MULTITHREADING AND THREAD MIGRATION USING MPI AND MYRINET

Stephen Jenks
Dept. of Electrical Engineering and Computer Science
University of California, Irvine
Irvine, CA 92697
email: sjenks@uci.edu

ABSTRACT
The balance between CPU speed and interconnection network throughput in distributed memory parallel computers varies with each generation of systems, but the trend is that CPUs are gaining performance faster than the interconnection networks. This means that remote data accesses are becoming more expensive relative to local accesses in terms of CPU cycles. Therefore, remote memory access mechanisms that were suited to a previous generation of parallel machines may be less appropriate for current clusters.

This research evaluates a multithreaded programming paradigm with cached remote memory accesses and thread migration to exploit array locality on a cluster with Myrinet. The approach, called Nomadic Threads, was originally developed for the CM5, but has been adapted to use MPI on Linux clusters. The results show that the current surfeit of CPU power vs. network throughput dramatically changes scaling characteristics of some programs while others behave much as they did on the decade-old CM5.

KEY WORDS
Parallelism, Multithreading, Migration, Interconnection

1 Introduction
The Nomadic Threads architecture [1] is a multithreading, frame-based abstract architecture designed to evaluate the efficacy of thread migration to exploit array-based spatial locality. Nomadic Threads (NT) supports both migration and conventional remote memory access mechanisms to allow direct comparison of the two approaches. An implementation of the NT runtime system [2] was developed for the Thinking Machines Connection Machine 5 (CM5) [3]. The runtime system includes efficient software caching of remote memory accesses in order to make the comparison between them and migration as fair as possible. In addition, a parallelizing compiler that takes an intermediate graph language, IF1 [4], produced by a SISAL [5] compiler and generates parallel code was built. The compiler builds code that uses migration or remote memory access or a combination of the two to support performance comparisons.

Many benchmark programs exhibiting various data access patterns and degrees of parallelism were tried using both migration and cached remote memory access to evaluate when one approach should be used over the other. The results [6] showed that migration is good for exploiting spatial locality, but some access patterns caused poor performance (the migration equivalent of thrashing).

These tests were performed on a CM5, with its seeming balance between CPU performance and network throughput. Modern cluster computers have much higher CPU performance, yet network throughput has not grown as much. Even high-performance interconnection networks, like Myrinet [7], have not kept pace with CPU speed increases since the CM5 days. This paper evaluates the impact of this imbalance in CPU speed vs. network throughput on migration and caching remote access to see if the results differ. Section 2 gives background on multithreading, its alternatives, and migration. Section 3 introduces Nomadic Threads, while section 4 describes the MPI [8] implementation. Section 5 explores the performance of NT on clusters and how it differs from the CM5.

2 Related Research
Multithreading [9, 10] tolerates latency of remote memory fetches by keeping the CPU busy while the fetch occurs. Once a long-latency operation is initiated, the CPU executes instructions from threads other than the one waiting for the operation’s results. While ready threads are available, the CPU performs useful work as the long-latency operations complete. Another latency tolerance approach is prefetching [11], where remote memory operations are initiated before the results are needed. Both approaches have advantages and drawbacks [12], as multithreading requires the presence of sufficient parallelism to tolerate long-latency operations, while prefetching needs knowledge of upcoming data accesses to make requests sufficiently before their use.

In this paper, we focus on non-blocking software multithreading implemented on conventional processors rather than hardware approaches, such as simultaneous multithreading [13]. One widely known research project to use this model is the Threaded Abstract Machine (TAM) [14], which was implemented for the CM5 and other machines. In the non-blocking multithreaded model, threads are sequences of instructions that, once started, run to completion without blocking. These threads consist of one or more operations or parts of operations. A thread is started when all the data it needs to execute is present in its frame, a memory block for inputs, outputs, and working storage.
Migrating a thread of control to a remote node to access data it needs is an approach to reducing the number of remote memory access messages. If the thread of control, which is not necessarily the same as a thread, as defined above, can exploit spatial locality and access several data items on the formerly remote, but now local, node, it can stay on the node and do so with no further communication required. A number of research projects, including Olden, have showed migration to be quite effective at exploiting locality for data structures such as lists and trees, where caching is not very effective [15].

3 Nomadic Threads Architecture

The NT architecture uses standard definitions for threads and frames, above, and defines a combination of a frame and the threads that use that frame as an activation. Threads, frames, and activations are the basic building blocks of NT programs and are built by the compiler. Threads are generated by the compiler from the IF1 graph operations. Though thread scheduling is very inexpensive, only taking a few tens of operations or less, it should be avoided if possible. For this reason, the compiler combines operations to build maximal threads. In many cases, dependencies prevent nodes from being joined with others, so most threads are small and consist of only a few operations. Others, such as mathematical computations, can contain many operations and produce several results.

