Thread Migration with Active Threads

September 29, 1997

Michael Holtkamp
International Computer Science Institute

Table of content

Table of content 3
Acknowledgments 6
1.0 Introduction..... 7
2.0 Processes and Threads..... 9
2.1 Processes 9
2.1.1 A process definition 9
2.1.2 Handling more than one process 11
2.1.3 Operations on processes 14
2.1.4 Modes of processes 14
2.1.5 Disadvantages of processes 15
2.2 Threads 15
2.2.1 Thread definition 16
2.2.2 User threads and kernel thread 17
2.3 User Threads vs. Kernel threads 19
2.4 Threads vs. Processes 22
3.0 Process Migration 25
3.1 Process Migration Mechanisms 25
3.2 Systems with Process Migration 27
4.0 Thread Migration 28
4.1 Thread Migration Mechanism 28
4.2 The Heap Pointer Problem 29
4.3 The Stack Pointer Problem 31
4.3.1 Pointer Manipulation 32
4.3.2 Preventive stack reservation 34
4.3.3 The pros and cons of both methods 35
4.4 Systems using Thread Migration 36
5.0 Environment..... 38
5.1 Sather 38
5.2 pSather 39
5.3 The Active Thread Model 40
5.4 The Brahma Network Interface 42

5.5 Distributed Shared Memory (DSM) 44
5.5.1 pSather 45
5.5.2 Parallel C Environment 46
5.6 Global Synchronization Objects 47
6.0 Thread Migration with Active Threads 50
6.1 Chosen Migration Mechanism 50
6.2 Implementation 51
6.2.1 Start of the migration library 51
6.2.2 Creation of a thread 52
6.2.3 Termination of a thread 53
6.2.4 Migrating a running thread 54
6.2.5 Get a thread from another cluster 54
7.0 Load Balancing 56
7.1 Classification of Load Balancing Algorithms 57
7.1.1 Example Algorithms for the major Categories 58
7.2 Load Balancing with Thread Migration 59
7.2.1 Migrating Running Threads 59
7.2.2 Migrating threads from a remote run queue 60
7.2.3 Thread mechanisms classified 60
8.0 Performance Results and Examples 63
8.1 Active Threads 63
8.2 Communication Network 66
8.3 Thread Migration Primitives 67
8.4 More effective with Thread Migration 70
8.5 Load Balancing Example 72
9.0 Conclusions 79
References 80
Index 83

Thread Migration with Active Threads

Michael Holtkamp

Acknowledgments

I want to take the time to thank all the people who helped to establish this thesis.

The first one to mention is Jürgen Quittek. Thanks for all his tips and support and our discussions that helped me finding my way and thanks for the patience he always had with me.

Thanks to the Sather group at the ICSI for their support. A special thanks to Boris Weissman for building Active Threads that made this thesis possible and thanks for all our discussions.

Thanks to Prof. Otto Lange as the supervisor of this thesis at the Technical University of Hamburg-Harburg and thanks to Prof. Jerome Feldman for his invitation for my stay at the ICSI.

Last, actually the most important persons, thanks to my parents.

1.0 *Introduction*

Several models for parallel programming have been introduced in the past. The most popular one is the message passing model. The user distributes the application over multiple processing nodes that are connected over a network. The distributed parts of the application interact with each other by sending messages across the network. The user has to take care of the synchronization of all parts. He has to spent a lot of effort in partitioning the application and finding an efficient placement of the parts on the processing nodes.

One improvement to avoid the explicit message passing was the software supported virtual shared memory. Objects of an application can be shared among distributed processors and accessed by all of them. The accesses to shared objects are transformed into remote accesses transparent to the user. This helps the user to reduce explicit message passing. Still the placement of the application was a major problem.

Another step to ease parallel programming has been the invention of SMPs (Symmetric MultiProcessor Systems), multiprocessor machines with physical shared memory, along with a multithreaded programming model. The user can express parallelism with multiple threads. He does not have to care for the placement of the threads, because all threads are scheduled by the runtime system on one machine. Instead of explicitly sending messages between threads, threads can access objects in the physically shared memory.

For a long time no solution has been available to use several linked SMPs easily. Again, the user has to use explicit message passing or a virtual shared memory and has to take care of the placement of its application on the different SMPs.

