Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems

Wilson C. Hsieh, Paul Wang, William E. Weihl

Abstract

We describe *computation migration*, a new technique that is based on compile-time program transformations, for accessing remote data in a distributed-memory parallel system. In contrast with RPC-style access, where the access is performed remotely, and with data migration, where the data is moved so that it is local, computation migration moves part of the current thread to the processor where the data resides. The access is performed at the remote processor, and the migrated thread portion continues to run on that same processor; this makes subsequent accesses in the thread portion local.

We describe an implementation of computation migration that consists of two parts: an implementation that migrates single activation frames, and a high-level language annotation that allows a programmer to express when migration is desired. We performed experiments using two applications; these experiments demonstrate that computation migration is a valuable alternative to RPC and data migration.

1 Introduction

One of the most important factors for efficient program execution on a distributed-memory parallel system is the locality of data accesses. If there are many non-local memory accesses in a parallel program, it is unlikely that the program will exhibit good speedup. There are two important factors in ensuring locality of accesses: the placement of data, and how remote accesses are handled. We deal with the second issue — where and how remote accesses should occur — and propose a new mechanism for remote access called *computation migration*. For some applications, computation migration can significantly reduce the communication cost associated with accessing remote data compared to other existing techniques. The approach we describe in this paper allows the programmer to write programs in a shared-memory style; the compiler (perhaps with guidance from the programmer) then decides how to map the program onto a particular machine, and uses computation migration and other techniques to optimize communication.

Several mechanisms for accessing remote data are widely used. RPC, or remote procedure call (e.g., [BN84, CAL+89, JLHB88, SB90]), is used in both distributed and parallel systems for executing procedure calls on remote data. Remote procedure calls resemble local procedure calls; compiler-generated stubs hide the underlying complexity of executing a procedure remotely. Another mechanism for remote access is data migration or data-shipping (e.g., [CKA91, JLHB88, LLG+90, Li88]). Using data migration (an example of which is cache-coherent shared memory), the remote data is moved (or copied) so that it is local to the access. Yet another mechanism for remote access is thread migration (e.g., [PM83, SN90, TLC85]), which can be viewed as the converse of data migration. Using thread migration, a thread that makes a remote access is moved to the processor where the data resides, and it continues execution where the data is located.

Computation migration can be viewed as partial thread migration.[1] Migrating entire threads can be useful for load balancing, but is usually not an effective technique for handling accesses to remote data because the amount of state to be moved is large. However, if we generalize thread migration to allow only *part* of the thread to be moved (for example, the part corresponding to one or more activation frames at the top of the stack), the amount of state that must be moved may be relatively small, and it may be more effective to move this state than to use RPC or data migration to perform a remote access.

We do not propose that computation migration should always be used instead of RPC, data migration, or thread migration. Indeed, which approach is best depends on the characteristics of the application and of the architecture on which the application is being executed. We believe that programmers (or compilers) should be able to choose the option that is best for a specific application on a specific architecture. The evidence we present in this paper shows that computation migration can provide performance advantages over other techniques, and thus programmers should be able to choose it instead of RPC or data migration as the method used to perform a remote data access. In addition, we explain how we can use compiler transformations to generate programs that use computation migration; such structures do not have to be programmed explicitly.

We have designed and implemented a prototype of computation migration as part of the Prelude parallel programming language [WBC+91], which we have implemented on the Proteus simulator [BDCW91]. This implementation has two parts: the low-level implementation of computation migration, and high-level annotations in Prelude that allow a programmer to choose when to use computation migration.

We believe that supporting portability and tunability requires providing a comprehensive suite of mechanisms

Authors' address: MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139. Email: wchsieh@lcs.mit.edu. The first two authors were supported by NSF Graduate Fellowships. This research was also supported in part by the NSF under Grant CCR-8716884, by DARPA under Contract N00014-89-J-1988, by an equipment grant from DEC, and by grants from AT&T and IBM.

[1] By "thread", we mean threads in the sense of lightweight threads packages [CD88], as opposed to threads in TAM, which are much lighter-weight [SCvE91].

for performing remote accesses, including RPC, data migration, and computation migration. Our current prototype was designed for evaluating the effectiveness of computation migration relative to RPC and to data migration. It supports RPC, data migration, and migration of single activation frames; we are working on extending it to support migration of multiple and partial activations. We believe that this additional flexibility is necessary to allow a programmer to tune and port programs without being forced to restructure them by hand.

Section 2 explains in more detail how computation migration can reduce the cost of communication relative to other techniques, and presents a simple model of how computation migration saves messages. Section 3 describes an implementation of computation migration, and Section 4 describes experiments that we performed with our prototype. Section 5 discusses some related work. Finally, Section 6 summarizes our results and discusses directions for future work.

2 Motivation

We begin by discussing RPC, data migration, thread migration, and computation migration in more detail. We then compare RPC, data migration, and computation migration, and describe how computation migration saves messages over RPC and data migration.

2.1 RPC-style Access

The most common mechanism for implementing remote access in a message-passing system is remote procedure call, which has been used in both distributed and parallel systems [BN84]. RPCs resemble local procedure calls; the underlying message-passing is hidden by compiler-generated stubs. When an RPC occurs, a local call is made to a client stub, which marshals the arguments and sends a message to the processor where the data resides. At the remote processor, a server stub unmarshals the arguments and calls the procedure. When the remote procedure returns to the server stub, the server stub marshals any result values into a message and returns the message to the client stub; the client stub returns the results to the RPC caller.

A problem with RPC-style access is that each call requires two messages, one for the call and one for the reply; this can be expensive, as there is substantial overhead involved in marshaling values into and out of messages. The major problem with using RPC for a series of accesses to remote data, however, is that there is no locality of access; each access is remote.

