A Multithreaded Runtime System With Thread Migration for Distributed Memory Parallel Computing

Stephen Jenks and Jean-Luc Gaudiot
Electrical Engineering and Computer Science
University of California, Irvine
Irvine, CA 92697-2625
{sjenks, gaudiot}@uci.edu

Keywords: Multithreading, Thread Migration, Parallel Compilation, Distributed Memory

Abstract
Multithreading is very effective at tolerating the latency of remote memory accesses in distributed memory parallel computers, but does nothing to reduce the number or cost of those memory accesses. Compiler techniques and runtime approaches, such as caching remote memory accesses and prefetching, are often used to reduce the number of remote memory accesses. Another approach to reduce the number of remote memory access messages is thread migration, which exploits spatial locality by making the entire memory space of any processing element (PE) local.

Nomadic Threads are compiler-generated fine-grain threads that migrate between PEs to access data. The runtime system that implements the Nomadic Threads architecture has several attractive features, including support for migrating significant amounts of work, the ability to mix caching and migratory remote memory accesses, and low scheduling complexity. Our straightforward implementation shows that migration may be easily added to many multithreading runtime systems. We also present some unique behavior characteristics of migratory programs to help determine when migration is appropriate or not.

1 INTRODUCTION
Programs running on distributed memory parallel computers often require access to data partitioned across the processing elements (PEs) of the system. Many programs access such remote data by means of request and response messages, which consume system resources and have much higher latency than local memory accesses. Multithreading [1] is often used to tolerate remote memory access latency by keeping the CPU busy while a remote memory request is fulfilled, but it does not reduce the number of such requests. Compilation techniques, such as those implemented by the SUIF project [2], can often reduce the amount of remote data and the number of messages needed, but normally cannot completely eliminate remote accesses. Adding caching to remote memory fetches reduces both the number of fetches and their latency. Our research has resulted in an alternative to caching remote memory accesses that uses migration of threads of control to make the entire memory space of a PE local. The Nomadic Threads (NT) approach [3] migrates the work to the data rather than data to the work.

The goal of this paper is to show the ease with which multi-thread migration can be added to a frame-based multithreading architecture and to explain some of the unique features of our runtime system implementation. We defined the Nomadic Threads architecture and built the runtime system in order to support our research into the effectiveness of migration at exploiting array access locality. The runtime system supports both migration and caching equally well in order to facilitate fair testing of programs with either access mechanism. It also supports mixing of migration and caching in order to take advantage of the best features of both. In this paper, we define our migration mechanism, show how we optimized the performance of our caching fetch operation to eliminate a performance penalty that can occur with multithreading, and explain the efficient two-level scheduling approach used to simplify the runtime system.

In addition to the runtime system, we also developed a parallelizing compiler that generates multithreaded code from functional language programs. The NT-SISAL compiler builds threads from an IF1 [4] intermediate form graph generated by the Lawrence Livermore OSC SISAL compiler [5]. The compiler generates either migratory or caching fetch code at the programmer’s directive, or it can mix caching and migration automatically based on heuristics developed during our benchmarking activities. Details of the compiler are not presented here, but some performance results are shown.

This paper is organized as follows: Section 2 presents an overview of previous research into multithreading runtime systems and thread migration for remote memory access. Section 3 defines the important features of the Nomadic Threads architecture. In section 4, we describe the implementation of the runtime system, including the scheduler, array accesses, migration, and cache line management. Section 5 shows some results of our research into locality exploitation using the NT runtime system and outlines some of our ongoing research. Finally, section 6 summarizes the developments explored in the paper.
2 PRIOR MULTITHREADING, LATENCY TOLERANCE, AND MIGRATION RESEARCH

Multithreading is one approach to tolerating the 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. As long as there are ready threads available and the thread switch overhead is low, the CPU performs useful work while the long-latency operations complete. The other major approach to such latency tolerance is prefetching [6], where remote memory operations are initiated, either automatically or under program control, before the results are needed. Both approaches have advantages and drawbacks over the other [7], as multithreading requires the presence of sufficient available parallelism to be able to tolerate long-latency operations, while prefetching needs advance knowledge of upcoming remote data access patterns to make requests far enough ahead of their actual utilization.

