Thread Migration in Distributed Memory Multicomputers

Scott Milton

February 1998
This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

Technical.Reports@cs.anu.edu.au

A list of technical reports, including some abstracts and copies of some full reports may be found at:

http://cs.anu.edu.au/techreports/

Recent reports in this series:


Thread Migration in Distributed Memory Multicomputers

Scott Milton
Department of Computer Science
Australian National University
Canberra
Tel: +61-417-396-277, Fax: +61-3-99031077,
Email: scott@cs.anu.edu.au

January 1998

Abstract

While the paradigm offered by SMP designs is relatively clean one, programming paradigms offered on distributed memory platforms rarely offer the same simplicity of comprehension, and ease of use. The most common paradigms for programming distributed memory computers offer either distributed shared memory, or a complex and error prone message passing library.

We have implemented a runtime system for distributed memory platforms, Nomad which offers not only the transparency of data location provided by distributed shared memory systems, but also transparency of processing location by a highly optimised and lightweight thread migration mechanism.

We argue that thread migration is in many cases is actually a more efficient computation strategy than the data migration in conventional distributed shared memory systems.

Introduction

Symmetric Multi-Processor (SMP) machines typically provide a number of CPUs all of which address the same local memory. Programmers of these machines do not have concern themselves with data locality, but program with the expectation that any data can be accessed on the local bus with little overhead. Unfortunately contention between processors for memory accesses increases with the number of processors, so the architecture does not scale. Distribution of memory is sometimes necessary to achieve scalability, greater parallelism, and sometimes for accessing specialist resources which may be distributed around a network.

Unlike SMP machines, distributed memory machines normally do not allow direct addressing of all memory in the system, hence data must be copied around the nodes as required. Typically mechanisms for this are either pre-emptive or reactive.

Pre-emptive systems rely on knowing where data will be required at a particular point in the program, and having the node with that data sending that data at the appropriate time. Typically these mechanisms rely on message passing libraries, such as RPC [12], the AP-1000 [21], or CM-5 CMAM [32] [31].

Reactive systems rely on detecting when a node requires data which it does not currently have, and fetching that data before continuing. This is similar to the virtual memory mechanism on modern architectures whereby pages which are not resident in main memory must be fetched before processing of the accessing program can continue.

Generally we might expect pre-emptive programs to be relatively difficult to write, because of the requirement of the program to know its fu-
ture behavior. We might also expect pre-emptive programs to be relatively efficient, as the pre-emption decreases (or in some cases even eliminates) idle time as processors wait for more data.

Similarly we might expect reactive programs to be relatively easier to write as the programmer can more readily ignore future data requirements of the processors. We might also expect the programs to be less efficient as all remote memory accesses will require significant thread idleness, though not necessarily processor idleness.

Page Migration – Distributed Shared Memory Systems and Transparency of Data

Distributed shared memory systems (DSMs) normally offer the view that all the memory in the system is available locally. All memory is accessible as one large address space, and can be accessed by an arbitrary number of nodes in the system.

When a process attempts to access data that is not locally available, a fault mechanism can be invoked to fetch that data into the local memory from the remote processor which holds it. Typically there are several orders of magnitude difference [1] between local and remote access times for memory.

Much of the access time for non-local memory accesses is latency time. That is, there is a relatively high fixed cost in transmitting even a small amount of data between nodes. Hence a large memory segment (that is a page of several kilobytes) can be transmitted at one time with little overhead [18] on a LAN. Intimate area networks on tightly coupled processors, such as the AP3000 [9] and the Cray T3E [22], typically have smaller latency times, but also higher transmission speeds. We can expect to fetch whole pages on this architecture with similarly low overheads. We refer to the fetching of data in this way as page migration.

We take the view that DSMs make data location transparent, but not processing location. That is the mechanism allows the programmer to ignore location of data when writing programs, but not the location of the processing.

Thread Migration – Transparency of Processing

SMP Models, such as the Sun LWP [26], offer transparency of processing. That is, the system dynamically schedules threads to run on processors as the scheduler makes them available. A model is offered to the programmer where there are a large number of virtual processors available, and threads run on these virtual processors. The threads run on a number of physical processors when these are available.