This thesis shows that load balancing with thread migration can relax the placement problem. The idea is that all machines balance their load automatically even if the initial distribution of the application has been unfavorable. Together with a virtual shared memory the user gets tools for an easier parallel programming.

This thesis is organized as follows: In chapter 2 basic definitions of processes and threads are given and compared. Known mechanisms for process migration are summarized in chapter 3, while mechanisms and pitfalls using thread migration are discussed in chapter 4. As the environment for thread migration is important to choose the best migration mechanism, the environment for this work is shown in chapter 5. In chapter 6 the implementation of the migration mechanism are discussed and in chapter 7 performance results for applications using thread migration are shown. The work closes with some conclusions.

2.0 *Processes and Threads*

This chapter starts with the description of processes from which threads have been derived. After this, it explains why threads have been introduced and what the differences between threads and processes are. Finally, it shows the benefits of using threads instead of processes.

2.1 PROCESSES

2.1.1 *A process definition*

Operating systems (OS) offer users an environment to run their applications. One basic concepts of an OS is the one of the *process*.

A definition of a process is [Tan92]:

'A process is an instance of a program in execution.'

This definition includes two important statements. First, a program has to be executed to become a process. The program's code is useless, unless it becomes active on a computer.

Second, there can be several processes belonging to the same program. If one program is started concurrently several times, a new process will be created for each start.

The interface between a process and operating system is the *process control block* (PCB). The PCB is a data structure that is used to handle the process by the OS. The OS maintains a *process table* with one PCB entry per process.

The PCB contains information about the process including:

- the current state of the process
  See figure 3 on page 13.
- a unique identification of the process
- the process priority
- an area to save the registers of the processor
- pointers to allocated resources.
- pointers to the process memory

All addresses referenced by a running process are *virtual addresses* which are mapped to physical addresses by the memory management unit. From the point of view of a process there is one continuous address space, called the *virtual address space* or *virtual memory* (VM).

The virtual address space of a process is divided into the following [Hen96]:

- Code area
  The code of the program to be executed is stored in this area. The current position in the program's execution is marked by the program counter.
- Global data area
  The global data area is used for statically declared objects like global variables and constants.
- Heap
  The heap is used to allocate dynamic objects. Objects in the heap are usually accessed with pointers.
- Stack:
  The stack is used to allocate local variables, return values and return addresses of called functions. The stack grows and shrinks with function calls or their return. Objects on the stack are addressed relative to the stack pointer and are usually single variables.

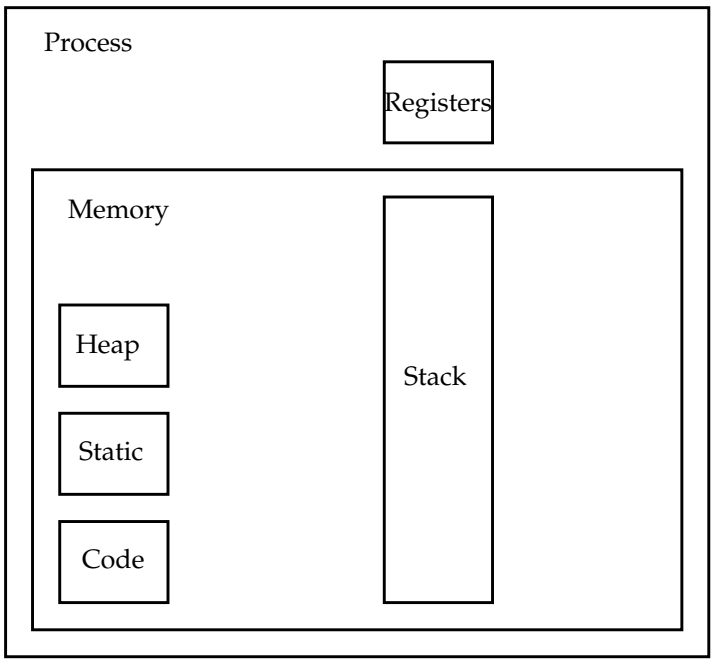

Figure 1: Process and its resources

The stack and the heap reside on opposite sides of the virtual address space. The stack can, for example in Sun Solaris, start at the higher addresses and grow down. The heap will then start at lower addresses and will grow up. During the process execution both run towards each other.