Figure 1. Threads and IF1 nodes. (a) Simple and (b) Compound

In this paper, we are primarily interested in non-blocking software multithreading implemented on conventional processors rather than hardware implementations, such as simultaneous multithreading [8, 9]. One of the most widely known research projects to use this model is the Threaded Abstract Machine (TAM) [10], which was implemented for the CM-5 [11] and other distributed memory 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. Most simple IF1 operations, for example, can be combined into threads if their dependencies are met, as shown in Figure 1(a). The simple nodes are combined to form a single thread, Thread 1. The compound ForAll node shown in Figure 1(b) is a complex, long latency operation, so it can not be combined with any threads that depend on its results. Because it is a long latency operation, two threads are needed. The first, Thread 2, spawns the ForAll activation, while Thread 3 is activated upon completion of the ForAll activation and handles the return of results and cleans up after the activation. The Fetch operation is a split-phase operation, so it too needs two threads: one to initiate the fetch operation, Thread 4, and the other to be activated upon the return of the data, Thread 5. 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. The thread and frame approaches used by Nomadic Threads are based on the definitions used in TAM, the systems are not compatible.

Migrating a thread of control to a remote PE 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, PE, it can stay on the PE 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 [12, 13]. Migration can also be used to balance the load on PEs, as in Cilk’s work stealing approach [14]. The migration approach used in Nomadic Threads is less complex than Olden’s, because of the frame-based approach used here. It is also under explicit program control, though normally visible only to the compiler, rather than implicit, as with Cilk.

3 NOMADIC THREADS ARCHITECTURE

The Nomadic Threads 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. Array data structures are also provided by the architecture.

3.1 Threads

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.

3.2 Frames

Frames are data structures whose type is defined by the compiler. Many instances of each frame type can be created as needed during program execution. Frames contain storage for thread inputs, thread outputs, working space, if needed, and overhead items required for synchronization. Figure 2 shows the frame from the inner loop of a simple
The nt_info item contains information used to schedule and migrate the activation, as described in section 4. The nt_migrationInfo item stores the identity and PE of the parent activation that spawned this one. This allows the activation to return results to and synchronize with the calling activation. All frames contain those two items, but the rest of the structure is specific to the threads of the activation, as generated by the compiler. Some of the items are inputs to the frame, but most are inter-thread results generated by threads in the activation. Any syncCount items are used to synchronize threads that have inputs from multiple other threads in the activation.

```c
typedef struct {
  NT_ActInfo nt_info;
  NT_MigrationInfoType nt_migrationInfo;
  double forallOut3_1, frameItem4,
  frameItem5, bodyResult7;
  unsigned char nt_thread5_syncCount;
  int k, y, i, j;
  NT_ArrayDescriptor a, b, frameItem2,
  frameItem3;
} ForAll_act4_NTFrame;
```

Figure 2. Matrix multiply dot product loop frame

### 3.3 Activations

Activations are the active entities in the Nomadic Threads architecture. An activation is the logical combination of a frame with its associated threads, as shown in Figure 3. Activations coincide with IF1 program graph boundaries and with compound nodes, which perform significant groups of operations and contain several subgraphs. These subgraphs contain parts of the compound node’s operation, such as a loop initializer or body, or an else clause in a conditional. Each subgraph consists of operations that are formed into threads and may spawn additional activations.

![Figure 3. An activation is a frame and its threads](image)

### 3.4 Arrays

The NT architecture provides the array construct for use by application programs. Arrays are linear collections of entities that may either be distributed across the PEs of the system or local to a single PE. As shown in Figure 4, multidimensional arrays are arrays of arrays, according to the SISAL language approach to constructing arrays. The top level of a multidimensional array consists of an array of array descriptors, each of which points to an array of elements that make up a lower dimension of the overall array. If the array has more than two dimensions, each of the intermediate-level arrays contain array descriptors until the last array level contains elements. Generally, the arrays have uniform size, but ragged arrays, where lengths of subarrays vary, are supported.

