Nomadic Threads: A Runtime Approach for Managing Remote Memory Accesses in Multiprocessors

Stephen Jenks and Jean-Luc Gaudiot

CENG Technical Report 95-01

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213) 740-2251

February 1995

Nomadic Threads: A Runtime Approach for Managing Remote Memory Accesses in Multiprocessors

Stephen Jenks and Jean-Luc Gaudiot

CENG Technical Report 95-01

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213) 740-2251

February 1995

Abstract

This paper describes a multithreaded runtime system for distributed memory multicomputers that significantly reduces the number of message transfers when compared to conventional "remote memory access" approaches. Instead of statically executing on its assigned processor and fetching data from remote storage, a Nomadic Thread transfers itself to the processor which contains the data it needs. This enables Nomadic Threads to take advantage of the spatial locality found in many algorithms that use arrays of data, because segments of arrays are made local by the migration of a thread to the proper node. By reducing the number of messages and taking advantage of locality, the Nomadic Threads approach allows programs to execute faster than conventional approaches while providing a simple runtime interface to compilers. Nomadic Threads are currently implemented using Active Messages [1] for the Thinking Machines Corp. Connection Machine 5 [2].

1. Problem Statement

Distributed-memory multicomputers provide tremendous processing and communications capability that is potentially much more scalable than shared-memory multiprocessors. This computational power is achieved at the expense of ease of programming such systems and high latencies for remote memory operations. Parallelizing compilers for standard languages, standard languages enhanced with parallel features (C* [3], for example), and functional languages, such as SISAL [4] and Id [5], attempt to solve the programmability problem by exploiting the parallelism in programs and mapping concurrent operations to different processors. These languages and their compilers allow programmers to write a single program that is partitioned across the nodes in the system. Software multithreading models, like the Threaded Abstract Machine (TAM) [6] or the Nomadic Threads runtime, provide runtime support for parallel programs on conventional distributed-memory multicomputers.

Programs may be partitioned among the processors in the system in several ways, depending on the architecture of the machine. Some distributed memory multicomputers, like the Warp machine [7] from Carnegie Mellon, support systolic processing where the user program is split into a series of pipeline stages and data is sent periodically from one stage to the next. Other machines use a Multiple-Instruction, Multiple Data (MIMD) approach, where each node runs an independent task that interacts with the tasks on other nodes, as required. Many current distributed memory systems operate in a Single-Program, Multiple Data (SPMD) mode, in which all nodes run the same program independently but with their own data. These node programs interact with each other to fetch data and exchange results. Our approach presented in this paper is currently aimed at SPMD machines and applied to the TMC CM-5, but can be easily applied to MIMD machines as well.

Programs running on SPMD machines typically need access to large arrays of data, often larger than the memory available in a single node. These arrays may either be stored on special nodes in the machine whose sole purpose is the storage of such arrays or partitioned across the nodes of the system much like parallel programs are in MIMD systems. When a program running on a given node needs to access some data that is not in the local memory of the node, it normally performs a remote fetch operation to acquire the required data. This fetch involves a request message to the node where the data resides and a response message containing the data from the remote node to the requester. The latency of the remote memory fetch depends on the characteristics of the underlying hardware, including the communications network, the network interface, and interrupt service times (if interrupts are used), as well as the runtime that provides remote memory operation support to the user program. No matter how fast the hardware, the two messages needed for a remote memory fetch greatly increase the average latency of memory fetch operations vs. those of a conventional uniprocessor running at the same speed.

This paper presents a runtime approach that eliminates at least one half of such messages by changing what happens when access to a remote piece of data is required. Section 2 reports existing hardware and software approaches to accessing remote data. Section 3 discusses our Nomadic Threads scheme in detail and its implementation on the CM-5. Section 4 presents results comparing the performance of Nomadic Threads execution to equivalent programs with remote memory fetches. Finally, section 5 summarizes the results of the paper and lists some future work and research directions to further develop Nomadic Threads and array handling for distributed-memory machines.

2. Background

Much work has gone into both reducing the latency of remote memory fetches and tolerating the remaining latency. Efforts have been directed at designing special-purpose architectures as well as software runtime systems.

2.1. Hardware Architectures

Dynamic Data-Flow systems [8] tolerate communications latency as data items are sent from the node that generated them to nodes that use them by exposing enough parallelism in the application program to keep the processors busy while the communications takes place. These systems were built as specialized machines without a von Neumann processor-to-memory architecture. Because these research machines are complex, have high overhead when matching data with instructions, and do not have strong industry support, they quickly fall behind the power curve of CPU speed and are being overtaken by commercial RISC processors. Data-flow research continues with Monsoon [9] and other modern data-flow machines.