One advantage of RPC is that it does not add "extra" load on a server; when a remote call finishes and returns, the thread that initiated the call does not consume any more resources on the server (until it makes another call). Another advantage of RPC is that it can work well in the case of write-shared data (shared data that is frequently written to), especially when compared to data migration.

2.2 Data Migration

Data migration involves moving remote data to the processor where a request is made. One example is hardware caching in shared-memory multiprocessors such as Alewife [CKA91] and DASH [LLG+90]; such systems provide automatic data replication as well as data migration. The Munin system [BCZ90] allows the programmer to choose among several different coherence protocols for shared data. The Emerald system [JLHB88] migrates entire objects without replicating them.

Data migration can perform poorly when the size of the data is large. In addition, data migration in the form of cache-coherent shared memory performs poorly for write-shared data because of the communication involved in maintaining consistency: when there are many writes to a replicated datum, it is expensive to ensure global consistency. Write-shared data appears to occur moderately frequently, resulting in data that migrates from one cache to another with relatively few copies existing at any one time [GW92]. Some researchers have tried to optimize their cache-coherence algorithms for migratory data [SBS92] to reduce the communication required to maintain consistency. Even so, RPC and computation migration will require fewer messages than data migration for write-shared data, and hence may result in better overall performance, particularly if message-passing is efficient (as in [vECGS92, HJ92]). (The performance results in Section 4 show that shared memory consumes significantly more network bandwidth than RPC or computation migration.)

One important advantage of data migration is that it can improve the locality of accesses: after the first access by a thread to some data, successive accesses will be local. In addition, cache-coherent shared memory increases concurrency for read-shared data (shared data that is rarely written) by replicating it in different caches that can be accessed concurrently.

2.3 Thread Migration

Thread migration can be used to improve load balance as well as locality, but its major problem is that the grain of migration is too coarse. Migrating an entire thread can be expensive, since there may be a large amount of state to move; it also may be unnecessary, since we often need to migrate only a small amount of state in a thread to improve locality substantially. In addition, migrating every thread that accesses a datum to the datum's processor could put too much load on that processor.

2.4 Computation Migration

Our proposed mechanism, computation migration, gives us the benefits of both thread migration and RPC. If we move part of a thread's stack to the remote data, we can receive the benefit of increased locality of accesses; for example, if the executing thread makes a series of accesses to the same data, there is a great deal to be gained by moving those accesses to the data. At the same time, we gain the benefit of RPC; we can vary the granularity of the computation to be performed at the remote processor. This allows us to avoid the problem of overloading the resources at a single processor, since we move only the amount of state necessary to improve locality.

The disadvantage of computation migration is that the cost of using it depends on the amount of computation state that must be moved. If the amount of state is large (such as a substantial portion of a thread) computation migration will be fairly expensive. In addition, when the amount of data that is accessed is small and rarely written, data migration (especially when done in hardware) should outperform computation migration.

Computation migration can also be viewed as the dual of data migration; instead of moving data to the computation, we move computation to the data. In the case where the data are not all on the same processor, some data migration may need to occur. In some cases it may be better to migrate the computation as well as some of

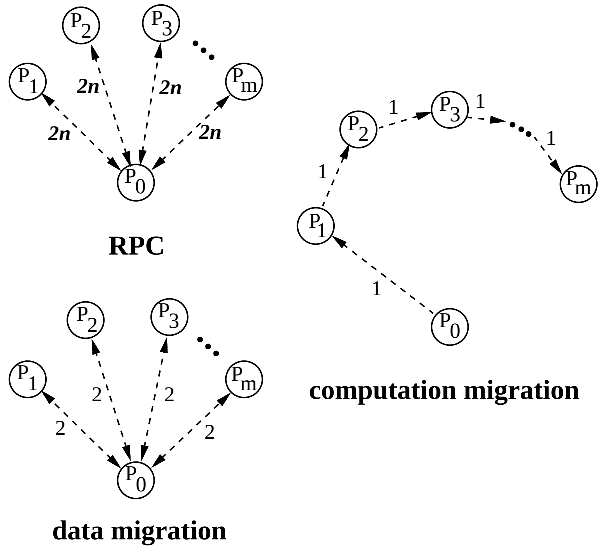


Figure 1: Message patterns under different mechanisms

the data; the pure data migration solution of always migrating all of the data to the computation may not always be the best.

Some applications require very little locality management: the computation of Fast Fourier Transform, in fact, requires data to be migrated exactly once during the entire computation; all accesses are local. However, we expect that for most applications (and dynamic time-shared systems), it will not be possible to layout the data once so as to achieve good locality.

Load balance is an issue that can conflict with locality management in parallel systems. For some regular numerical problems, it is sometimes possible to layout the data initially so as to provide both good locality and load balance under an "owner-computes" rule; languages such as Fortran D [HKK+91] and Vienna Fortran [ZBC+92] use such a rule. However, most applications (and certainly multiprogrammed systems) cannot be dealt with so simply. Markatos and LeBlanc have shown that for NUMA shared-memory multiprocessors, locality management can have a much larger impact on performance than load balancing [ML91]; even though computation migration can also be viewed as a mechanism to achieve load balance (since it is a form of thread migration), for most applications the locality aspects of computation migration are probably more important.

2.5 Comparison

In this section we compare the communication cost of computation migration to that of RPC and data migration by examining a scenario where a single thread makes a series of remote accesses to data on other processors. We measure cost in terms of the number of messages each mechanism requires; for the moment, we ignore other issues such as contention and message sizes.

Figure 1 illustrates the different message-passing patterns using RPC, data migration, and computation migration, where one thread on Processor P_0 makes n consecutive accesses to each of m data items on processors 1 through m, respectively. Relative to RPC, both computation migration and data migration can reduce the number of messages by making repeated accesses to the same data local.