![Figure 4. Multidimensional array handling](image)

The NT architecture provides the NT_ArrayDescriptor identifier for the application to use to designate arrays. Supported array operations include creation, deletion, copying, assignment, fetch, and upper and lower index manipulation. Arrays are specified to be either distributed or local at creation time. Though the architecture does not define the partitioning scheme for array distribution, the initial implementation is a straightforward block partitioning scheme. The normal fetch operations are generic and work with both local and distributed arrays with elements of any size. The details of both the migratory and caching fetch implementations are described in section 4.

### 4 RUNTIME SYSTEM IMPLEMENTATION

This section describes the implementation of the Nomadic Threads architecture via a user-level runtime system. The runtime system is a set of C functions that use the underlying message passing libraries provided for the target system. Our initial implementation used the CMMD [15] library on the CM-5, but we are currently porting the
Nomadic Threads Runtime System to use the Message Passing Interface (MPI) [16] libraries on conventional cluster computers.

4.1 Scheduling and Resource Allocation

Two of the primary tasks of the NT runtime system are to manage the allocation and deallocation of activation frames, and to schedule activations and their threads. The scheduler is implemented in two levels: one to manage activations and one to manage threads. This two-level approach was chosen to exploit the maximum possible locality by running as many of an activation’s threads as possible before switching to another activation.

4.1.1 Call and Return/Activation Creation

Activations are created by threads in other activations. Such a thread creates the frame and fills in the return information, including the parent activation’s frame address, the PE where it resides, and the thread to activate upon return. The spawning thread also fills in the input slots in the frame. This is the only time input slots are filled. Most of the time, activations are spawned without a specific ready thread, in which case, an initialization thread generated by the compiler knows which threads depend only on the inputs and are, therefore, ready. Other times, the spawning thread specifies which thread is to be made ready first. Then the new activation is placed in the ready queue.

When a spawned activation completes all the work assigned to it, the activation synchronizes with its parent activation. If the child activation has migrated to a different PE from the parent, it migrates back to the parent’s PE. When the child thread is on the parent’s PE and is activated, it activates the return thread the parent assigned during the spawn operation. This places the parent activation in the ready queue. When the scheduler activates the parent activation, the return thread is run. The return thread, which takes the address of the child’s frame as a parameter, retrieves any required return values from the child’s frame and stores them in the appropriate slots of the parent’s frame. The return thread then deletes the child’s frame and activates threads that depend on the return values.

Because each child activation must rejoin with its parent activation, the parent activation must be prevented from migrating so the child can find it. Therefore, the parent is locked down until all outstanding children have synchronized with it. Other approaches to this problem were considered, including forwarding pointers to allow returning children to follow the parent’s migrations, but the utility of such an approach appears to be quite low, especially in light of the complexity and overhead it would introduce. Generally, parents do not need to migrate while they have outstanding children, so locking them to a PE during that period does not impact their performance.

4.1.2 Activation Scheduler

Activations are the large-grain schedulable entities in the architecture. The activation scheduler maintains a queue of ready activations. When an activation is created, migrates to a PE, or is activated by a returning child activation, it always has at least one ready thread, so the activation is added to the ready queue. The activation scheduler picks the first activation from the ready queue, dequeues it, and calls the activation’s thread scheduler, described below. When the thread scheduler returns control to the activation scheduler, all of the activation’s ready threads have completed. Since there are no ready threads and the scheduler dequeued the activation, the activation will not run again unless a returning child activation activates one of the parent’s threads.

4.1.3 Thread Scheduler

An activation’s thread scheduler is a small compiler-generated routine that accesses the list of ready threads in a continuation vector in the nt_info structure of the activation’s frame. It extracts the thread ID and a parameter word for the first ready thread and jumps to that thread. When the active thread finishes, it may activate other threads that depend on its results. If the thread only has one dependent thread, control jumps directly to that thread and the thread scheduler is bypassed. A threads that has multiple dependent threads adds all but one of those threads to the continuation vector and then jumps the remaining one directly. When a thread has no direct dependents, control jumps to the activation’s thread scheduler which checks the continuation vector for the next thread to run. If there are no ready threads, control returns to the activation scheduler.