There are few systems however which attempt to provide this model on distributed memory machines. Several systems, such as MPVM [4] do migration of processes on different machines, but normally not thread migration itself.

Virtual Memory and Memory Management Units

Most modern architectures offer virtual memory, and use a Memory Management Unit (MMU) to support it efficiently. An MMU provides indirect addressing, which involves a lookup from a program’s address to a physical memory address. This is a highly efficient mechanism because it is implemented in hardware.

Data is typically divided into physical pages, several kilobytes long, and memory addresses are divided into a page number, and an offset into the page. The MMU dynamically converts the virtual page number into a real page number and the offset is added to determine the real physical address.

Typical Address Space Layout in UNIX Processes

Typically a process’ address space is divided into text, heap and stack, as shown in figure 1. Text
This mechanism ensures a clean separation of heap and stack space, maximizes the available virtual address space, and normally prevents the stack from growing down and overwriting the heap in applications which use a very large stack. Only memory which is actually allocated is used, and most systems would exhaust their virtual memory before a stack could actually overwrite the heap.

Figure 1: Normal address space layout for UNIX processes

The stack consists of the machine code which is executed during the course of the process’ execution, and is not changed during the course of the execution. The address of the elements of the text do not change between invocations of the program.

The stack consists of the activation frames of the current process. This typically starts at some high address and grows downward through the memory. A procedure call (or method invocation) generates a new activation frame containing return information and automatic variables.

The heap stores other data used by the program, and typically grows upwards starting at the end of the text area. Calls such as malloc(2) [13] and language constructs such as new [10] typically exist to reserve more space in the heap.

Figure 2: Normal Memory Layout in UNIX Processes With Threads

When lightweight processes [26] are used each thread requires its own stack, and these threads can be placed within the heap, as per figure 2. Preventing the stacks overwriting parts of the heap is more difficult in this case. Programmers have to anticipate the maximum size of thread stacks on thread creation, to prevent stacks growing and overwriting parts of the heap.
This can be detected with mechanisms such as red zones [26] which use the MMU to make the end of each stack unwritable. In this case a fault mechanism can be triggered when the overwriting is likely to occur.

We have found it convenient to allocate the stacks in unmapped memory as per figure 3 to separate stack and heap, and to facilitate and simplify the migration of threads and heap data. Each thread starts off with a small number of pages, and as stacks grow downward they may generate memory faults and more pages can be allocated as desired. Thread stacks can be placed far enough apart in virtual memory to effectively solve overwriting problems.

![Virtual Thread memory layout employed by Nomad](image)

Other uses of Memory Management Units

The MMU can be used in a number of ways. Opal [5] uses the MMU to provide global, non-reusable address spaces. This is in opposition to the standard UNIX methodology (eg [8]) which uses the indirect addressing scheme so that all programs access the same address spaces. Text and stack start points are usually at the same virtual address for all UNIX processors. Each Opal program has a unique range of virtual addresses for its text, and each Opal process has a unique range of addresses for its stack and heap.

The Opal approach to using the MMU allows shared memory between processors to have a unique global address which avoids problems caused by different processes mapping shared segments to different virtual addresses. It also allows the thread pointers in the system to be persistent, avoiding swizzling used in systems such as Texas [23].

DSMs can be supported by page faulting mechanisms as described by Tan [29] and Ananthanarayanan [1]. MMUs support page faulting. UNIX processes which try to access memory which is not mapped into their address space cause an segmentation fault which can be caught by the UNIX signal mechanism [16], the appropriate page of memory can be fetched from its source, and the the process can be resumed when the data is available.

Thread Migration With NOMAD

Motivation

There is a clear requirement for a system which can run a single application across many machines on a network, and which can offers transparency of data, and also transparency of computation. The DSM model provides for a shared memory, but does not assist with the distribution of the processing on cooperating machines on a network.

There is a further requirement for a system which
can dynamically distribute processing over a network as the data distribution and processor load change.