While data-flow machines handled the transfer of data tokens well, they had trouble with large arrays, because it was not reasonable to send copies of such arrays around the system. I-structures [10] were added to solve the problem. I-structures provide storage space on certain dedicated nodes in the system to hold arrays. I-structure *fetch* requests are "split-phase" operations consisting of two messages: a request to the I-structure storage node, and a response with the data. If the requested data element is not yet available, the response will be delayed until the data becomes available (is written). The operation is called split-phase because the caller does not wait for the response to the fetch request. When the response arrives, the instruction waiting for the data will be matched and executed just like any other data-flow instruction. The concept of I-structures can also be applied to data-driven execution on conventional machines, as discussed below.

Instead of building special-purpose processors that require new compilers, some computer makers have connected commercial RISC processors, each with local memory, together with high-speed interconnection networks. For example, the Thinking Machines Corp. CM-5 uses SPARC processors connected by FAT trees and Intel's Paragon series uses i860s in a mesh. Such machines provide scalable computing and memory and can use existing compiler back-ends, since the processors are standard. Their advanced interconnection architectures can use techniques such as wormhole routing to reduce message latencies, and the CM-5 provides polling I/O to avoid the latency of interrupt handling. This class of distributed-memory machines is much more popular than data-flow machines and is more scalable than shared-memory multiprocessors. Because these types of distributed-memory computers comply with international and de facto commercial standards, are readily available, and can contain many processors and much memory, they are the best choice for large-scale parallel processing today.

2.2. Multithreaded Architectures

Functional languages like SISAL and Id are very good at exposing both fine-grained and coarse-grained parallelism in programs. It is easy to utilize fine-grained parallelism in data-flow machines, but it is more difficult to map such fine-grained execution to a collection of conventional processors connected by a network. The problem is that message transfers take two or three orders of magnitude longer than instruction execution time. Instead of executing one instruction for every one or two tokens that arrive, as in data-flow systems, many instructions must be executed to balance the message transfer time in order to increase the efficiency of the processor significantly above zero.

Multithreaded systems collect sequential operations together into *threads* of instructions that can run on conventional or hybrid data-flow processors. Each of these threads becomes ready to run when its inputs are available, much like data-flow operations. Ready threads may be executed in parallel or in series based on available processing resources.

Threads derived from a given source-language function or algorithm constitute a *code block* and share a *frame* for their input and scratch space. The combination of a frame and the code block that uses it is called an *activation*, as shown in figure 1. The frame stores the state of execution for the threads and can represent a single iteration of a loop or other suitable unit of execution that can be executed independently and concurrently with other, similar activations. Each node in the system can have one or more activations, many of which may use the same shared code block with different data in each of their frames. The threads associated with these activations run concurrently and perform the source algorithm in parallel.

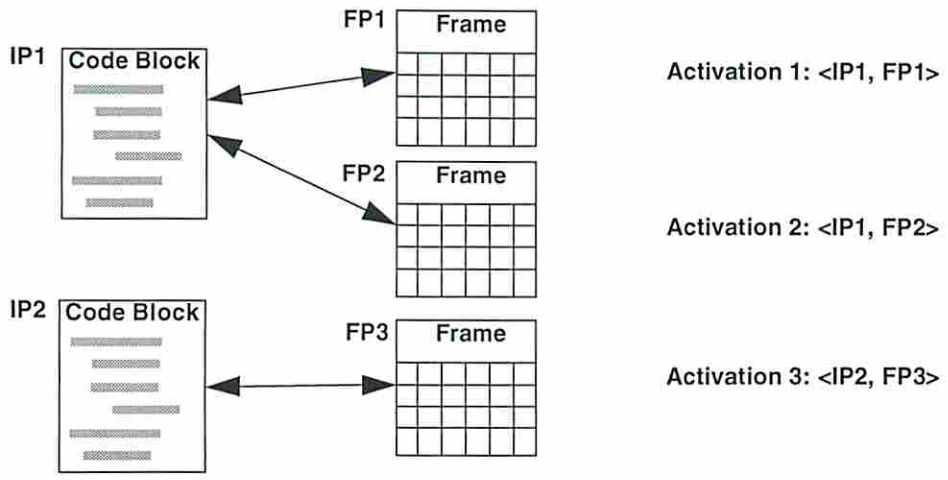


Figure 1. Relationship between Code Blocks, Frames, and Activations

When a thread performs a split-phase fetch from an I-structure, that thread does not wait for the result from the fetch command. The result from the I-structure fetch is used as input for the thread that follows the one that issued the fetch instruction. Once all its inputs are available, the subsequent thread may run. Because threads do not wait for fetch completion, thread scheduling is very simple: Any thread whose inputs are all available may be run, and once a thread is started, it is run to completion without preemption, although I/O operations may cause activity unseen by the thread.

Since thread scheduling is simple, multithreaded systems have reasonably low overhead and perform efficiently. The goal of multithreaded systems, like their data-flow predecessors, is to keep the processors in the system busy doing application program work while I/O (remote memory fetch) takes place, so the latency of remote operations is “tolerated.” The way to accomplish this is to make threads large enough that their computation time is not completely overwhelmed by the time required to fetch required data from another node. Having many threads available to run also helps keep the processor from becoming idle, but, while thread scheduling is simple, its computation is not free, so having many small threads requires more overhead than fewer, larger threads.