4.2 Migration

Migration in the NT architecture is very simple, consisting only of transmitting an activation’s frame data from one PE to another along with the list of threads to activate upon receipt of the activation. Though migration is straightforward, we will address its usage for array accesses below.

4.2.1 Array Access

The migratory fetch operation is very simple in that it either succeeds, because the requested array element is local, or fails, because the element is remote, in which case the fetch call returns the remote PE number. First, the fetch routine determines if the array is of the distributed or local variety. If it is not distributed, the current PE number is checked against the PE number in the array descriptor. If they match, a local array fetch operation is used to retrieve the element and store it into the frame, and a success value is returned. Otherwise, the PE number where the array resides is returned to the calling thread, so the activation can migrate to that PE to get the element.
If the array is distributed, each PE has the array header information, since distributed array creation is done simultaneously on all PEs. The fetch routine checks the bounds of the local segment of the array to determine if the desired element is local. If so, it computes the offset into the storage buffer and copies the element to the frame slot, just as with a local array fetch, and returns the success value. If the element is remote, the routine uses partitioning information to determine which PE has the element in question and returns that PE number to the calling thread.

4.2.2 Migration Implementation

When a fetch call returns a remote PE number, the activation must migrate to that PE to get the element. Other threads in the activation may be runnable locally first, however, so the activation must wait until there are no runnable threads before it can migrate. In addition, the activation may be locked down and unable to migrate because of outstanding child activations. Therefore, the thread that needs the remote data requests a migration, specifying the destination PE and that the thread itself should be run again when the migration completes. Multiple threads may similarly request migrations, possibly to different PEs. Migration requests are stored in a migration vector, which is similar in design and implementation to the continuation vector.

When control returns to the activation scheduler and the activation is not locked down, the frame along with the list of threads to activate upon arrival is sent to the requested destination PE (the last request made if multiple requests were made). Figure 5 shows a nominal migration timeline, consisting of a single message transfer, which is what we expect using MPI. When the fetch operation occurs, a localness check (denoted by FLC) determines that the migration is needed. Then, the migration should consist of a short message origination (MO) time, a message transfer time (MTT), and message reception (MR). Since activation frames tend to be small, the message transfer time should be quite small compared to the overhead for both message setup and reliable protocol. Figure 6 shows the CM-5 implementation to be much more complex than the nominal case. Because of the nature of the CM-5 block transfer mechanism, a setup handshake must first be used to make the remote node ready for the incoming frame. The setup messages are extremely small and fast on the CM-5, but they can be delayed if either PE is busy. Most frames are between 64 and 96 bytes in length and take a minimum of 118µs from initiation of the migration through message reception and insertion into the scheduling queue when the PEs are not busy. Modern reliable network architectures that only require a single user message for a frame transfer should perform much better for migration than the CM-5 does.

When the activation arrives at the destination PE, all the threads that requested migrations are made ready and the activation is inserted into the ready queue. All local array elements will be fetched successfully when the threads run, but those on other PEs need further migrations, which will be requested by the threads that need them. Because the activation can only migrate to a single PE at a time, pathological cases may require $O(n^2)$ fetch attempts. Happily, this case does not appear in real-world programs often, if ever, so normal activations are satisfied after a single migration or two when remote array elements are required.

4.3 Fetch with Caching

Caching fetch operations are more complex than migratory fetches because of cache management and interactions with the scheduler. We describe the architecture of caching fetches below, followed by the implementation details.