Computation migration also saves messages by "short-circuiting" return messages. For each datum accessed, only one message needs to be sent; using data migration and RPC, at least two messages need to be sent per datum. (RPC sends two per access to each datum.) In addition, no communication is needed to maintain coherence of copies of data, which is necessary with cache-coherent shared memory when the data is written.

Reducing the total number of messages improves performance (as long as the messages do not get much larger) for two reasons. First, it can reduce the demands on network bandwidth. More important, it removes the various costs of extra message handling from the execution of the thread: marshaling and unmarshaling values, allocating packets, moving through the network, and dispatching to the code to execute.

It is important to remember that our model only counts the number of messages; there are other factors that can affect the relative costs of the three mechanisms. First of all, we do not take the size of data into account; if we must send a large amount of data, we will need to use multiple messages. More important, we do not take contention of any sort into account. Data contention, for example, will strongly affect whether data migration or computation migration is more efficient: if the data is write-shared between many threads, computation migration will almost always perform better than data migration; otherwise, data migration should perform better. Contention is likely to be a very important factor in determining the best mechanism to use, but we do not yet have a good model of its effects.

Our model also does not take into account techniques for hiding latency, such as prefetching and multithreading. Prefetching will lower the relative cost of performing data migration, since the delays involved with data migration can be overlapped with computation. We also do not model the effects of increased concurrency that results from replication in the case of cache-coherent shared memory.

3 Implementation

We have so far discussed computation migration abstractly, in terms of migrating portions of stacks; we now describe an implementation of computation migration that allows the programmer to migrate a single activation record on the top of the stack. There are two parts to our implementation: how migration is expressed by the programmer, and the low-level mechanics of migration. First we describe how the programmer can express the migration of an activation using a simple annotation, and we argue why such an annotation is much more desirable than coding computation migration explicitly into the structure of a program. Then we describe two mechanisms for migration: the first is the one we have implemented, and the second is a potentially more efficient one that is not possible on our prototype.

3.1 Annotations

Prelude [WBC+91] is an object-based language that provides procedures and instance methods, where instance methods always execute at the object on which they are invoked. Instance method calls are the remote accesses in our language, and we permit procedures (single activations) to migrate to objects on which they invoke instance methods. We do not permit instance method activations to migrate, as we currently require all instance method calls to run at the site of the invoked object.

We provide a simple program annotation that allows the programmer to migrate a procedure. The annotation indicates an instance method call within the procedure where computation migration should occur. Migration is conditional on the location of the computation: if the computation is already local to the datum accessed, it does not migrate.

There are several important characteristics of our approach. First, it is very simple to experiment with computation migration: changing where migration occurs simply involves moving the annotation, and the programmer can easily switch between using computation migration, RPC, and data migration. Second, the annotation affects only the performance of a program, not its semantics. Finally, computation migration imposes no additional cost on local accesses: if the data item being accessed is local, the cost of the access is not affected by the use of an annotation indicating that computation migration should occur.

Some languages permit a programmer to code a structure explicitly that mimics the effect of computation migration. For example, Actor-based languages [Man87] encourage a "continuation-passing" style for programs, in which each actor does a small amount of computation and then calls another actor to perform the rest of the computation. In effect, the computation migrates from actor to actor. By carefully designing the message-passing structure of the program, a programmer can optimize the communication cost.

For several reasons, we think it is an extremely bad idea to code computation migration explicitly in programs. The most important reason is that such programs are very difficult to tune and port. Performance decisions in such a program are closely intertwined with the description of the algorithm; changing the communication pattern may require substantial changes to the program, and may result in inadvertent errors being introduced into the computation. It is far better to separate the description of the computation from the decisions of how it is to be mapped onto a particular machine, and to let the compiler handle the details of the message-passing for computation migration.

In addition, our experience with both parallel and distributed systems indicates that programming in terms of explicit message-passing can lead to programs that are much more complex than ones written in terms of shared memory. Such an explicit programming style is also undesirable from a software engineering perspective, since it requires the modification of the interfaces of data objects to include additional procedures that represent the migrated computations.

3.2 Our Implementation

We implemented computation migration in the Prelude compiler and runtime system and ran experiments on the Proteus simulator [BDCW91]. Our current Prelude compiler generates C code, which does not give us the flexibility to code the implementation we describe in Section 3.3.

Our implementation generates a special "continuation" procedure to handle computation migration. The continuation procedure's body is the continuation of the migrating procedure at the point of migration; its arguments are the live variables at that point. Client and server stubs are generated for the special procedure to handle the message-passing required to invoke it. At a call to an instance method that is annotated for computation migration, we check if the invoked object is local; note that this check happens for every instance method call, and so is not an extra cost for computation migration. If it is local, execution continues within the normal code. If it is remote, the client stub for the continuation procedure is invoked; it sends a message to start the server stub on the remote processor.

If the procedure that is migrating is actually invoked from a server stub (i.e., it is at the base of its stack), we pass the linkage information for the procedure to the continuation procedure and destroy the original thread. This happens whenever a procedure is called remotely and migrates, or whenever a continuation procedure (the result of a previous migration) migrates again. Thus, if a procedure accesses a series of remote objects on different processors, it will migrate from one to the next, and in the end return directly to its caller. If the migrating procedure is not at the base of its stack, we leave the client stub to wait for the result from the continuation procedure and return it to the procedure's caller.

3.3 Alternate Implementation

The implementation we describe above suffers from several problems. First, it leads to a large expansion in the size of our generated code: for each occurrence of computation migration, we generate a continuation procedure and its two stubs. Second, it is somewhat inefficient, since the live variables at the point of migration need to be passed as arguments to the continuation client stub, and then marshaled into a message.

If we had control of the actual code generation for Prelude, we could avoid generating the extra continuation procedure. Instead, at the point of migration the live variables (which reside in the registers and on the stack) would be marshaled into a message, which would be sent to the destination processor. At the destination processor, an alternate entry point to the migrating procedure would be called, which would unmarshal the values (back into an identical activation record and into the registers), and would then jump back into the original procedure's code. The only tricky point would be if the migrating procedure was originally expected to return values on the stack; we must ensure that the code on the destination processor returns to a stub that returns a message to the original processor.