The Threaded Abstract Machine (TAM) is a software implementation of a multithreaded architecture that runs on conventional distributed-memory computers. It executes compiler-generated threads in parallel and emulates I-structure operations for array handling. A TAM thread is a collection of sequential instructions that do not jump out of the thread and only reference data available in the current frame. Results from I-structure fetches and data from other threads are placed into *inlets* in the frame. Each thread has a set of inlets that, when full, allows the thread to become enabled.

TAM threads are constructed by a compiler for the functional language Id and consist of primitive assembly language-like instructions in a language called TL0. This TL0 code is then converted to code for the target machine, normally a CM-5. Runtime code provides for the transfer of data, I-structure handling, and message passing that underlies the application code.

For the interested reader, Dennis and Gao provide an in-depth discussion of multithreaded architectures and principles in [11].

3. Nomadic Threads Approach

If Mohammed cannot go to the mountain, the mountain will come to Mohammed.

In our approach, we shall help Mohammed go to the mountain, because it is easier than moving the mountain.

The basic idea of our Nomadic Threads approach is to send the activations to the location of the data they need rather than performing remote fetches. By doing this, the request/response message pairs used in remote fetches are replaced with far fewer, but somewhat larger, activation transfer messages. Once at the remote node, Nomadic Threads can take advantage of spatial locality in array accesses.

3.1. Overview

Nomadic threads eliminate the concept of remote memory fetches by transferring the activation from the current node to the node where the data resides. Therefore, all fetches are local. This has the effect of reducing the number of messages needed to access data on a remote node from two, for I-structure fetches, to one message, for transferring the activation state. In addition, once the activation is running on the remote node, it may access as much local data as needed. For algorithms that exhibit spatial locality of references, the Nomadic Threads approach requires *far* fewer than half the messages required for remote memory fetch schemes. In the worst case, where the activation required a data item from a remote node and one from the current node in each itera-

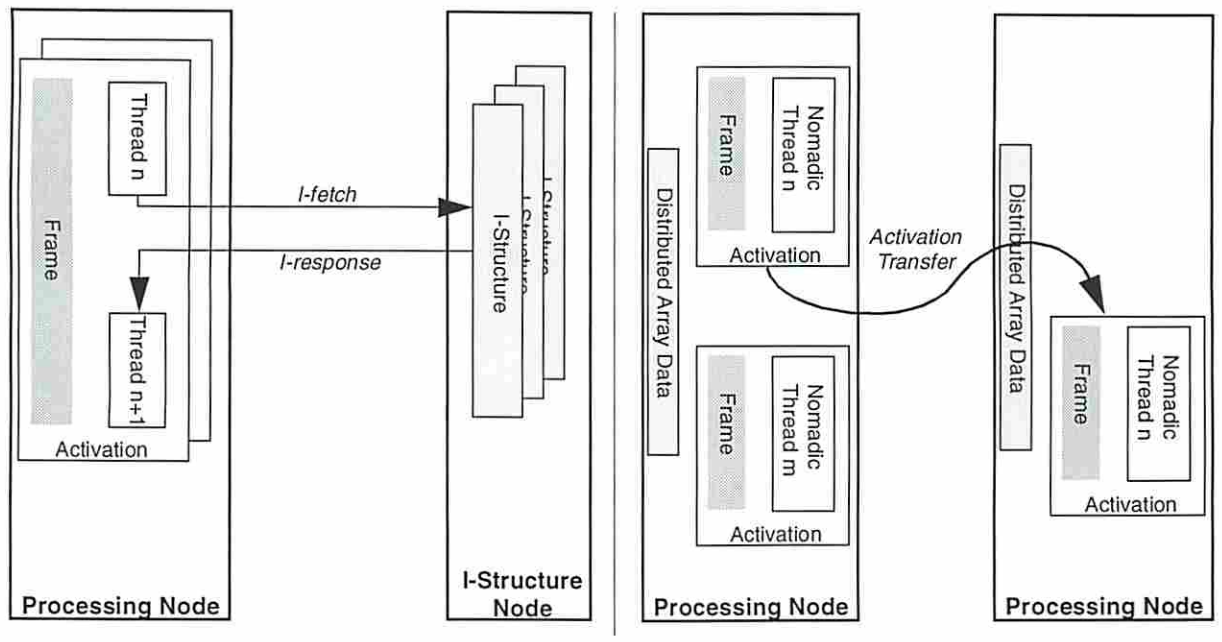


Figure 2. Remote Memory Fetch vs. Nomadic Thread Control Flow

tion, a Nomadic Threads program requires the same number of messages as a remote fetch program. Figure 2 shows the message traffic and control flow for both the conventional approach and the Nomadic Threads approach.

