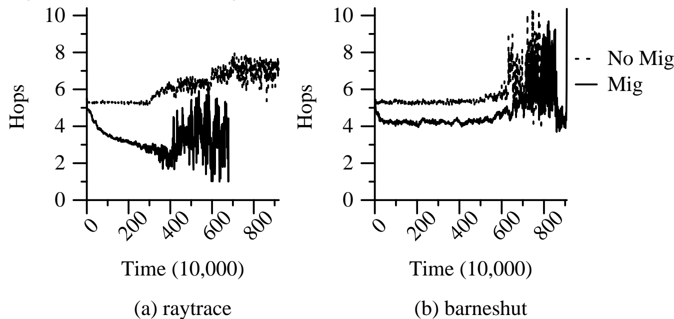

Fig. 4. Average communication distance with and without data migration

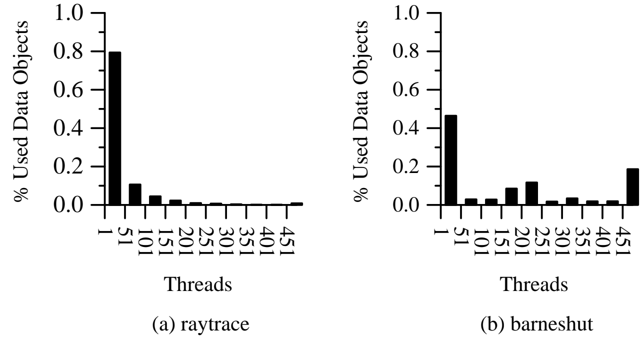

Fig. 5. Histogram of the number of threads accessing each data object.

near its accessing threads. Data accessed by small numbers of threads are more likely to have non-zero attraction vectors because thread accesses are less likely to cancel out. Figures 5(a) and 5(b) show that data objects in raytrace are accessed by small numbers of threads (~80% accessed by fewer than 50 threads) while those in barneshut are accessed by larger numbers of threads (~40% accessed by more than 200 threads). Because threads in raytrace are initially placed in order to benefit from locality between threads, data objects are even more likely to have non-zero attraction vectors.

The more threads that access a data object, the less likely that data object can be close to all of them. For example, each node in barneshut executes 32 threads, meaning the 200 threads that access a given object must be distributed across at least 7 nodes. Raytrace does not suffer as much from this limitation because 256 threads execute on each node.

Barneshut's average communication distance decreases slightly despite data being pulled in all directions because the network model is a mesh (versus a torus). Consequently, data near the chip's edges moves towards the center of the chip, creating network contention and increasing execution time. Incorporating resource usage into migration decisions reduces this network contention, however, a stronger repulsion force is needed at the chip's center to prevent all of the contention.

VI. THREAD MIGRATION

In order to isolate the benefits of thread migration, we hide remote communication latency by allowing multithreading (eight threads) on each processor. This large amount of multithreading also allows thread migration to impact performance on the applications studied. Figure 6 shows the effect of thread migration in its various forms on execution time, including another idealized metric called perfect. The perfect metric assumes all-local memory accesses and perfect processor load balance, where every cycle one instruction is executed for up to 64 threads regardless of actual thread placement.

A. Thread Migration to Reduce Latency

As threads finish, multithreading is unable to hide all remote communication latencies. Thread migration based on locality should therefore still improve performance by reducing latency. As Figure 6 shows, however, this strategy hurts performance, resulting in barneshut running six times slower and raytrace almost twice as slow. Certain nodes become overworked as threads migrate to improve locality. Therefore, the benefits from improved locality are smaller than those from maintaining a balanced distribution of threads.

B. Thread Migration to Improve Resource Utilization

The perfect metric suggests that execution times for raytrace and barneshut can be reduced by 54% and 35% respectively over baseline execution times, although only 10% of barneshut's improvement is due to balancing resource demands.

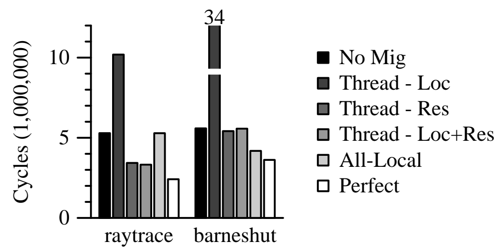

Fig. 6. Effects of thread migration on overall execution time

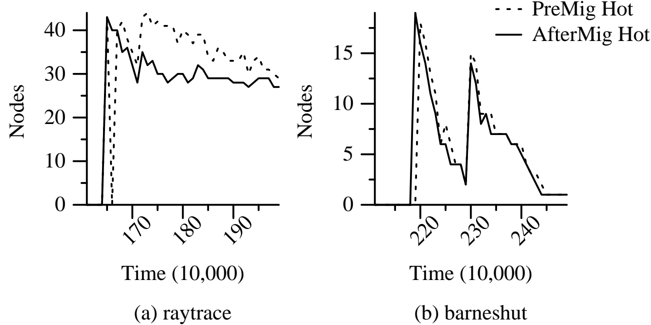

Fig. 7. Number of hot nodes before and after migration

As shown in Figure 6, thread migration based on resource utilization reduces execution time by 35% compared to the ideal 54% for raytrace but is only able to improve barneshut by 3%.