4 Experiments

We have performed experiments with two different applications, both of which are a form of distributed data structure traversal: a counting network and a distributed B-tree. We used the Proteus simulator [BDCW91] to run our experiments; we simulated a RISC machine similar to the Alewife machine, but without its multithreading capability [ACD+91]. In addition, our target machine, unlike Alewife, provides both private memory and shared memory; it uses the same cache coherence protocol that Alewife does [CKA91], and each processor has a 64K shared-memory cache with a line size of 16 bytes. For each application, we compared using RPC, cache-coherent shared memory, and computation migration for data accesses. We would like to compare our results to object migration, such as the mechanism in Emerald [JLHB88], but our group has not finished implementing object migration in Prelude yet.

It is important to remember that we are actually comparing a software implementation of RPC and computation migration to a hardware implementation of data migration, namely Alewife-style cache-coherent shared memory. Cache-coherent shared memory benefits from hardware support in three ways: it provides cheap message-passing, since data migration messages are implemented directly in hardware; it provides a global address space, without which a global object name space must be built in software; and it provides automatic replication.

We performed several experiments to estimate the effects of providing hardware support for RPC and computation migration. First, we estimated the effects of having hardware support for message-passing, as described by Henry and Joerg [HJ92]. In particular, we assumed that the network interface is mapped into ten additional registers in the register file, so that marshaling and unmarshaling would be cheaper; we did not take advantage of other modifications that Henry and Joerg describe. We also estimated the effects of having hardware support for global object identifier translation, as in the J-Machine [DCC+87].

The last benefit of cache-coherent shared memory, replication, is the biggest difference in performance between shared memory and computation migration, both of which outperformed RPC in our experiments. The results of our experiments show that computation migration with hardware support performs as well as cache-coherent shared memory in the counting network, and slightly worse on the B-tree. In our B-tree experiments, we implemented a software replication method for the root of the tree [WW90] to estimate the benefits of replication for computation migration; our results show that replication is the primary reason why shared memory performs better, and that it can also be used to improve performance when computation migration is used.

4.1 Counting Network

Our first application is a counting network [AHS91, HLN92]. A counting network is a distributed data structure that supports "shared counting"; multiple threads request values from a given range of values. For example, when a set of threads executes the iterations of a parallel loop, each iteration should be executed by exactly one thread. The simplest solution is to use a single counter protected by a lock to store the iteration count; a thread acquires the lock and updates the counter when it needs a loop index. This solution scales poorly due to contention; a counting network is a distributed data structure that trades latency under low-contention conditions for much higher scalability of throughput.

A counting network is built out of balancers; a balancer is a two-by-two switch that alternately routes requests between its two outputs. We performed our experiments on an eight-by-eight counting network, which is essentially a six-stage pipeline; each stage has four balancers in parallel. We laid out the counting network on twenty-four processors (one balancer per processor). We will not go into further detail on how counting networks function; please see the references for details.

We ran with eight, sixteen, thirty-two, forty-eight, and sixty-four threads making requests; each of the requesting threads was placed on a separate processor. We used two different think times, zero and ten thousand cycles.

Figure 2 shows the results for the counting network under high contention (zero think time) and lower contention (ten thousand cycle think time). Under high contention, computation migration with hardware support can perform better than shared memory. Every access to a balancer modifies it, so the balancers are essentially write-shared. Therefore, the performance benefit of replication due to shared memory is negligible: in our experiments we measured a twelve percent hit rate on data objects in shared memory.

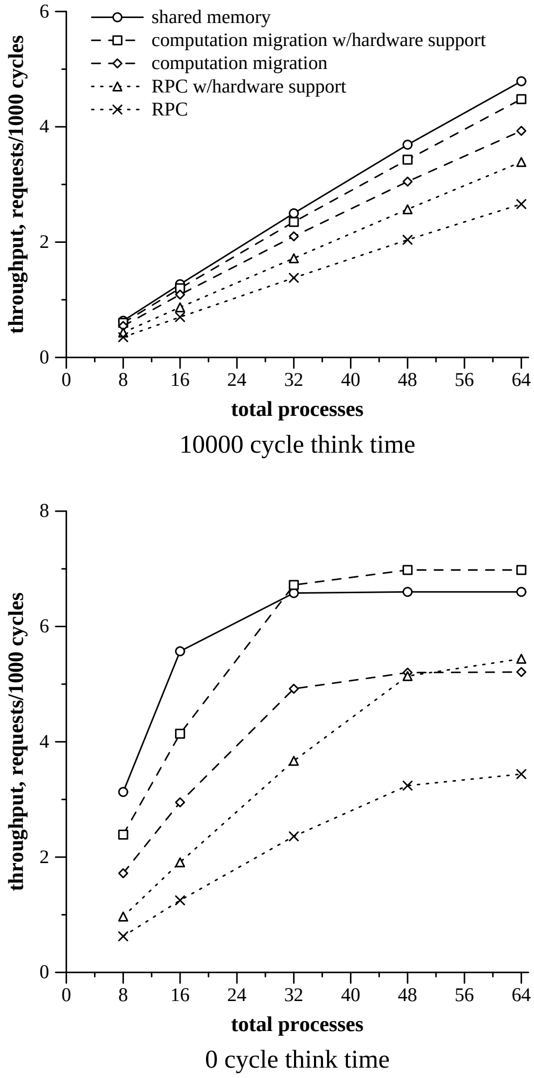
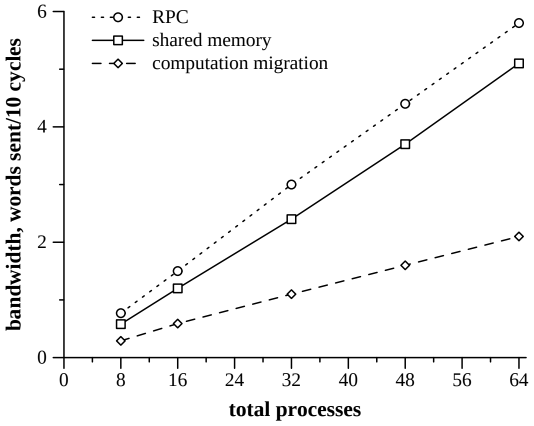
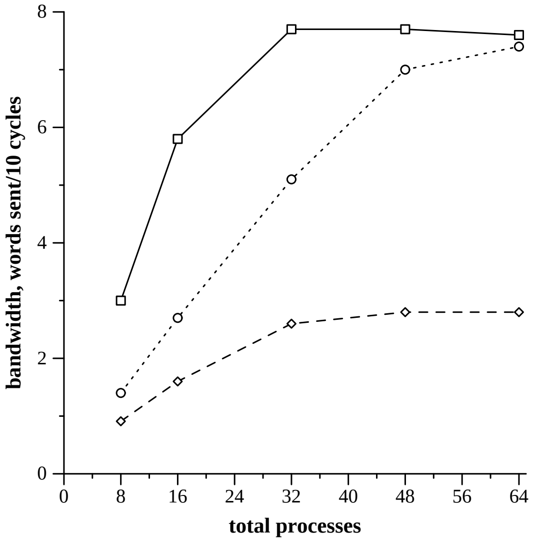