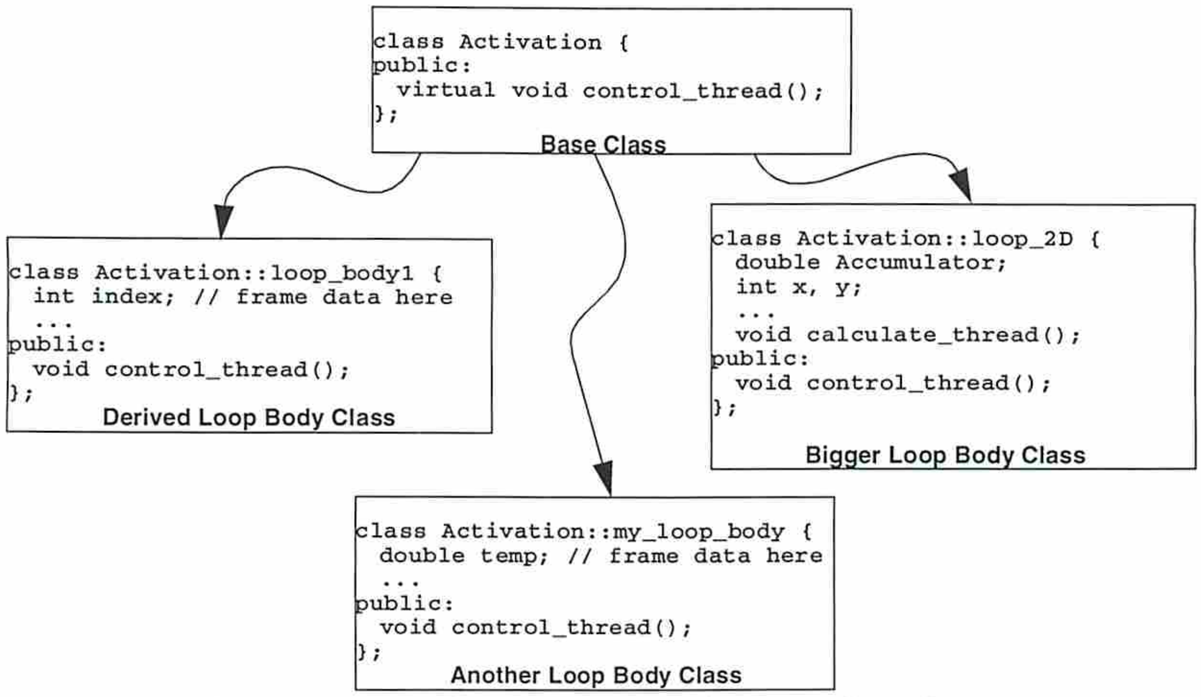
As the figure shows, the Nomadic Thread activation transfer requires only one message, while the remote memory access requires two messages. The activation transfer message contains the state of the activation, so it is somewhat larger than either of the two messages used in the remote memory access. Although the single activation transfer message is larger than the remote memory access messages, the transfer time is less, except in extreme cases, because the overhead required to set up two message transfers is approximately twice that required for one. In addition, once a message transfer is initiated on most machines, the transfer rate is so high that a few extra words do not significantly increase the transfer time. The CM-5 is an exception to this and will be discussed in section 3.2.

Besides reducing the number of messages, another benefit of the Nomadic Threads approach is that threads can take advantage of spatial locality of their array accesses. Many algorithms reference array elements that are very close sequentially to the last element accessed. If a Nomadic Thread activation transfers itself to a node to access one array element, it will be able to access other array elements on that node as local accesses. In the conventional remote memory access approach, separate accesses are required for each array element on the remote node. Fetching several array elements around the current one and caching them can reduce the number of messages in the conventional scheme at the expense of larger fetch response messages. Caching can also be used by Nomadic Thread activations to reduce the number of activation transfers.

The Nomadic Thread approach requires that data arrays be partitioned across computation nodes so threads running on a node can access some elements of an array locally. Conventional remote memory access approaches do not require that arrays be stored on computation nodes. Instead, some system nodes can be dedicated to I-structure handling, while others only perform computations. Because this paper is assuming implementations on distributed-memory computers based on commercial processors in which nodes are generally functionally identical, no special hardware support is provided for handling I-structures. This means that the I-structure nodes must do the same work to handle array accesses as do the nodes in the Nomadic Threads approach. If nodes are dedicated to handling I-structures, they may be very heavily utilized and possibly become overloaded, thus bogging down the rest of the system. In cases where data arrays are partitioned across the nodes of the system, as in Nomadic Threads, the array access load is better distributed, so network load may be lighter. The drawback is that the nodes spend some time handling messages because of either activation transfers (Nomadic Threads) or fetch requests (remote memory access) instead of performing computations.