Fetching a required page into the local address space when a fault occurs may not be the optimal strategy. It is frequently cheaper to migrate the requesting thread to the location of that page. This may be because

- Two threads on separate nodes may be requesting that page frequently, and migrating them both to the same processor will reduce thrashing.

- The mechanics of thread and page migration mean that that page migration requires two network transmissions - one request and one reply. Thread migration may only require one send. This is particularly useful on networks which have high transmission latencies.

- The current loads on the two machines may be better balanced by migrating the thread to the other node.

It is also desirable to make the fetch or migrate decision dynamically depending on the layout of the data, the accessing pattern, and the current load on the processors.

**NOMAD**

We have developed a runtime system, **NOMAD**, which provides transparency of data location as well as transparency of processing on a distributed memory machine. **NOMAD** assumes that

- the application is run in a number of threads
- multiple machines are available on a network to run the application
- each machine may have its own memory
- each machine has a MMU.

**NOMAD** offers a programming view of one large address space, which may be extant on an arbitrary number of nodes, and a thread mechanism in which a large number of threads can run. Attempts by threads to access data which is not available locally result in either the data being fetched or the thread being migrated to the location of the data. These operations happen dynamically, and below the level of the application programmer’s involvement.

The layout of memory in a **NOMAD** cluster is illustrated in figure 4. The layout described in figure 3 is split across a number of nodes. Threads run on these nodes, and access data as required. All data is divided into pages. Pages of data and threads migrate between nodes as required. The details are explained in the following sections.

![Figure 4: Distributed Nomad Memory Layout](image)

**Moving Pages**

The distributed memory in **NOMAD** is divided into pages. Pages can reside on any one of a number of participating nodes, and are mapped into the virtual address of that node. Data can be migrated between nodes by unmapping it from the virtual memory of the current node, copying it to another node, and mapping it into the appropriate virtual address.

Most variants of the UNIX operating system [3] implement calls `mmap`, `munmap` and `mprotect` [15].

5
which allow manipulation of the MMU. *mmap* can map pages into determined virtual address, while *munmap* unmaps them. *mprotect* changes the read, write and execute protection for these pages.

A page can be *imported* to the address space of the current node by requesting a page from the MMU at a specified address, and then filling it with the appropriate data (which may have been sent from another node). A page can be *exported* from the address space of the current process by sending the page’s contents to a remote node, and requesting that the MMU deallocate it. Clearly both of these operations must appear atomic to all program threads.

Execution on the individual nodes can proceed like a normal SMP architecture program until an attempt is made to access a page which is not held locally. At that point the offending thread is suspended, and a handler routine enabled. This is enabled by the UNIX signal [16] mechanism.

The signal handler mechanism requests the page from another node, installs it, and then continues execution of the suspended thread, which will reattempt the interrupted instruction, and access the required memory address.

**Figure 5: A Thread Attempts to Access a Non Local Heap Page**

Figure 5 depicts a thread accessing a non-resident heap page. In this case, the page can be moved from the node which has it, as depicted in figure 6.

**Figure 6: Heap page is Fetched to the Requesting Thread**

**Moving Threads**

The thread migration in the NO M A D system is designed to be very lightweight, and in principle involves only the migration of the top page of the thread’s stack, and the current register values to the new node. A thread can be created on the new node, primed with the registers, and then resumed. This can be done with a single send.

**Figure 7: Migrating a Thread To Access Data**

It is useful at this stage to examine the structure of the stacks employed by NO M A D. Figure 8 depicts activation frames and context information in a thread stack. For threads with a small nesting, and few automatic variables, the entire stack fits in one page, and hence the entire thread can
be migrated with the migration of a single page of memory.

Threads with multipage stacks are migrated by migrating the top and bottom pages of the stack, and this will include all of the stack and context for medium sized threads. Threads with larger stacks may have holes in them. That is, threads with many pages of stack may not have all those stack pages in the node that they are currently on.

A further optimization involves placing the context [14] information for the thread in the the highest address of the bottom page of the stack, reducing the data to be transmitted, and the work done on either node.

Figure 7 demonstrates migrating a thread instead of fetching the page.