Figure 2: Throughput results

In fact, the automatic replication provided by shared memory consumes a great deal of bandwidth; even though shared memory often has the best overall performance in terms of request throughput, it puts a very high demand on the network. Figure 3 illustrates the bandwidth consumed by each mechanism. Computation migration requires less bandwidth than both RPC and shared memory. Shared memory requires the most bandwidth under high-contention conditions, due to the amount of coherence activity; at low contention, shared memory requires less bandwidth than RPC, but computation migration still requires less than half of the bandwidth of both RPC and shared memory.



10000 cycle think time



0 cycle think time

Figure 3: Bandwidth requirements

Under lower contention, we see that computation migration does not perform as well as shared memory. As we describe more precisely in Section 4.3, much of this is due to inefficiencies in our runtime system; if we were to tune the runtime, computation migration would probably perform as well as or better than shared memory.

4.2 B-tree

Our second application is a distributed B-tree. A B-tree [BM72] is a data structure designed to represent *dictionaries*; i.e., dynamic sets that support the operations *insert*, *delete*, and *lookup*. The basic structure and implementation of a B-tree is similar to that of a balanced binary tree. However, unlike a binary tree, the allowed maximum number of children for each B-tree node is not constrained to two; it can be much larger. Comer [Com79] presents a full discussion of the B-tree and its common variants.

Bayer and McCreight's original B-tree [BM72] was designed to support sequential applications. Since then, researchers have proposed many different algorithms to

Scheme	*Throughput (ops/1000 cycles)*
SM	1.837
RPC	0.3828
RPC w/HW	0.5133
RPC w/repl.	0.6060
RPC w/repl. & HW	0.7830
CP	0.8018
CP w/hardware	0.9570
CP w/repl.	1.155
CP w/repl. & HW	1.341

Table 1: Throughput results: 0 cycle think time

Scheme	*Bandwidth (words/10 cycles)*
SM	75
RPC	7.3
RPC w/HW	9.9
RPC w/repl.	7.0
RPC w/repl. & HW	9.3
CP	3.5
CP w/hardware	4.3
CP w/repl.	3.8
CP w/repl. & HW	3.9

Table 2: Bandwidth results: 0 cycle think time

support concurrent operations on the B-tree (e.g., [BS77, LY81, MR85, Sag86, LS86, CBDW91, Wan91, JC92]) and have conducted many studies examining their performance (e.g., [LS86, JS90, Wan91, Cos93]). Some algorithms [Wan91, CBDW91, JC92] are designed to run on a distributed-memory multiprocessor, where the B-tree nodes are laid out across several processors.

Two primary factors limit performance in distributed B-trees: *data* and *resource contention*. Data contention occurs when concurrent accesses to the same B-tree node must be synchronized. The most critical example of data contention is the *root bottleneck*, where an update to the tree's root node causes all incoming B-tree operations to block. Even if no synchronization among concurrent accesses is necessary, performance can still be degraded by resource contention. For example, since every B-tree operation accesses the root node, we would expect the memory module that contains it to be heavily utilized. If the rate of incoming operations exceeds the rate at which requests are serviced, then a bottleneck still exists.

Our distributed B-tree implementation is a simplified version of one of the algorithms proposed by Wang [Wan91] (it does not support the *delete* operation). In our experiments, we first constructed a B-tree with ten thousand keys and with the maximum number of children or keys in each node constrained to at most one hundred. The nodes of the tree were laid out randomly across forty-eight processors. We then created sixteen threads on separate processors that initiated requests into the B-tree; we ran experiments with two different think times, zero and ten thousand cycles.