Frames contain storage for thread inputs, thread outputs, working space, and overhead items required for synchronization. The frame contains information used to schedule and migrate the activation, as described in section 4.3. The frame stores the identity and node of the parent activation that spawned it, which allows the activation to return results to and synchronize with its parent. The rest of the structure is tailored to the threads of the activation.

Activations are the active entities in the Nomadic Threads architecture. An activation is the logical combination of a frame with its associated threads. Activations coincide with IF1 program graph boundaries and with compound nodes, which perform significant groups of operations and contain several subgraphs.

The NT architecture provides the array construct for use by application programs. Arrays are linear collections of elements that may either be distributed across the nodes of the system or local to a single PE. Multidimensional arrays are arrays of arrays, according to the SISAL language approach to constructing arrays.

4 MPI Implementation of Nomadic Threads

Implementation details of much of the NT runtime system are discussed in [2]. The creation of frames, handling of calls and returns, and scheduling of threads and activations have not changed from the CM5 implementation, thus they are not repeated here. This section details the implementation of the remote memory access mechanisms provided by NT, migration and caching remote memory access, using MPI and contrasts that with the CM5 implementation.

4.1 Communications Architecture

Nomadic Threads programs are different from most MPI programs. Rather than sending data to nodes that need it, a Nomadic Threads activation either migrates to the node that contains the data or dynamically performs a remote memory fetch if the data is not local. In either case, the message size required is quite small (from 64 bytes to 512 bytes) and even smaller messages are used for coordination and requests. This worked very well on the CM5, which had a network architecture well-suited to small messages.

On the CM5, the NT runtime system used the CMMD message passing library. In particular, CMMD provided an active messages [16] implementation called CMAML for coordination messages and remote memory requests. Bulk data transfer calls were used for migrations and to transfer remote memory fetch responses, i.e. “cache” lines. The active messages consisted of five 32-bit words. The first word held the address of the message handler, while the other four words were passed to the handler as arguments. This approach was very efficient, because no identifier decoding, tag checking, or protocol processing was needed. Since the CM5 used the Single-Program, Multiple-Data (SPMD) model [17], the handler addresses are known to all the node programs. Active messages took just 5-10 microseconds to transfer their small payload between nodes.

4.2 Message Handler

The NT runtime system is very asynchronous and less strictly coordinated than most MPI programs. The extremely lightweight thread scheduler runs all ready threads from the current activation before letting the activation scheduler choose a new activation with ready threads. When the activation scheduler is notified to switch activations, it initiates a polling operation to handle any messages that may have come in while the activation was running.

On the CM5, the poll call hid the message transfer details from the application programmer, and incoming active messages were passed to their user-specified handler routines during the poll call. Bulk transfers also had callbacks that operated similarly. As such, diverse operations in different pieces of the system could occur because of a single poll call receiving multiple messages.

The message poll operation is much more complex with MPI. Now, messages are received and handlers dispatched according to the tag or other information. Since commonly used MPI implementations, MPICH and LAM-MPI, are not thread safe, the NT implementation uses a single-threaded message poll function that dispatches incoming messages based on tags, as shown in Figure 1.

When the NT runtime system starts on each node,
The message poll function extracts testing to determine if 
non-blocking receive calls for each of the message types (tags) that can be received by the program. The message poll function is then called by the activation scheduler or thread programs often enough to keep message traffic moving. The poll routine uses MPI TestAny to determine if messages have arrived. When a message is received, its MPI_TAG is used to initiate the proper action, as shown in the figure. Migration activity is described in section 4.3, while section 4.4 elaborates on remote memory access operations. Less complex actions are described below. Once the action is complete, a new MPI_Irecv call is issued to prepare for another message of that same type. After initiating the new receive request for recurring messages, the message poll function checks again for other messages that may have arrived. It keeps checking for and receiving incoming messages until none remain, at which point, control is returned to the code that initiated the polling operation.

Several distributed array creation, deletion, and modification operations are handled by the message poll routine. Local arrays are created, destroyed, and modified locally, but distributed arrays are partitioned across all the nodes used by the program, so the operations must occur on all nodes. Distributed array creation and deletion are needed quite frequently because of SISAL’s functional nature.

Like the distributed array operations, the shutdown request is sent to all nodes as a signal that the program is complete and they should clean up and terminate. The shutdown request is handled within the message poll code by setting a global flag that terminates the activation scheduler and returns control to the node’s main function.