Some complications occur when threads attempt to access pages other than their top of stack page, as depicted in figure 9. This may occur when frames on a thread’s stack straddle pages of memory or when procedures de-reference pointers to automatic variables of parent procedures.

Figure 9: Memory Access Violation In A Shrinking Stack

It can also occur when procedures exit, activation frames are popped, and the stack shrinks onto a previous stack page.

Figure 10: Missing Stack Page is Fetched

These can be resolved by fetching the missing page from its resident page as in figure 10 or by migrating the thread back to the node with the missing page as in figure 11. These strategies are not significantly different from fetching missing heap pages.

In fact if we are running a higher level language which cannot access activation frames other than the current one, then we are guaranteed that pages other than the top of stack page will be required only when popping frames (for procedure
movement (thrashing) which the page and thread have done, the memory capacity of a node, and the number of pages residing in that node. To date we have experimented with basing the decision on the time since the last migration of the data page, and the time since the last migration of the thread (trying to minimise both). Experiments comparing the merits of various policies are continuing.

### Implementation

We have implemented Nomad on a network of Sun SPARC machines running Solaris [27] on a LAN. The implementation runs as non-privileged processes on a number of Suns which communicate via the UDP protocol [20].

Coincidentally sending the top and bottom stack pages requires 8K of memory in Sun SPARC architectures, which is the same as the maximum UDP packet size, allowing an efficient implementation on this architecture. This makes the migration of threads in Nomad efficient and lightweight.

Verification packets must be used in a real network, and these are handled by the Nomad runtime system. These are done asynchronously, allowing restarting the threads as soon as possible, and therefore reducing the critical path time of the overall program.

Unfortunately on Solaris there is no call to make the MMU re-map memory from one virtual location to another, so extra measures have to be taken to make page export and import atomic. Before a page is exported it is made read-only, so that it cannot be changed as it is being written to the network. When a page is being imported, all threads in the process must be suspended to ensure that they do not read or write the page as it is being copied from the network. This overhead could be removed by adding a call to the system which allowed a memory page to be atomically mapped into another location, but this would involve kernel modification.

This MMU limitation also means that the cur-
rent implementation must copy data into a send buffer before it is sent. Direct manipulation of the MMU would also allow the top and bottom pages of a stack to be mapped into contiguous pages and transmitted together.

Language Implementations

The underlying model is very general, and allows implementation of many distributed languages.

The system has been used as a base for nSather, a transparently distributed version of pSather [19], which provides an SMP programming model, but runs on a distributed machine. nSather is an experimental language originally designed for experimenting with parallel language constructs. We have found it highly suitable for porting onto NOMAD.

We have also ported Java [2] onto the NOMAD system [17]. Our implementation translates Java bytecode [28] and to C code which makes calls into our own Java Virtual Machine which uses the NOMAD call library.

Concurrency has been designed into the Java language, and our implementation does not change any of the semantics of the language. The ease with which the Java port could be done after the nSather port suggests that the NOMAD model is highly portable, and therefore suitable for a large number of parallel languages and constructs.

Timing Tests

Timing tests were conducted on a network of Sun SPARC workstations running at 40MHz on a 10Mb Ethernet.

These results show that thread migration is faster than page fetching in the NOMAD system. Most of the time is believed to be latency time for the ethernet, and the networking layers of Solaris. Intimate area networks have much lower transmission latencies, eg the 77us for the AP3000 [24], and migration would clearly be many times faster on such a machine.

Discussion

The strategy we employ is normally non-invasive. That is, when no data or stack migration is necessary, then there is no overhead for the system. In the case where the data, thread and stack placement is optimum, the system will run at the same speed as a normal SMP implementation.

Ironically the non-invasive strategy makes monitoring of access to pages difficult when the access is local, because the underlying system is not involved. The system can easily count how many times a page has been fetched by a particular node, but cannot so easily count how many times it has been accessed in its current node. Clearly this compromises optimal bidding strategies. If this kind of monitoring is to be done, it must be done by a higher level mechanism in the run-time system, or in the future by a hardware mechanism.