Tables 1 and 2 present the results of the experiments using zero cycle think time (SM, CM, HW, and repl. represent shared memory, computation migration, hard-

Scheme	*Throughput (ops/1000 cycles)*
SM	1.071
CP w/repl.	0.9816
CP w/repl. & HW	1.053

Table 3: Throughput results: 10000 cycle think time

Scheme	*Bandwidth (words/10 cycles)*
SM	16
CP w/repl.	2.5
CP w/repl. & HW	2.7

Table 4: Bandwidth results: 10000 cycle think time

Category	*Cycles*	*Percent*
Total time	651	100%
User code	150	23%
Network transit	17	3%
Message overhead total	484	74%
Receiver total	341	52%
Copy packet (32 bytes)	76	10%
Thread creation	66	10%
Procedure linkage	66	10%
Unmarshaling	51	8%
Object ID translation	36	6%
Scheduler	36	6%
Forwarding check	23	4%
Allocate packet	16	2%
Sender total	143	22%
Procedure linkage	44	7%
Allocate packet	35	5%
Message send	23	4%
Marshaling	22	3%

Table 5: Approximate costs for migration in counting network

ware support, and software replication for the root of the B-tree, respectively). In all of the experiments, computation migration has higher throughput than RPC, but lower throughput than shared memory. In fact, even with hardware support for RPC and computation migration, the throughput of shared memory is over three times that of RPC and almost twice that of computation migration. As explained in Section 4.3, much of the relatively poor performance of computation migration and RPC can be attributed to the inefficiencies in the runtime system.

In addition, automatic replication contributes significantly to shared memory's throughput advantage. As stated above, one of the limiting factors to B-tree performance is the root bottleneck; it is the limiting factor for RPC and computation migration throughput. Computation migration, for instance, moves an activation for every request to the processor containing the root. The activations arrive at a rate greater than the rate at which the processor completes each activation. Shared memory, however, provides local copies of B-tree nodes to alleviate resource contention. In fact, the amount of work required to maintain cache coherence can be seen in Table 2; shared memory imposes a substantial load on the network. Furthermore, the cache hit rate was less than seven percent; we can conclude that shared memory benefited from its caches only because of its replication and not because of any locality.

We would expect computation migration's performance to improve considerably when software replication is provided for the root; as shown in Table 1, this is the case. However, the throughput of computation migration is still not as high as shared memory's. This is because we are still experiencing resource contention; instead of facing a root bottleneck, we experience bottlenecks at the level below the root — the root node has only three children. By changing the parameters of the experiments to alleviate this new source of resource contention, we get more favorable results.

In a separate experiment where the B-tree nodes are constrained to have at most only ten children or keys (but all other parameters are identical), the resulting throughput for continuation passing with root replication was 2.076 operations/1000 cycles vs. 2.427 operations/1000 cycles for shared memory. While shared memory still performs better, computation migration's throughput is better because the bottleneck below the root has been alleviated. The reason for this is twofold. First, the roots of the trees in these experiments have four (instead of three) children. Second, activations accessing smaller nodes require less time to service; this lowers the workload on the frequently accessed processors. From these experiments, we can conclude that the main reason for shared memory's performance advantage is the automatic replication due to hardware caches. This correlates with conclusions made by other studies examining B-tree performance and caching [Wan91, Cos93].

With the above information in mind, we should expect that the throughput of computation migration and shared memory to be much closer for a B-tree experiment where contention for the root is much lighter (thus reducing the advantage provided by caches for shared memory). Tables 3 and 4 show performance for experiments conducted with a ten thousand cycle think time; for brevity, RPC measurements have been omitted. With hardware support, computation migration and shared memory have almost identical throughput. Again, shared memory uses more bandwidth because it must maintain cache coherence.

4.3 Cost Breakdown

We measured the costs in RISC cycles for computation migration in Prelude; Table 5 shows an fairly accurate breakdown of the time to migrate one activation from one processor to another in the counting network application. As in most message-passing systems, the software overhead for sending a message dominates the execution time.

The table explains how our estimate of hardware support for message-passing improved our results by about twenty percent. When we assumed the presence of network interface registers, we assumed that we could reduce the copying overhead to approximately twelve cycles, removing about eight percent of the overhead. The registers also remove the need to allocate packets (since messages are composed in registers), and marshaling and unmarshaling costs are reduced by about half; this led to a removal of approximately another six percent of overhead. Finally, the assumption of hardware for global object identifier translation removed another six percent.

There are several obvious places where we could reduce the costs in Prelude. First, we spend ten percent of our time copying thirty bytes of data at the receiver, which is clearly inefficient. Eight percent of the time (fifty-one cycles) is spent unmarshaling data and translating global object identifiers on the receiving side; twenty-six of those cycles (eighteen for the translation and eight for linkage) are wasted because our implementation actually performs an unnecessary global object identifier translation during a computation migration.

We spend approximately another ten percent of our time creating a thread to handle the request and in copying the arguments for the thread (which were already copied once before). Currently, Prelude creates a new thread for most remote calls. If we were to move to an Active Messages-style implementation [vECGS92], we could avoid this overhead most of the time; for some of the short methods (such as remote record access) we already use such an approach, and it has proven to be quite effective.

As shown in the table, we also spend seventeen percent of the time in procedure linkage. If we were to rewrite portions of our runtime in assembly (or inline more of the procedures), we could eliminate almost all of this overhead. In addition, there is some additional overhead (which we did specifically measure) on the sending side due to our use of general-purpose stubs for all remote calls; we could increase performance by using special-purpose stubs for computation migration.

4.4 Summary

Our experiments show that computation migration and cache-coherent shared memory both outperform RPC. We should note, however, that we have optimized short methods in Prelude so that threads are not created for them; many of the extra calls performed using RPC call these faster short methods. If we were to perform similar optimizations for computation migration calls (as discussed in Section 4.3, computation migration would perform even better.

The factor that has the largest effect on computation migration is replication. In a counting network replication is not an issue, since all of the balancers are write-shared. Our experiments show that computation migration with hardware support for message-passing performs as well as cache-coherent shared memory at higher contention levels.

In a B-tree replication does become an issue. Even though cache hit ratios are still low in the B-tree, the lack of replication limits the overall throughput available using computation migration; we run into the problem that several activations all migrate to the same processor, thus causing resource contention. Shared memory, on the other hand, lets non-conflicting accesses run in parallel. When we add software replication to the B-tree, thus reducing resource contention, computation migration with hardware support exhibits the same performance as shared memory.