4.3.1 Array Access

The caching fetch operation is a split-phase operation in which the requesting thread does not depend on the fetch results, but one or more other threads are activated by the results. Normal split-phase fetches require control flow changes as shown in Figure 7(a). The fetch handler initiates
the element retrieval and quickly returns control to the calling thread. When the element is retrieved and stored in the designated frame slot, the thread waiting for the data is awakened. Even if the element is local and the fetch completes immediately, the thread scheduler must be used to run the newly ready thread. While thread scheduling is very fast, it does require runtime system operations to get the next ready thread’s identifier and parameter, if any. The overhead of these additional scheduling operations caused programs using caching fetches to run roughly 30% slower than their migratory counterparts on a single PE.

In order to reduce this penalty for caching, we enhanced the fetch mechanism to eliminate the additional scheduling steps if the element is local or in the cache. The revised control flow is shown in Figure 7(b). If the element is local or cached, the fetch handler simply returns “True” and does not add the next thread to the continuation vector. The calling thread detects the return value and activates the dependent thread directly, eliminating the scheduling overhead. This approach achieves parity with migration fetch times for local and cached elements. If the element is remote, the regular control flow is needed because of the delay in retrieving the remote item.

4.3.2 Cache Line Fetch Implementation

The caching fetch mechanism is more complex than the migratory one because it must handle more cases. It must differentiate between local and remote distributed arrays as well as local and remote “local” arrays and handles each differently. The least complex case is for distributed arrays. The routine first checks if the requested element is in the local segment of the array, and if so, writes it to the frame slot just like the migratory fetch does. If the element is remote, the routine checks if it is in the cache. Each distributed array structure on each PE has a set of cache lines specific to the array. There are 512 cache elements per array, arranged into 16-element lines that are directly mapped by index number. Because of the direct mapping, line lookup is extremely fast. A bit mask and index base are used to check if the proper index is present. If so, the element is written to the frame and the fetch is complete.

If the element is not cached, the cache line containing it must be fetched from its home PE. Using partitioning information, the routine determines the element’s PE, the starting index for the cache line to be fetched, and the number of elements to fetch. Because of the partitioning scheme, the line could extend past the end of the remote PE’s segment. In that case, only the elements that are part of the remote segment are requested. Since multiple PEs may have elements belonging to a single cache line, multiple sources for cache line portions are supported and fetches of different portions do not invalidate a cache line. The remote PE is sent a message requesting the array elements and specifying the destination cache line address. The routine on the remote PE uses a block transfer mechanism to send the array elements to the requesting PE and store them in the proper memory locations. If multiple threads request elements from the same cache line that will be satisfied by the same remote PE, only the first message is sent. When the requested cache line elements arrive, a handler routine notifies the cache manager, which then traverses a queue of requests for that array. Each request that can be satisfied by the newly arrived elements is handled and the dependent thread is placed on the ready queue. Since all the requests are satisfied without interruption and the elements are stored into the frame slots from the cache, cache invalidation cannot affect a ready thread’s data availability.

The “local” array case is more complex than the distributed array case. If the array descriptor shows that the array resides on the PE, the fetch is handled by a local fetch operation. If the array is remote, the cache must be checked. Unlike distributed arrays, local arrays do not have a presence on all PEs, so there is no preassigned space for their cache lines and their array information. In order to
access array elements for a “local” array on a remote PE, the array information, including high and low bounds, must be fetched. A table of array information structures is used to store the remote array information. Each of those structures also has a 512-element block of 16-element cache lines. Accessing the remote array information is the first step of the fetch. Once the array information table entry has the proper array information, whether it already existed or was fetched, the bounds and element size data are used to make the request for the proper cache line elements from the remote PE. While the request is outstanding, the array information entry is locked to prevent invalidation. Other fetches for elements of a cache line that has a request outstanding are queued until the cache line data arrives.

Figure 8 shows the timeline of a cache line fetch operation. Components of the timeline are: cache miss determination time (TCM); message origination times (TCO), transfer times (TT), and reception times (TCR) for both request and response messages; the time to respond to the cache line request (TRR) with the desired memory items; and the time to notify or wake up (TWU) the threads waiting for elements of the newly-fetched cache line. The time to create and send the request message as well as receive the response, update the cache, and activate any waiting threads keep the requesting processor busy. After the request is sent, the requesting processor will run available threads until the response arrives. The time spent by the remote processor to service the request is at the expense of any ready threads on that processor. A cache line fetch of 16 8-byte doubles requires a total of 110µsec on the CM-5.