The *process context* describes a process completely. It consists of the corresponding PCB and the virtual address space. For processes currently being executed, also the content of the processor's registers are part of the context (*hardware context*). Registers are used to store variables that are often referenced like loop counters, the stack pointer or the program counter, because register accesses are very fast.

2.1.2 *Handling more than one process*

Modern OSs like offer a multitasking kernel. This kernel is able to handle multiple processes concurrently. In general there are two ways scheduling multiple processes, *preemptive* and *non preemptive.*

Nonpreemptive scheduling means that processes run to completion ones they got access to the CPU. The method is used in early batch systems like DOS and Windows 3.1. The contrasting strategy is called preemptive scheduling, which is used by UNIX systems and Windows NT.

Preemptive scheduling gives every process an amount of the processing power of the computer according to its priority and the overall number of processes. At a time only one process is running. But the scheduling algorithm gives every process access to the CPU during a larger time slice. So the user gets the impression that all processes are running concurrently.

Usually, the operating system puts one process on the CPU for one time slice. The process will run till the end of its time slice or till one of the following conditions occur:

- The process has to wait for a special event, e.g. an I/O to complete. Then the process is blocked.
- The process puts itself to sleep.
- The process terminates or it is killed.

A context switch has to take place to put the next process on the CPU. The context of the running process has to be saved, therefore the content of all registers are stored in the register save area of the process PCB.

After this, the register contents of the next scheduled process are placed on the processor's registers. The program counter is a special register that is restored, too, and so the processor knows which instruction to execute next.

If we neglect that processes can have different priorities[1], figure 2 shows the basic scheduling model.

The OS takes the next runnable process out of the run queue, if it schedules a new process. The currently running process is placed at the end of the run queue if it is not blocked. Otherwise, it has to wait for an event before it will be placed at the end of the run queue.

1. Otherwise, a process with a lower priority is only allowed to run, if there are no processes with higher priority. This can be modeled with different queues for each priority.