In terms of bandwidth consumption, computation migration puts a far smaller load on the network than shared memory. This would seem to indicate that as we push down message-passing costs, computation migration should perform better relative to shared memory. In fact, if we examine words transmitted per high-level operation (counting network or B-tree), computation migration still places a far smaller demand on the network. Finally, we showed that our current implementation of message-passing in Prelude is rather inefficient; given that we could remove quite a few wasted cycles from the fast path for message-passing, our results are pessimistic for both computation migration and RPC.

5 Related Work

Rogers, Hendren, and Reppy [RRH92] have developed mechanisms for migrating single activations that are similar to ours: on any access to remote data, they always migrate the top activation. The idea of migrating activations is similar to our work, but their approach is more restrictive than ours. Migrating an activation is often, but not always, the best approach. We let the user choose the appropriate mechanism; this flexibility can be vital for achieving good performance. In addition, fixing the granularity of migration to a single activation forces programmers to organize their programs so that procedures correspond to migration units; this will make it very difficult to tune or port programs. Our current prototype migrates only single activations, but an essential aspect of our overall approach is to provide the flexibility to migrate both larger and smaller units of computation.

Draves, *et al.* [DBRD91] suggest using a continuation-based structure to improve the performance of operating system services; they use continuations to reduce the amount of context-switching. They program continuations explicitly; the compilation techniques we use in this paper could probably be used to construct the continuations automatically, thus greatly reducing the programming effort involved in their approach.

Fowler and Kontothanassis [FK92] suggest using a similar "continuation-passing" structure to reduce context-switching costs in parallel programs. They also suggest using "object-affinity scheduling" to improve locality in parallel computations, where threads are scheduled based on information about what data they will reference in the near future. However, although they say that this information can be provided by the programmer or the compiler, they give no clear description of an implementation.

6 Conclusions

Ensuring high locality of data access is crucial to the performance of parallel systems. The choice of method for non-local accesses is critical: some methods of remote access can be far more expensive than others. We have described a new mechanism for remote access, computation migration, that for some applications can outperform traditional methods of access such as RPC and data migration.

Using a prototype we ran several experiments that demonstrate how computation migration outperforms RPC in applications that involve the traversal of distributed data structures, and how it can perform as well as cache-coherent shared memory. Computation migration also places a far smaller demand on the network than either shared memory or RPC. In non-shared memory systems, it would certainly be more efficient to use computation migration than data migration for the applications we have studied.

Our experiments showed that computation migration with hardware support performs as well as cache-coherent shared memory when replication is unnecessary. However, when replication is necessary, computation migration alone does not perform nearly as well as shared memory; this merely says that it is important to use replication with computation migration, just as when one uses data migration.

We intend to experiment with computation migration and replication more thoroughly to see if software replication and computation migration can perform as well as (or better than) shared memory. In addition, given the overheads of our message-passing implementation, we could certainly improve performance with further tuning. A rewrite of our runtime in an Active-Messages-style could lead to far better performance; the addition of hardware support, such as a register-based network interface, would also benefit computation migration.

We allow programmers to use computation migration with a simple program annotation; they do not need to code migration explicitly into the structure of their programs. Our program annotation affects only performance, not semantics, so the programmer can easily change the annotation to tune performance without affecting the correctness of a program; this should make tuning and porting parallel programs that use a migratory structure much easier.

Our current prototype allows the programmer to express single-activation migration. We are designing annotations to allow a programmer to express migration of multiple and partial activations. We believe this is essential for providing flexible control over the message-passing behavior of a program. We are also developing compiler analysis techniques for automatically choosing among the remote access mechanisms.

7 Acknowledgments

We would especially like to thank Anthony Joseph for helping us debug everything. Thanks to Anthony and Carl Waldspurger for helping us build Prelude; Eric Brewer for his help with PROTEUS; and Anthony, Carl, Eric, Ulana Legedza, Brad Spiers, Shail Aditya, Bradley Kuszmaul, Steve Keckler, John Keen, Sanjay Ghemawat, Donald Yeung, and the anonymous referees for giving us feedback on drafts of this paper.

References

[ACD+91] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. "The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor". In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. Extended version is MIT/LCS/TM-454.

[AHS91] J. Aspnes, M. Herlihy, and N. Shavit. "Counting Networks and Multiprocessor Coordination". In *Proceedings of the 23rd Annual Symposium on the Theory of Computing*, May 1991.

[BCZ90] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence". In *Proceedings of the 2nd PPoPP*, March 1990.

[BDCW91] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W.E. Weihl. "Proteus: A High-Performance Parallel Architecture Simulator". Technical Report MIT/LCS/TR-516, MIT LCS, September 1991. Shorter version appears in 1992 SIGMETRICS.

[BM72] R. Bayer and E.M. McCreight. "Organization and Maintenance of Large Ordered Indexes". *Acta Informatica*, 1(3):173–189, 1972.

[BN84] A.D. Birrell and B.J. Nelson. "Implementing Remote Procedure Calls". *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BS77] R. Bayer and M. Schkolnick. "Concurrency of Operations on B-trees". *Acta Informatica*, 9:1–22, 1977.

[CAL+89] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield. "The Amber System: Parallel Programming on a Network of Multiprocessors". In *Proceedings of the 12th SOSP*, pages 147–158, December 1989.

[CBDW91] A. Colbrook, E. Brewer, C. Dellorocas, and W. E. Weihl. "Algorithms for Search Trees on Message-Passing Architectures". In *Proceedings of 1991 ICPP*, pages III138–III141, 1991.

[CD88] E.C. Cooper and R.P. Draves. "C Threads". Technical Report CMU-CS-88-154, CMU Computer Science Dept. June 1988.

[CKA91] D. Chaiken, J. Kubiatowicz, and A. Agarwal. "LimitLESS Directories: A Scalable Cache Coherence Scheme". In *Proceedings of the 4th ASPLOS*, pages 224–234, April 1991.