In order to improve performance with thread migration, resource usage must be imbalanced over an interval during which threads can be migrated. Raytrace becomes imbalanced after 20% of its execution, providing a long period during which thread migration can improve performance. In contrast, barneshut is imbalanced for only 33% of its execution time which limits thread migration's benefits. During these intervals of imbalance, thread migration affects performance for both applications by improving processor utilization, although the benefits for barneshut are small. This higher utilization is achieved by moving threads off of overutilized nodes as shown in Figures 7(a) and 7(b). The graphs depict the number of hot nodes before and after thread migration, where a hot node is defined to be a node whose resource demands exceed the threshold used in migration decisions. For clarity, only 40,000 cycles of each application's imbalanced phase are shown. For both applications, thread migration reduces the number of hot nodes, moving work to less utilized nodes.

Raytrace's larger performance benefits also derive from its wider variance of imbalance in terms of thread count per node. For example, one node may be executing seven threads while its neighbor is idle. In contrast, nodes in barneshut tend to have fairly even demands; most nodes differ from the average by one thread. Barneshut's relatively balanced demands make it difficult to improve performance through thread migration.

C. Combining Locality and Resource Utilization Demands

Figure 6 also shows the benefits of using both locality and resource utilization in thread migration decisions. The negative effects of migrating threads based solely on locality are restrained by the resource utilization criteria. Barneshut sees no change in execution time when locality is added to resource utilization while raytrace receives a slight improvement.

VII. Combining Data and Thread Migration

Figure 8 shows the effects of combining both thread and data migration when both locality and resource utilization criteria are used. Barneshut is neither hindered nor helped by the combination of the two as it was not significantly helped by either independently. Raytrace, however, achieves a 35% reduction in execution time out of a 54% idealized reduction.

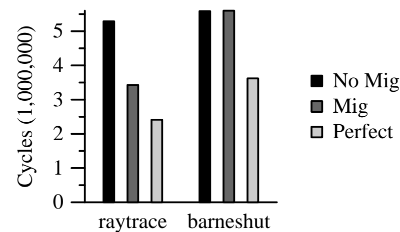

Fig. 8. Benefits of both thread and data migration in comparison to perfect resource balance and only local memory accesses

VIII. Conclusions

We introduced a vector approach for data and thread migration on single chip multiprocessors that takes advantage of these chips' new communication cost function. We examined the effects of thread and data migration separately and simultaneously. On raytrace, which has both distinguishable locality and resource utilization imbalance, the migration technique obtains 64% of the idealized reduction in execution time. Simultaneous thread and data migration should be able to improve performance on applications that, like raytrace, either have data that is used by a limited number of threads and/or have threads that perform varying amounts of work.

Although migration did not improve barneshut's performance, it did not hinder performance. One possible way of improving performance on applications like barneshut is to improve their locality by restructuring the application. For example, by dividing each long-running thread into shorter threads that each touch a small number of data objects and executing these smaller threads near the data they touch, it may be possible to obtain non-zero attraction vectors. These attraction vectors could then be used to reduce communication distances.

Study of this approach on applications with varying degrees of locality and resource utilization (in particular higher memory and communication demands) is needed for better insight. However, initial observations show this method improves performance in applications with locality and resource imbalance.

References

[1] R. Chandra, A. Gupta, and J. Hennessy, "Data Locality and Load Balancing in COOL," in *Proc. of the 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, San Diego, CA, pp. 249–259, May. 1993.

[2] F. C. H. Lin and R. M. Keller, "The Gradient Model Load Balancing Method," in *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, pp. 32-38, Jan. 1987.

[3] L. V. Kale, "Comparing the Performance of Two Dynamic Load Distribution Methods," in *Intl. Conf. on Parallel Processing*, University Park, PA, pp. 8-11, Aug. 1988.

[4] E. Markatos and T. LeBlanc, "Load Balancing vs. Locality Management in Shared-Memory Multiprocessors," in *Intl. Conference on Parallel Processing*, St. Charles, Illinois, pp. 258-265, Aug. 1992.

[5] V. Soundararajan, M. Heinrich, B. Verghese, K. Gharachorloo, A. Gupta, and J. Hennessy, "Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors," in *Proc. of the 25th Intl. Symp. on Computer Architecture*, Barcelona, Spain, pp. 342–355, June. 1998.

[6] M. Squillante and E. Lazowska, "Using Processor-Cache Affinity Information in Shared Memory Multiprocessor Scheduling," Tech. Rep. FR-35, University of Washington Computer Science Department, Feb. 1990.

[7] A. Tucker and A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed, Shared Memory Multiprocessors," in *Proc. of the 12th Symp. on Operating Systems Principles*, Litchfield Park, AZ, pp. 159-166, Dec. 1989.

[8] R. Vaswani and J. Zahorjan, "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors," in *Proc. of ACM Symp. on Operating Systems Principles*, Pacific Grove, CA, pp. 27-40, Oct. 1991.

[9] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating System Support for Improving Data Locality on CC-NUMA Compute Servers," in *Proc. of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, pp. 279–289, Oct. 1996.