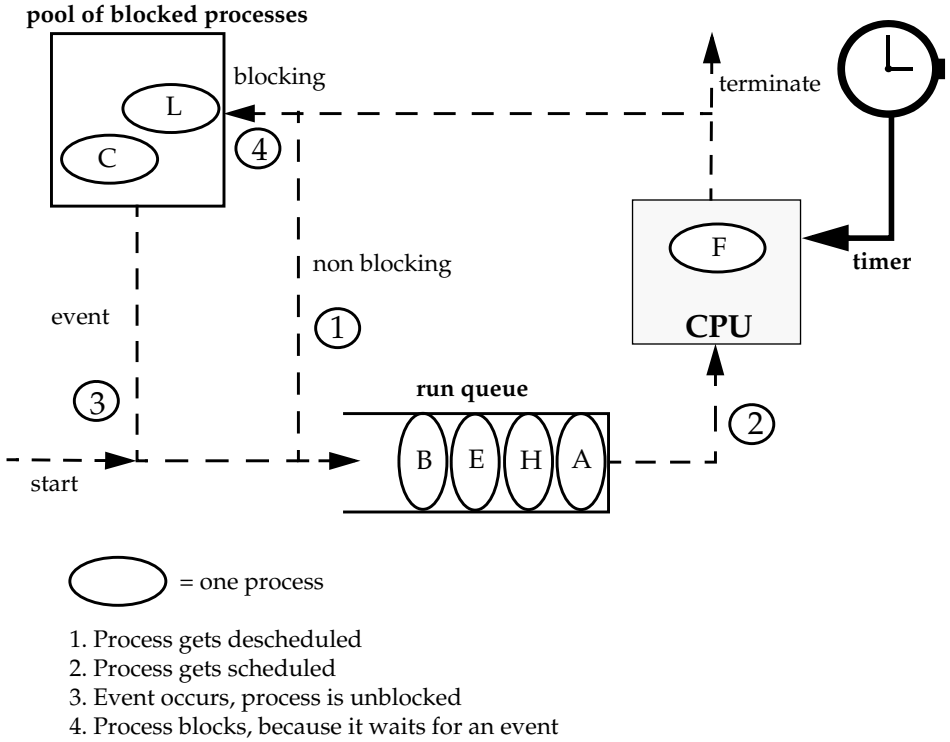

Figure 2: Preemptive scheduling of processes

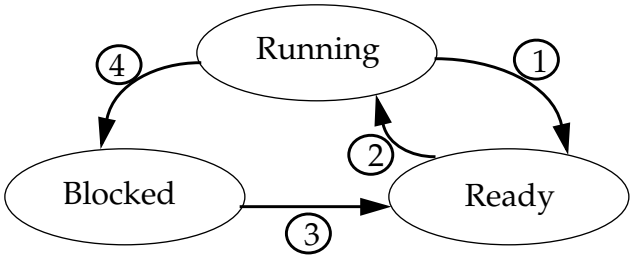

Figure 3: State diagram of a process, transitions are explained in figure 2

The state diagram of figure 3 shows the possible discrete states of a process and their transitions. The transitions are also marked in figure 2.

2.1.3 *Operations on processes*

The OS gives processes the environment for their execution. It can perform certain operations to manage processes. These include:

- create a process
- destroy a process
- schedule a process
- deschedule a process
- block a process
- unblock a process
- enable interprocess communication between two processes

2.1.4 *Modes of processes*

A process can run in two modes with different privileges: the kernel mode and the user mode.

These modes have hardware support on most CPUs. If a CPU is running in the kernel mode, all instructions are allowed. In user mode the CPU forbids some I/O and other instructions. The OS runs in the kernel mode of the CPU, while in general applications of the users run in the user mode. A special instruction switches the CPU from user mode to kernel mode. The context of the running process is saved and the CPU is placed in kernel mode. On the switch back, the saved context has to be restored and the execution can continue.

A process running in kernel mode has higher privileges than a process running in user mode. All services of the OS kernel are functions that are running in kernel mode. The kernel mode was designed to protect the OS from illegal changes of its data that would lead to crashes.

A process in user mode has no direct access to services of the kernel. If a process running in user mode wants to read data from disk, the process has to do a system call. A system call executes the trap instruction, that switches the machine from user mode to kernel mode. After this trap the process begins to execute a service function in kernel mode.

2.1.5 *Disadvantages of processes*

A process context consists of the virtual memory area (with code, static data, heap, stack) and a PCB (with all information about the process like tables for signals and file I/O).

Each of them is unique for the process. There is no sharing of resources between two processes except a special memory area used for communication and shared libraries. Even if one process spawns a child process (e.g. with *fork()*), the only part that is shared is the code.

Processes have a significant overhead when switching. During the switching the register content has to be saved and the context of the next process has to be restored. As the next process works on its own virtual memory, large parts of the physical memory might have to be swapped. This overhead makes task switching slow.

The only way to exchange messages between two processes is through pipes or shared memory. Every Interprocess Communication (IPC) involves a trap into the OS kernel which is also a context switch making it expensive.

One process can run on only one processor at a certain time. Although a program might include parts, that can run independently, it is not possible to run them in parallel on different processors within a process.

A similar problem occurs when one process is blocking. The OS takes away the processing power from this process. Even if there are other parts of the program, which could run independently from the blocking event, they can not continue their execution until the event occurs.

2.2 THREADS

In the previous section we have seen that processes have several disadvantages. There is no effective process switching and no possibility to use multiprocessor machines with single processes. This section introduces threads, which do not have these disadvantages.

2.2.1 *Thread definition*

In some literature threads are called *light weight processes* which gives an idea of what threads are. But this term has a special meaning in the Active Thread library that has been used for this work and will therefore not be used as thread definition in this thesis.

A more detailed definition is the following:

'Threads are multiple, independent, logical executable entities within a process, all sharing the process address space, yet owning unique resources with other threads within one process.'

The main difference to the previous sections, where only one executable entity (thread) was in one process, is that now several threads are running independently in one process. Each process can be seen as *virtual processor* of its threads.

The pendant to the PCB of processes for threads is the *thread control block* (TCB). It contains information about:

- the thread state
- a unique identification
- pointers to the thread memory

The thread usually owns only memory for its stack. But it can access parts of the virtual memory of the surrounding process which are shared between threads like the code area, the static data area and the heap. So two threads can work on the same variable if it resides in the process heap. Multiple threads within one process share the process resources (see figure 4). If one thread changes e.g. the working directory[2] of the file system, this change is visible to all other threads of the same process.

The *context* of a thread is given by its TCB, its stack, its register content and also by the shared parts of the context of the surrounding process.