4.3 Migration

Migration consists of moving the state of computation to a remote node. The frame-based nature of the current Nomadic Threads runtime system greatly simplifies migration, because the frame contains much of the state of an activation and its threads. When a thread requests an array element that turns out to be remote, the runtime system, which knows the array partitioning scheme, determines the node where the data resides and notifies the requesting thread that the data is remote. The requesting thread then requests its associated activation to migrate to that remote node. It also specifies that after the migration, the thread should run again so it can fetch the data that will be new local after the migration.

This leads to a key point about migration: migration eliminates temporal locality. While migration is aimed towards enhancing spatial locality by making vast quantities of data local (i.e., the entire address space of a node), the act of migrating from one node to another makes all recently-used items on the source node remote when the activation arrives at the remote node. For this reason, programs that exhibit greater temporal locality than spatial locality are poor candidates for migration.

The implementation of migration is straightforward. The migrating activation’s frame and the identifiers of threads to run upon arrival are sent to the remote node as a message. When a migration message is received, it is delivered directly into a frame data structure created before the call to MPI_Irecv. The message poll function extracts the list of runnable threads and inserts them into the activation’s ready queue. Finally, the activation is “activated” by placing it into the activation scheduler’s queue.

4.4 Caching Remote Memory Access

Remote memory access is the other approach to data access besides migration implemented by Nomadic Threads. It is based on split-phase remote access commonly used with multithreading. One thread initiates the long-latency request for remote data and another thread (or threads) are activated by the response to the request. Meanwhile, other threads can run, helping the program tolerate the latency of the remote request. This latency tolerance of split-phase remote accesses alone is not enough, particularly when it takes thousands of cycles to get remote data and there may not be enough parallelism to tolerate that. Therefore, the NT runtime system fetches blocks of remote data and provides a software cache to allow subsequent and nearby fetch requests immediate access to the cached data. This caching is facilitated by the write-once nature of SISAL, so no cache coherence is needed.

When a thread makes an array memory access request, the runtime system checks if the data is local or remote. If the data is remote, but the needed element is cached, or local, the request is satisfied immediately. In cases where the remote request is not cached, messages are used to get the remote data. If the array is distributed, the runtime system sends a request to the appropriate remote node for a cache line (64 array elements) starting on the nearest lower 64-element boundary. If the array is remote and not distributed (it exists entirely on another node), the array ID encodes the array’s node. When the message containing the cache line arrives back, the message poll routine
Table 1. Average Remote Operation Time

<table>
<thead>
<tr>
<th>Activity</th>
<th>CM5</th>
<th>MPI/Myrinet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activation Migration</td>
<td>69 µsec</td>
<td>17.5 µsec</td>
</tr>
<tr>
<td>Remote Cache Line Fetch</td>
<td>77 µsec</td>
<td>44 µsec</td>
</tr>
</tbody>
</table>

This disparity between performance gains in processor speed and network speed between the CM5 and a modern cluster makes programs behave differently than on the CM5. Where the CM5’s processor and network were balanced, the cluster node’s CPU greatly outpaces the network speed. This makes remote accesses much more expensive per CPU cycle than on the CM5, thus the balance between communications and computation is dramatically changed.

5 Performance Evaluation

This section describes the performance of Nomadic Threads on a cluster of nodes containing a single Athlon CPU connected with Myrinet. The cluster runs the ROCKS Linux distribution [18]. Sixteen cluster nodes were allocated exclusively to these measurements during the experiments. The CM5 used for comparison was a 128-node machine at the University of Adelaide and was used for Nomadic Threads development from 1996 through early 2000. The Athlon processors in the cluster are roughly two orders of magnitude faster than the SPARC processors in the CM5, but the Myrinet is only about one order of magnitude faster. The CM5 data network could achieve up to 20MB/sec, while the cluster’s Myrinet has a maximum throughput of 200MB/sec.

This disparity in performance gains between processor speed and network speed is marked in a way similar to the CM5. Where the CM5’s processor and network were balanced, the cluster node’s CPU greatly outpaces the network speed. This makes remote accesses much more expensive per CPU cycle than on the CM5, thus the balance between communications and computation is dramatically changed.

5.1 Runtime System Measurements

The most relevant performance metrics for NT programs are the time it takes for an activation to migrate and the time it takes to fetch a remote cache line. Beyond the approach used for remote memory access, a program that migrates to fetch remote memory performs almost exactly as much computation as the same program built to use caching remote memory access (CRMA). Thus, the number of migrations/cache line fetches and their cost determine differences in execution time. NT benchmark testing shows a single Athlon CPU to be roughly 75 times faster than a single CM5 CPU. The next section will show how parallel application performance differs between the two.