Figure 8. Cache line fetch timeline

5 RESULTS AND FUTURE EXPLORATION

Our benchmark results from the CM-5 show that programs that use migration to access remote array data are very efficient when there is significant spatial locality to exploit, as expected. Since migration makes the entire memory space of a PE local, the spatial locality can have a significantly greater spread and less regularity than that exploitable by cache line fetches. Migration is not effective at exploiting temporal locality, however, because migration to another PE excludes local accesses to elements that were recently used. Caching exploits both temporal and spatial locality, so it generally performed very well in our benchmarks.

While migration performed very well in some cases, there are some operations that are significantly slowed by migration. In parallel loop cases, matrix transpose, for example, where the source element(s) and destination element(s) for each loop iteration are on different PEs, a migratory activation must bounce between the PEs at every step, thus incurring two (or more) migration costs per iteration. This behavior, which we call the “writer-returns property,” prevents exploitation of locality and hampers execution speed.

We present a summary of the timing and locality exploiting characteristics of a small set of the benchmarks that we ran on the CM-5 here. More performance details, particularly on the parallel benchmarks can be found in [17]. The earlier benchmarks in [3] are hand-coded, while the ones we present here and in [17] were compiled from SISAL source code using our parallelizing compiler and code generator. When our current effort to port Nomadic Threads is complete, we will compare how a cluster with both Myrinet [18] and Ethernet perform compared to the CM-5. The message transfer time on the cluster is on the order of ten times faster than that of the CM-5, so migration will only take 10 to 15 microseconds, compared to well over 100 microseconds on the CM-5. Cache line fetches will still require two messages, request and response, so migration will see a greater speed improvement compared to a cache miss. On the other hand, processor speed has increased by about 100 times over the CM-5’s SPARC processor and memory capacity is also 20 to 100 times greater per PE. This makes checking the presence of an item in the cache much faster, so programs with many cache hits will perform very well. Our preliminary estimates do not show many cases where caching and migration will significantly change their relative performance characteristics on modern clusters vs. their performance on the CM-5. In other words, migration will still do well for cases where it did well on the CM-5 and the same is true for cached remote accesses.

Table 1 shows some statistics for several benchmarks. These examples are a small sample of the benchmarks we used to characterize the behavior of Nomadic Threads. These benchmarks are illustrative of various access patterns that are either well suited to migration or are handled poorly by migration, thus the most important features of the performance space can be seen from them.
Caching does quite well in all of the example benchmarks, while migration’s performance is mixed. The matrix multiply benchmark is a simple implementation that transposes one of the operand matrices to align the columns to the dot product operation has maximum locality. The hit rates for both migration and caching are quite good, but they do not tell the entire story. In this case, the large locality provided by the aligned vectors is extremely well suited to the migratory inner product activations racing across the PEs. The caching activations still needed to fetch cache lines from all the PEs, thus moving more data in the long run. Smooth image rotation uses a stencil to access neighboring pixels to perform rotation of an image by an arbitrary angle with weighted smoothing of the resulting pixels. While this stencil is good for migration, the temporal and spatial locality provided by caching wins out over the spatial locality of migratory stencil operations.

The 2D sequential access pattern is one in which, due to dependences, for example, a distributed array is accessed sequentially. In the caching case, all the elements of the array are fetched a cache line at a time to the one PE running the sequential code. With migration, the activation jumps to each PE in turn, thus greatly reducing the amount of data to be moved. Migration’s advantages for such sequential operations are significant and we will continue to study them.

Finally, the array reversal benchmark is a worst-case example of how to make migration perform poorly. In this extremely simple case, an activation jumps to a PE containing the element of the source array to be placed into its opposite end of the destination array. Then, the activation jumps back to the original PE to store the element in its destination array. This is a clear example of the writer returns property in which no locality is exploited and migration’s behavior is poor. Because of this type of performance penalty, migration should only be used in cases where the writer-returns property is absent or minimal.