2. The working directory is unique within a process.

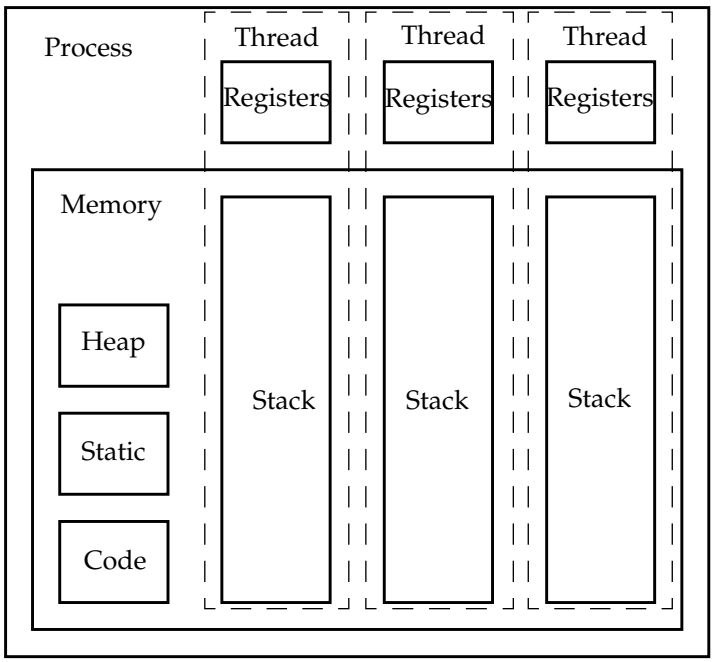

Figure 4: Threads and their resources within a process

2.2.2 *User threads and kernel thread*

Similar to the state of a process there are two different types of threads:

- user threads (user space threads)
- kernel threads (kernel space threads)

The difference between both is that kernel threads are scheduled and created directly by the OS while user threads are scheduled and created by the thread library in the user space (see figure 5).

User threads run completely in user space of the OS and are invisible to the OS kernel. To get access to the processor, user threads have to be associated to an entity in the kernel space, which is visible to the OS and can be scheduled by the OS.