Table 1 shows average remote operation timing for the CM5 and the MPI/Myrinet cluster. Though the cluster’s CPUs and network are much faster, the time for remote operations has not benefited much. Neither operation approaches the ten times faster that the network speed improvement would imply. On the CM5, migration was not significantly faster than a remote cache line fetch because it required three separate transactions, while two transactions were required for a cache line fetch. Using MPI, migration now takes a single MPI_Send operation (however that is implemented on the underlying network), while remote fetches obviously still take two operations.

5.2 Application Benchmark Measurements

The application benchmarks that follow are only a portion of the Nomadic Threads benchmarks run on both the CM5 and the MPI cluster. These benchmarks are relatively simple, as they are intended to evaluate access patterns and locality rather than perform useful work, though many do. Those presented are chosen either because they have interesting characteristics or their behavior is markedly different between the CM5 and the cluster. In all cases, the problem sizes run on the cluster are much larger than those on the CM5, because the CM5 was quite limited due to the single node memory size (32 MB). When execution time is reported, it does not include reading inputs and writing outputs, but is the elapsed time from when the first activation is created until all nodes are done with the application.

The first benchmark is a matrix multiplication enhanced to increase spatial locality. In MGTC, one source matrix is transposed so that the dot product operations take place along the columns. This suits migration well, because corresponding elements of each column always reside on the same node, due to the block partitioning scheme. MGTC exhibits a great deal of locality, so CRMA should be okay, but often the data from both source matrices will be remote and need to be fetched. Figure 2 shows the timing results for the CM5. Migration scales very well, while the CRMA version is not as good, but still scales fairly well. The “Mixed” version used CRMA for the transpose operation and migration for the matrix multiplication.
Figure 3 shows the speedup of MGTC on the cluster. The left side of the figure shows the migration version scaling well. The superlinear speedup is an artifact of the columnar array access pattern making poor use of the CPU cache when all the matrices are on one node. Once the matrices are partitioned between multiple nodes, the execution with migration scales very nicely. Only on many nodes and with many threads per node is there enough parallelism to make up for the extreme expense of remote fetches.

Another illuminating benchmark is smooth image rotation. This program rotates an image by an arbitrary angle using a stencil pattern to fetch multiple neighboring pixels and smoothes the result to avoid jagged lines. The program exhibits both temporal and spatial locality, so CRMA is better than migration, but, as Figure 4 shows, both did fairly well on the CM5. Figure 5 shows that CRMA is better than migration on the cluster with an image five times the size of the CM5 image, but both scale somewhat and perform better than a single processor by a fair margin. Here, two or four threads per node tend to outperform eight threads per node. There is sufficient locality in this program that the additional parallelism is not needed.

Many programs that run on parallel computers have sequential algorithms in them. Normally, such algorithms are kept to a minimum, but some programs require significant sequential segments, particularly algorithms with data dependences that prevent their execution in parallel. When such a sequential program must access data that is distributed across the nodes of a parallel machine, the expense of fetching data is prohibitive, even with caching. If the data access pattern permits it, migration is a much better answer; by moving the work to the data, the sequential code can run more efficiently. The final program simulates this by performing a sequential access over a large distributed 2D array. In this case, the CM5 and the cluster exhibited similar behavior, as shown in Figure 6, though the data set size on the cluster is 50 times larger than the one used on the CM5. Because CRMA is quite expensive on the cluster, CRMA falls behind migration even more than on the CM5. The bump on the CM5 CRMA graph is likely due to the difference in network speed beyond the local group of nodes in the fat tree.

6 Conclusions and Future Research

The results shown in this paper demonstrate that communication speed and latency is more of a problem than ever. While CPU speed within cluster nodes is measured in billions of operations per second, most interconnection networks only transfer hundreds of MB/second. In contrast, a CM5 CPU could do tens of millions of instructions per second and the network could transfer tens of MB/second, so the balance has tipped towards CPU speed.

This change leads to execution characteristics on clusters that are very different than on the CM5. Demand-based remote memory fetch operations are now relatively more expensive than migration in the Nomadic Threads architecture. All remote operations are vastly more expensive in terms of CPU cycles than they were before, as well. Therefore, programs that did very well with remote fetches on the CM5 may not scale as well on a cluster, as several benchmarks demonstrate. The results also show that for cases where migration did extremely well on the CM5, it continues to perform well on the cluster. Migration is an efficient operation using MPI over Myrinet, so algorithms that take advantage of the spatial locality it provides scale well.

References

[1] Stephen Jenks and Jean-Luc Gaudiot. Exploiting locality and tolerating remote memory access latency using thread