[Com79] D. Comer. "The Ubiquitous B-Tree". *ACM Computing Surveys*, 11(2):121–128, June 1979.

[Cos93] P.R. Cosway. "Replication Control in Distributed B-Trees". Master's thesis, MIT, 1993. To appear in May.

[DBRD91] R.P. Draves, B.N. Bershad, R.F. Rashid, and R.W. Dean. "Using Continuations to Implement Thread Management and Communication in Operating Systems". In *Proceedings of the 13th SOSP*, pages 122–136, October 1991.

[DCC+87] W.J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills. "Architecture of a Message-Driven Processor". In *Proceedings of the 14th ISCA*, pages 189–196, June 1987.

[FK92] R.J. Fowler and L.I. Kontothanassis. "Improving Processor and Cache Locality in Fine-Grain Parallel Computations using Object-Affinity Scheduling and Continuation Passing (Revised)". Technical Report 411, Univ. of Rochester Computer Science Dept. June 1992.

[GW92] A. Gupta and W.D. Weber. "Cache Invalidation Patterns in Shared-Memory Multiprocessors". *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

[HJ92] D.S. Henry and C.F. Joerg. "A Tightly-Coupled Processor-Network Interface". In *Proceedings of the 5th ASPLOS*, pages 111–122, October 1992.

[HKK+91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. "An Overview of the Fortran D Programming System". In *Proceedings of the 4th Workshop on Languages and Compilers for Parallel Computing*, August 1991.

[HLN92] M. Herlihy, B.H. Lim, and N.Shavit. "Low Contention Load Balancing on Large-Scale Multiprocessors". In *Proceedings of the 4th SPAA*, June 1992.

[JC92] T. Johnson and A. Colbrook. "A Distributed Data-Balanced Dictionary Based on the B-Link Tree". In *Proceedings of the 6th IPPS*, pages 319–324, 1992.

[JLHB88] E. Jul, H. Levy, N. Hutchison, and A. Black. "Fine-Grained Mobility in the Emerald System". *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[JS90] T. Johnson and D. Shasha. "A Framework for the Performance Analysis of Concurrent B-tree Algorithms". In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, April 1990.

[Li88] K. Li. "IVY: A Shared Virtual Memory System for Parallel Computing". In *Proceedings of the 1988 ICPP*, volume II, pages 94–101, 1988.

[LLG+90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor". In *Proceedings of the 17th ISCA*, pages 148–158, May 1990.

[LS86] V. Lanin and D. Shasha. "A Symmetric Concurrent B-Tree Algorithm". In *1986 Proceedings Fall Joint Computer Conference*, pages 380–386, November 1986.

[LY81] P.L. Lehman and S.B. Yao. "Efficient Locking for Concurrent Operations on B-Trees". *ACM Transactions on Database Systems*, 6(4):650–670, December 1981.

[Man87] C.R. Manning. "ACORE: The Design of a Core Actor Language and its Compiler". Master's thesis, MIT, August 1987.

[ML91] E.P. Markatos and T.J. LeBlanc. "Load Balancing vs. Locality Management in Shared-Memory Multiprocessors". Technical Report 399, Univ. of Rochester Computer Science Dept. October 1991.

[MR85] Y. Mond and Y. Raz. "Concurrency Control in B+ Trees Using Preparatory Operations". In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 331–334, August 1985.

[PM83] M.L. Powell and B.P. Miller. "Process Migration in DEMOS/MP". In *Proceedings of the 9th SOSP*, pages 110–119, 1983.

[RRH92] A. Rogers, J.H. Reppy, and L.J. Hendren. "Supporting SPMD Execution for Dynamic Data Structures", 1992.

[Sag86] Y. Sagiv. "Concurrent Operations on B-Trees with Overtaking". *Journal of Computer and System Sciences*, 33(2):275–296, October 1986.

[SB90] M.D. Schroeder and M. Burrows. "Performance of Firefly RPC". *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[SBS92] P. Stenström, M. Brorsson, and L. Sandberg. "An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing", November 1992. Unpublished manuscript.

[SCvE91] K.E. Schauser, D.E. Culler, and T. von Eicken. "Compiled-Controlled Multithreading for Lenient Parallel Languages". In *Proceedings of 1991 FPCA*. Springer Verlag, August 1991.

[SN90] M.S. Squillante and R.D. Nelson. "Analysis of Task Migration in Shared-Memory Multiprocessor Scheduling". Technical Report 90-07-05, Univ. of Washington Dept. of Computer Science, September 1990.

[TLC85] M.M. Theimer, K.A. Lantz, and D.R. Cheriton. "Preemptable Remote Execution Facilities for the V-System". In *Proceedings of the 10th SOSP*, pages 2–12, December 1985.

[vECGS92] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. "Active Messages: a Mechanism for Integrated Communcation and Computation". In *Proceedings of the 19th ISCA*, pages 256–266, May 1992.

[Wan91] P. Wang. "An In-Depth Analysis of Concurrent B-Tree Algorithms". Master's thesis, MIT, January 1991. Available as MIT/LCS/TR-496.

[WBC+91] W. Weihl, E. Brewer, A. Colbrook, C. Dellarocas, W. Hsieh, A. Joseph, C. Waldspurger, and P. Wang. "PRELUDE: A System for Portable Parallel Software". Technical Report MIT/LCS/TR-519, MIT LCS, October 1991. Shorter version appears in *Proceedings of PARLE '92*.

[WW90] W.E. Weihl and P. Wang. "Multi-Version Memory: Software Cache Management for Concurrent B-Trees (extended abstract)". In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 650–655, December 1990.

[ZBC+92] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. "Vienna Fortran — a Language Specification". Technical Report ICASE Interim Report 21, ICASE Nasa Langley Research Center, March 1992.