Some CPUs, such as the Intel 386 architecture [30] employ an page access bit which is set when a page is accessed, to assist in paging to disk optimisation. The mechanism allows a least-recently used algorithm to page out data. This could provide the basis for further optimisation.

DSM systems by their nature are reactive systems, as opposed to pre-emptive systems. Pre-emptive systems have some programmed knowledge about which data is going to be accessed by which nodes, and these systems begin moving data before the data is required. In general we can expect these systems to perform better than

<table>
<thead>
<tr>
<th>Function</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Migration</td>
<td>8ms</td>
</tr>
<tr>
<td>Page Migration</td>
<td>15ms</td>
</tr>
</tbody>
</table>

Table 1: Timings for migrations for NOMAD on Sun SPARC Machines running at 40MHz on a 10Mb Ethernet.
reactive systems, but the DSM systems can improve their performance by improving their hit rates.

Many applications exhibit localisation of data access. That is, processes are likely to access memory close to memory which has been accessed recently. A thread may be accessing adjacent elements in an array, or several elements in the same record or object may be accessed in close succession. This factor increases the ratio of local versus remote accesses, i.e the hit rate.

Improving the hit rate may be achieved by:

- encouraging use of algorithms which use a higher data localisation
- using allocation algorithms which result in a higher data localisation
- discouraging use of algorithms which result in high numbers of threads sharing the same page.
- discouraging use of memory allocation algorithms which result in high numbers of threads sharing the same page.
- choosing good bidding policies for the nature of the particular application being run.

While we accept that these strategies compromise the stated goals of removing the distributed aspects of the programming from the user, an understanding of the underlying computing mechanisms is an advantage in most programming environments. For instance naive programmers do not need to be aware of paging aspects of virtual memory, but an awareness of the mechanism allows implementation of more efficient programs.

Our opinion is that object-orientation, and polymorphism in particular compromise static analysis of programs, as does the need to make programs event-driven in shared environments.

Work is continuing on optimising the communications layer of the software, and studying the requirements for good bid optimising strategies in environments where little is known in advance about the layout of the data, threads or processing bottlenecks.

**Related Research**

PVM [25] is a framework for parallel distributed computing on a network of workstations. MPVM [4] is an extension to PVM which allows migration of processes in a PVM environment. PM2 [11] is a system which claims to migrate lightweight threads, but details of the implementation are not given.

Other approaches include Cronk [6] in which lightweight threads have their own heap, and both heap and stack are migrated together. While this solves the problem of risking having to fetch data on the heap after the thread is installed, it has the disadvantage of limiting the amount of heap space for a thread, as well as making migration more expensive.

Millipede [7] is a system which migrates lightweight processes between nodes in a shared memory system. The Millipede model is similar to the Nomad model in that all nodes share the same address space, and pages are instantiated on a particular nodes as threads, and data migrate. It differs from the Nomad model in that migration of a thread involves migrating the whole of the thread’s stack, instead of limiting the data transfer in a migration to two pages.

**Future Work**

As yet it is not entirely clear what constitutes good bidding strategies. It is also not clear whether these strategies should be entirely dynamic, or based on a static program analysis.

**Conclusion**

We have presented Nomad, a system which attempts to solve the problem of distribution complexity, but not parallelisation complexity for the programmer. Just as high-level language pro-
grammers can program machines without a full understanding of machine instructions and CPU architecture, so too can a NOMAD programmer program without a full understanding of the distributed aspects of the underlying machine.

We provide a reactive system which doesn’t try to statically analyse the flow of data in the program, but takes a pragmatic run-time view, and attempts to place pages and threads according to dynamic measurements made at run-time. The system is non-invasive, which makes processing optimally fast when no network fetching is required, but also to some extent compromises placement strategies.

Further we have presented an ultra-lightweight thread migration mechanism, which offers allows dynamic thread migration as an alternative for page fetching even for short-term (or short-stay) remote memory accesses.

Real programming systems invariably involve trade-offs. As the field of computing has seen a relative shift cost from machine to programmer, so too have the programming languages seen more emphasis on programmability and reusability rather than machine efficiency. We expect that trend to continue, and see distribution transparency as a powerful tool in architectures of the future.

References