3.2. Implementation

Like activations in other multithreaded schemes, activations in the Nomadic Threads approach consist of a *frame* for temporary information storage and one or more code segments that make up the *threads* that share the frame. The concept of a frame with associated threads may be thought of as conceptually similar to C++ objects and their associated methods. Activations in the current Nomadic Threads implementation are implemented as C++ classes derived from a base class, as shown in figure 3. The base class has no data elements defined and contains a single method called `control_thread` that has no return value. The figure shows several classes derived from the `Activation` class. Each of the derived classes has its own `control_thread` method and storage for any required local data, such as accumulators, loop indices, or temporary


```
class Activation {
public:
  virtual void control_thread();
};
```

```
class Activation::loop_body1 {
  int index; // frame data here
  ...
public:
  void control_thread();
};
```

```
class Activation::loop_2D {
  double Accumulator;
  int x, y;
  ...
  void calculate_thread();
public:
  void control_thread();
};
```

```
class Activation::my_loop_body {
  double temp; // frame data here
  ...
public:
  void control_thread();
};
```

Figure 3. Nomadic Thread Activations are C++ classes

storage. Classes derived from the `Activation` superclass may also have other methods that can be called by the control thread during its execution.

The control thread of an activation (or any thread called from it) may directly access any array elements local to the node on which it is running. Some sort of distributed array handler must be provided by the runtime to allow access to local array elements and to inform the thread as to the location of non-local array elements. In the current implementation, arrays are C++ objects with access methods to facilitate operations on the elements. When an attempt is made to read or write an array element that is not local, the methods return the address of the node where the data element resides.

When a thread discovers that it needs to perform a non-local read or write, the thread sends a copy of its activation frame to the appropriate remote node. When the activation reaches the remote node, the control thread is called, completing the transfer of control. Normally, the activation on the source node is deleted after the migration is completed. If a *fork* operation is desired, it may be implemented simply by not deleting the source activation after the activation transfer. The resulting activation and the source activation will execute independently. *Join* can be implemented by having threads *rendezvous* at a node determined before the fork takes place.

The activation transfer mechanism in the CM-5 implementation of Nomadic Threads is based on *Active Message* transfers. The Connection Machine Active Message Library (CMAML) part of the CMMD message-passing library provides a set of active message routines that allow small messages to be sent between nodes in the CM-5. An active message is a small message that contains up to four 32-bit data words and specifies a handler that is called when the message is received by the destination node. Active messages are very fast and are sometimes large enough to handle request/response messages for remote memory access schemes. A remote memory access handler routine can decode the request, fetch the requested data, and send a response message to the requester, where another handler can deliver the data to the correct place. Unfortunately, active messages are so small that some requests or responses are too big to fit, so multiple messages or block transfers must be used. The activation frames used in Nomadic Threads are typically larger than four words, so they are too large to fit in a single active message. However, because the CM-5's communications mechanism transfers Active Messages very quickly, activation transfers are fast enough to keep the processor nodes busy.

The CMAML library provides a mechanism to transfer blocks of data from one node to another, but it requires a complete request/response message pair just to set it up. Due to this overhead, the CMAML block transfer approach is slower than a block transfer protocol layered on active messages and tailored to Nomadic Threads. The CM-5 network guarantees message delivery, but does not guarantee in-order delivery, so the protocol specifies where the data in each message fits in the block being transferred, and how large the block is, as well as transfer sequence numbers and the source node address. All this information takes one of the data words of each message, so only three data words are available in each message. This added overhead turns out to behave better than the block transfer mechanism in CMAML and results in faster transfers in benchmark tests.

When a Nomadic Thread decides to transfer its activation to another node, it calls a block transfer routine on its frame. The transfer routine splits the frame into active messages and sends them to the specified destination. The handler function on the destination node receives each of the messages in any order and stores their data into the proper place in an incoming message buffer. The handler function has as many incoming buffers as it needs to store all the message sequences that it is receiving at any time. Once the handler receives a complete block transfer, it queues the activation whose frame is contained in the block for execution once the handler is complete. When

```
% ******* Matrix Multiplication code fragment
% ******* From SISAL 1.2: A Brief Introduction and Tutorial
% ******* by David C. Cann

type OneDim = array[ integer ];
type TwoDim = array[ OneDim ];

function MatMult( A,B:TwoDim; M,N,L integer returns TwoDim )
  for i in 1, M cross j in 1, L
    S := for k in 1, N
      returns value of sum A[i,k]*B[k,j]
      end for
  returns array of S
  end for
end function
```

Figure 4. Matrix Multiply Function in SISAL

the handler returns to the main program, the control thread of each queued activation is called, thus continuing the execution of the just-migrated activations. Once the control thread of an activation completes, the frame is discarded. In some cases, a thread may reinsert its frame on the bottom of the queue in order to wait for some event that has not occurred, such as a rendezvous or data becoming available.

4. Performance Observations

A benchmark program that performs matrix multiplication was chosen initially to explore the performance of Nomadic Threads because matrix multiplication is an extremely parallel operation in which each result element can be computed independently. Further, it entails many remote memory accesses which should exercise well our scheme. However, each such computation requires access to entire rows and columns of the source matrices. Our benchmark program uses a "naive" approach to the matrix multiplication algorithm, as shown in the SISAL code fragment of figure 4. This algorithm is "naive" because it is written as it would be for a uniprocessor or a shared memory multiprocessor, and does not explicitly use any parallel constructs or message passing. In other words, it is not particularly designed to run on a distributed memory multicomputer. A naive algorithm is desirable because the Nomadic Threads runtime is intended for use with a compiler that parallelizes programs without explicit parallelism.