Moving the array elements a cache line at a time is a much better approach, though the fetchbot concept proposed below is a migratory approach that may close the performance gap.

An important behavior to note for migratory activations is the lack of latency tolerance for remote accesses. Because all ready threads must have completed before an activation can migrate, as described in section 4.2.2, no multithreading latency tolerance is possible. The only way to keep the CPU busy while the frame is being transmitted and after it is gone is to have sufficient parallelism that other work remains on the PE. Incoming activations as well as other locally-spawned work can keep the CPU busy, but multithreading does not benefit migration by itself. Since caching fetches may be mixed with migratory activations, multithreading is still an important feature of the Nomadic Threads architecture.

While migration is able to exploit spatial locality very well in many cases, it is not well suited to others. It does show tremendous potential for sequential program segments, in which a distributed array must be visited sequentially because of dependence between the elements. In many of these cases, it is less expensive to move a small activation between the PEs in the system than to fetch all the required data to the single PE running the sequential program segment.

In addition, migration will be useful when access patterns are not easy to determine at compile time. Here, we propose a fetchbot, an activation that can be sent to gather the required remote data. A fetchbot would be less complex than a normal activation, but would have a payload storage area in which it can store its retrieved data (or it can send data back to the waiting parent activation as it finds it). The fetchbot would be able to follow run-time computed strides for distributed arrays. Fetchbots have the potential to be more efficient than cache line fetches for accesses with stride. We will prototype fetchbots once the conversion to MPI is complete.

### 6 CONCLUSION

We have shown that migration provides significant benefits to some remote array access patterns on distributed memory parallel computers. We have also shown how straightforward the implementation of migration is for a frame-based multithreaded architecture. In fact, the caching remote memory access implementation is far more complex than migration and its fetch mechanism. For these reasons, we believe that migration is not only reasonable, but desirable to add to multithreading architectures and runtime systems in the future. Because of migration’s ease of implementation, it should be present to support the applications it is particularly suited for, especially some sequential operations and irregular access patterns with complex spatial locality. In addition, the introduction of

<table>
<thead>
<tr>
<th>Access Type</th>
<th>Matrix Multiply</th>
<th>Smooth Image Rotation</th>
<th>2D Sequential</th>
<th>Array Reversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Locality Type</td>
<td>Spatial</td>
<td>Both</td>
<td>Spatial</td>
<td>Spatial</td>
</tr>
<tr>
<td>Migration Hit Rate</td>
<td>94%</td>
<td>77%</td>
<td>&gt;99%</td>
<td>0%</td>
</tr>
<tr>
<td>Migration Time (sec)</td>
<td>75.7</td>
<td>7.5</td>
<td>14.7</td>
<td>2.5</td>
</tr>
<tr>
<td>Caching Hit Rate</td>
<td>81%</td>
<td>92%</td>
<td>97%</td>
<td>94%</td>
</tr>
<tr>
<td>Caching Time (sec)</td>
<td>311.6</td>
<td>3.8</td>
<td>33.4</td>
<td>0.14</td>
</tr>
</tbody>
</table>

Table 1. Sample benchmark statistics
“fetchbots” may be quite useful for access patterns with complex strides.

We also presented three important behavior characteristics of migration. The first is the writer-returns property, in which some migratory activations travel to one PE to fetch inputs, yet it must return to another PE to write its result. If this pattern is repeated at each iteration of an inner loop, a potentially severe performance penalty may result. The second feature of migration is that it ideally takes only a single message transfer, though this is not the case on the CM-5. This contrasts with a cache line fetch, which always takes two transfers, a request and a response, even on modern architectures. Finally, when migration operates in concert with multithreading, the remote memory access latency tolerance characteristics of multithreading are excluded by the need for all ready threads to complete before an activation migrates. Threads from other activations, given sufficient parallelism, can be used to keep the CPU busy while the activation is transmitted and afterwards.

REFERENCES