Since no compiler exists yet that uses the Nomadic Threads runtime, the algorithm was hand-translated into Nomadic Threads code using only techniques that are reasonable for a smart parallelizing compiler. For purposes of comparison, a conventional remote memory access version of the benchmark was also created using the same techniques. A very simple array partitioning scheme was used in which the matrices were partitioned along either their row axis or their column axis. It was assumed that a smart compiler would be able to determine the order of access of the source matrices, so source matrix A (see figure 4) is split along its row axis (so entire columns exist on nodes), and matrix B is split along its column axis (complete rows exist on nodes). The destination matrix is split along the row axis like matrix A.

Each activation in either of the parallel programs is responsible for calculating one element of the destination matrix and starts execution on the node where that element resides. The conventional parallel matrix multiply (PMM) activations step through the inner loop of the algorithm, fetching elements from the source matrices multiplying them, and summing them until the loop is finished. The Nomadic Threads parallel matrix multiply (NT-PMM) activations start on the node where their destination element resides, but they may migrate anywhere within the system to get data needed to calculate the inner loop. Once an activation completes its loop, it returns to its home node where it writes its result into the destination matrix.

Figure 5 shows the execution times of PMM and NT-PMM for various matrix sizes and number of processors used on a 32-processor CM-5. Timings for a C implementation of the naive matrix multiplication algorithm run on an Intel 486DX/2-66 PC running Linux are included for comparison. All cases used double-precision floating point arithmetic and matrix elements. Table 1 lists the execution times of the benchmarks. The table shows that the NT-PMM benchmark runs significantly faster than the normal remote memory access PMM and the uniprocessor 486. It is interesting to note that the NT-PMM timings for the 512x512 case and the 1024x1024 case in either the 32-node or 8-node group differ by almost exactly a factor of 8, which is to be expected from an O(n^3) algorithm.

Table 1 shows that for very small matrix sizes, NT-PMM runs faster on 8 nodes than it does on 32 nodes. This is because the matrices are partitioned across fewer nodes, so each local segment is larger, and locality plays a larger role. Once the matrices grow large enough, the contributions from the number of nodes performing computation overwhelms the contribution from differences in the per node array segment size, so the 32-node version performs much better than the 8-node NT-PMM.

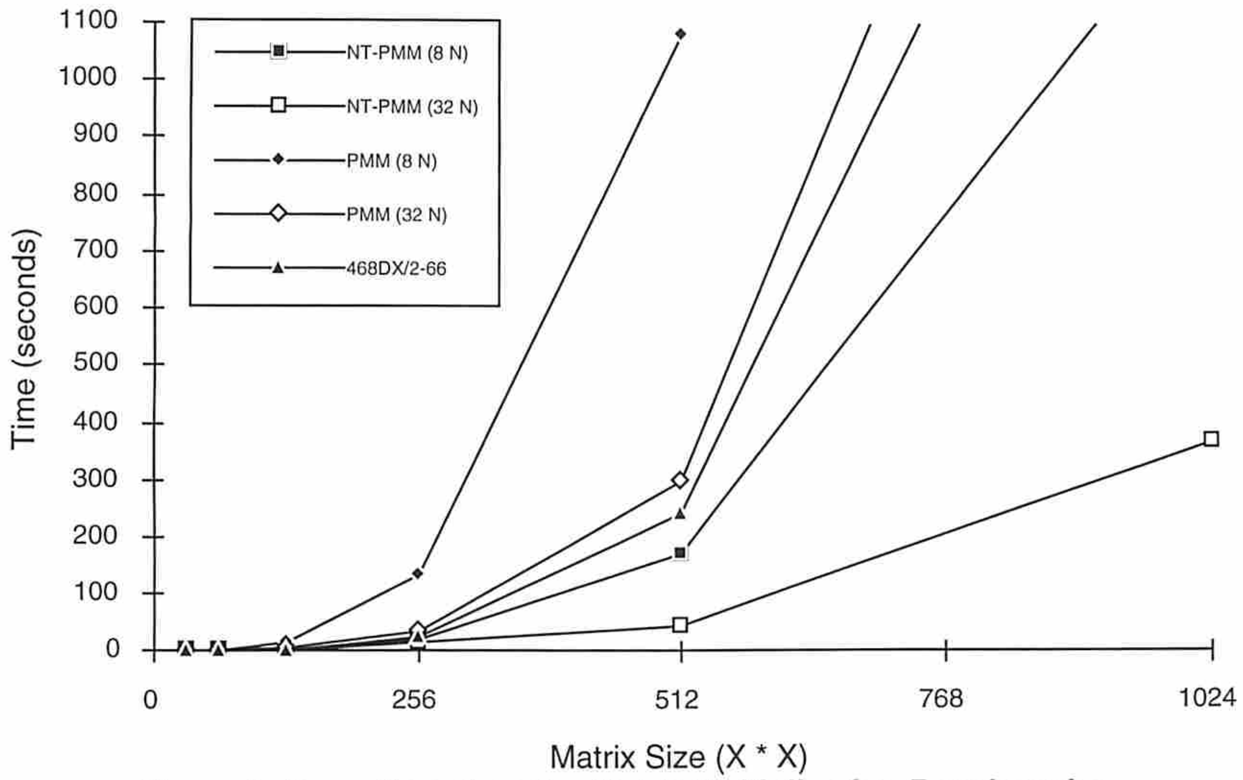


Figure 5. Execution Time For Matrix Multiplication Benchmarks

Table 2 lists the execution times for NT-PMM in the 512x512 matrix case with various processor configurations on the CM-5. Figure 6 shows the corresponding speedup graph. As shown in the figure, the speedup curve is nearly linear, so the computation scales up well and the amount of required computation is sufficiently large that each of the processors in the 32-node CM-5 make a contribution of nearly their full potential.

Table 1: Matrix Multiplication Execution Time Listing

Matrix Dimensions	NT-PMM 8 Nodes	NT-PMM 32 Nodes	PMM 8 Nodes	PMM 32 Nodes	486DX/2 66 MHz
32x32	0.08	0.09	0.26	0.08	0.02
33x33	0.1	0.11	0.33	0.15	0.02
64x64	0.46	0.42	2.18	0.57	0.19
128x128	2.87	2.72	16.22	4.8	2.12
256x256	19.85	15.28	132.62	36.43	27.26
512x512	170.03	46.26	1079.97	297.42	243.93
1024x1024	1362.15	368.51	Not Avail.	2543.09	2150.18

Table 2: NT-PMM Execution Time

Number of Nodes	Execution Time (seconds) 512x512 Matrix Multiply
1	1341.35
2	674.42
4	342.22
8	170.03
16	86.02
24	61.58
32	46.26

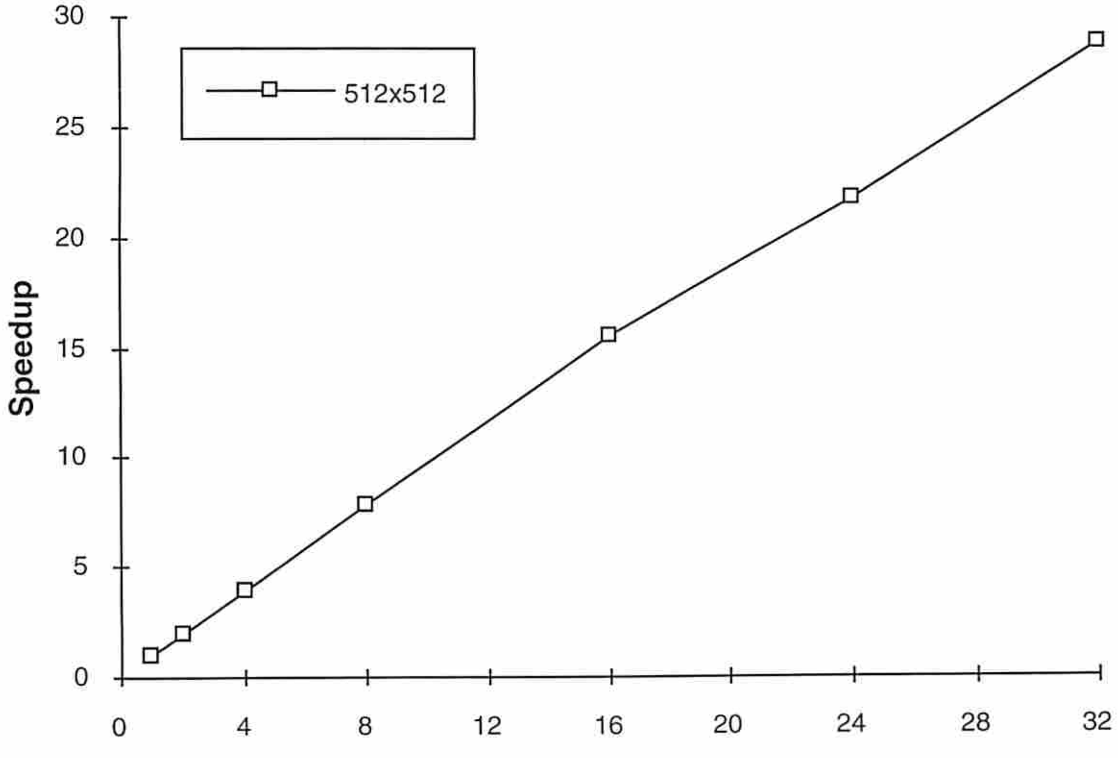


Figure 6. Nomadic Threads Speedup on the CM-5

The high utilization of all of the nodes of the CM-5 illustrates the advantages of the data partitioning scheme used in the Nomadic Threads approach. First, it shows that all the nodes on the system were actively participating the computation. In I-structure systems, some of the nodes are dedicated to array handling only, so they could not participate in computation, thus hardware utilization may not be as great. It also shows that the load can be distributed based on the partitioning of the matrices. In the matrix multiplication case, the matrix accesses are regular and uniform, so the matrices were equally partitioned among the nodes. In other algorithms that access array elements non-uniformly, it is possible to distribute the load by choosing a partitioning scheme where heavily used areas of arrays are partitioned across as many nodes as reasonable. Nomadic Threads will then migrate among the nodes as required to gather the data they need.

5. Summary and Future Research

This paper has demonstrated a runtime technique for SPMD machines where, instead of bringing data to the program, the program goes to the data. The Nomadic Threads runtime approach provides a mechanism that allows multithreaded applications to run on conventional distributed-memory multicomputers, such as the CM-5, and always access elements of arrays partitioned across the nodes of the machine as local array operations. By migrating program activations to nodes where the data they need resides, far fewer data transfers are needed when compared to traditional remote memory access approaches. Once an activation migrates to a node that has data it needs, it can take advantage of spatial locality of array accesses, as the next array access will probably reference an element on the current node.

Benchmark tests using a "naive" implementation of a matrix multiplication algorithm showed that the Nomadic Threads version is much faster than a remote memory fetch version. The execution time of Nomadic Threads matrix multiplication benchmark scaled linearly with increases in matrix sizes for large matrices, and the speedup scaled nearly linearly with the number of processors. These encouraging results make it possible for Nomadic Threads to have a bright future in multicomputer runtimes.

The most critical step required for future development of Nomadic Threads is the implementation of a compiler backend to generate code that interfaces with the Nomadic Threads runtime mechanism. A SISAL compiler that generates Nomadic Threads code would permit many different benchmark types to be used to study the performance of activation migration for various array access and partitioning techniques. Before such a compiler can be designed, a general mapping from loop constructs to Nomadic Threads must be developed, and array partitioning techniques must be adapted or developed to support the execution of Nomadic Threads activations.

Further study on possible caching techniques is also needed. In cases where several array elements from the current node will be needed in the near future, but the activation must migrate before then because some required data is on another node, it is possible to collect the next few elements of the local data and carry them with the activation during its migration as a sort of cache. The benefits and effectiveness of such an approach need to be studied, and the number of data elements to cache must be traded off against the overhead of delivering larger activation frames and the overhead and complexity to determine if a needed data item is in the cache. This and other performance enhancements and tuning techniques should be researched in order to make the Nomadic Threads runtime approach as useful as possible.

References

[1] T. von Eicken, D. E. Culler, S. C. Goldstein and K. E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," *Communications of the ACM*, pp. 256-266, 1992.

[2] *The Connection Machine CM5 Technical Summary*, Thinking Machines Corporation, Cambridge, MA, 1991

[3] *C* User's Guide-Version 6.0.2*, Thinking Machines Corporation, Cambridge, MA, 1991

[4] J. R. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. R. W. Glauert, I. Dobes and P. Hohensee, *SISAL: Streams and Iterations in a Single Assignment Language: Language Reference Manual, version 1.2*, Technical Report TR M-146, University of California - Lawrence Livermore Laboratory, 1985

[5] R. S. Nikhil, *Id (Version 88.0) Reference Manual*, Technical Report CSG Memo 284, MIT Lab for Computer Science, Cambridge, MA, 1988

[6] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken and J. Wawrzynek, "Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine," 1991.

[7] M. Annaratone, F. Bitz, E. Clune, H. T. Kung, P. Maulik, H. Ribas, P. Tseng and J. Webb, "Applications and Algorithm Partitioning on Warp," in *Proceedings of COMPCON Spring '87*, 1987.

[8] Arvind, L. Bic and T. Ungerer, "Evolution of Data–Flow Computers," in *Advanced Topics in Data–Flow Computing*, J.-L. Gaudiot and L. Bic, (Eds.), Prentice Hall, Englewood Cliffs, NJ, 1991, pp. 3-33.

[9] D. E. Culler and G. M. Papadopoulos, "The Explicit Token Store," *Journal of Parallel and Distributed Computing*, Vol. 10, pp. 289-308, 1990.

[10] Arvind and R. E. Thomas, *I-Structures: An Efficient Data Type For Functional Languages*, Technical Report LCS/TM-178, MIT Laboratory for Computer Science, 1980

[11] J. B. Dennis and G. R. Gao, "Multithreaded Architectures: Principles, Projects, and Issues," in *Multithreaded Computer Architecture: A Summary of the State of the* Art, R. Iannucci, G. Gao, R. Halstead, Jr., and B. Smith, (Eds.), Kluwer Academic Publishers, Boston, 1994, pp. 1-72.