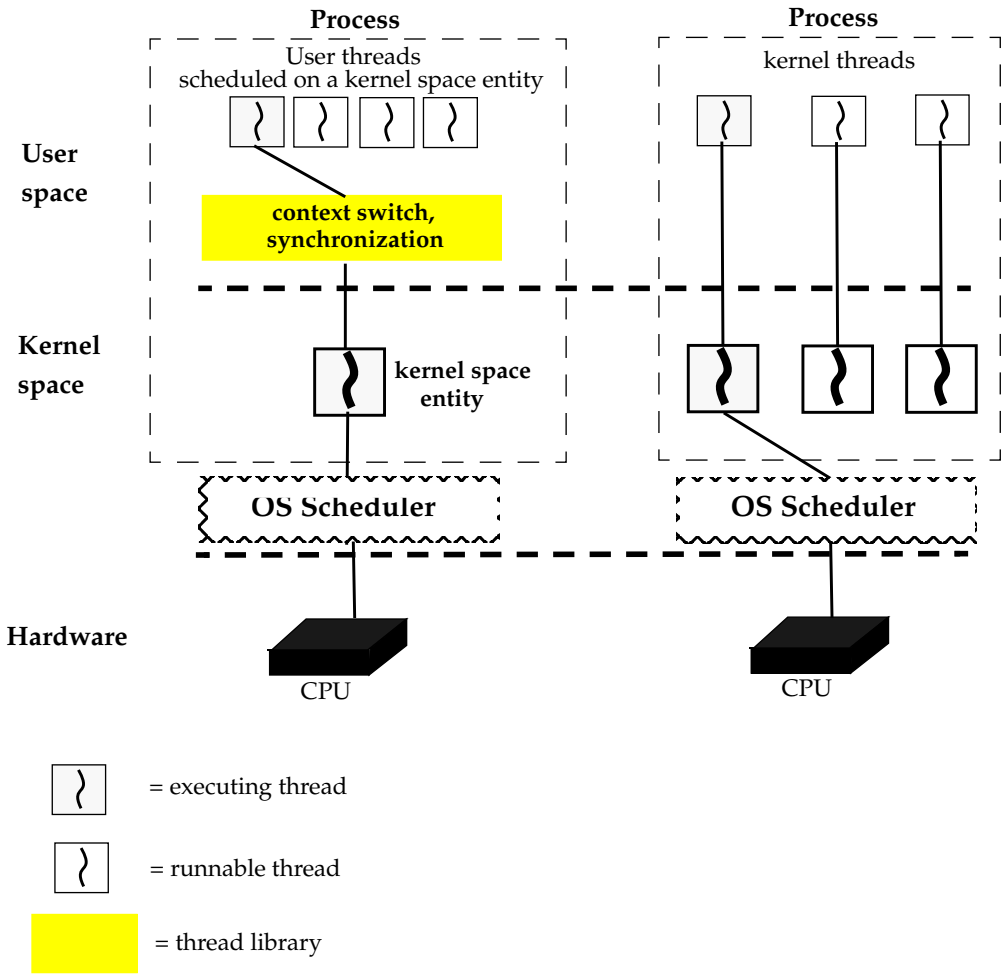

Figure 5: User threads and kernel threads

Kernel threads are implemented with direct access by the kernel. Each kernel thread has its own schedulable entry in the kernel space. So the OS can schedule each kernel thread within the timeslice of its process, but every context switch includes a trap into the kernel.

2.3 USER THREADS VS. KERNEL THREADS

All scheduling, creation and termination of user threads is done in the user space. For kernel threads each of these operations include a system call to trap into the kernel. This makes the execution of user threads usually faster than the one of kernel threads except for I/O intensive applications.

A disadvantage of user threads is that a switch from one thread to another can not be forced by a timer signal, because the thread library with its scheduler is running in user space and can not be woken up by hardware interrupts. User threads are non preemptive. It is only possible that each thread itself calls an explicit switch to another thread. Kernel space threads are scheduled by the OS kernel within the timeslice of their process. To schedule from one kernel thread to the next one the OS has to get control (e.g. by a timer interrupt).

A big problem for user threads are blocking system calls. If a user thread calls a service routine of the OS kernel which blocks (I/O is blocking, page fault), there is no possibility to switch back to the user space. As the OS kernel only sees the blocking kernel space entity of the process and does not see the entities in the user space, which are ready to run, it deschedules the whole process. Although most system calls can be wrapped in non blocking calls (e.g. doing polling), there are still some which are blocking (page faults). The additional wrapping is an overhead for system calls of user threads that reduces their performance.

If one kernel space thread blocks, it is still possible to switch to another thread of the same process, because each thread has a entity in the kernel space that is visible to the OS.

There is also no way of taking advantage of parallel machines like SMP's (Symmetric MultiProcessor Systems) with multiple user space threads. As the OS kernel only sees one schedulable entity for one process, the OS kernel schedules this entity on one processor. Using kernel threads there are many visible entities for the OS kernel for one process. So different threads from one process can be scheduled on different processors.

User thread packages are very flexible and portable. They can be implemented on top of an existing OS that does not support threads. The OS has

the view of a single threaded process and all managing of the user threads is done by the thread library. But this makes them very flexible. It is easy to implement several scheduling algorithms in the thread library in order to customize user threads for special applications.

Table 1 summarizes the advantages and disadvantages of user and kernel threads.

Table 1: Users threads compared to kernel threads

	User threads	**Kernel threads**
Switching, Creation, Termination	+ completely in the user space	- thread has to trap into the OS kernel
Preemption	- user threads can not be preempted by the user space thread library	+ kernel threads can be preempted by the OS
Blocking calls	- if a system call blocks, the whole process will be descheduled	+ if a system call blocks, the next thread of the same process will scheduled
System calls	- to avoid blocking, system calls have to be wrapped	+ no wrapping needed
Multi processors machines	- no direct use of multiprocessor machines	+ different threads of one process can run in parallel on several processors
Flexible scheduling	+ user defined scheduler can be used	- only the scheduler of the OS can be used
Portability	+ user threads can be built on top of many existing OS	- kernel threads have to be supported by the OS

2.4 THREADS VS. PROCESSES

Each process has its own full virtual address space and operating system state. Two different processes do not share their resources (exceptions are shown in section 2.1.5). In contrast, two threads running in the same process share the resources of the process like the address space and open files.

The resources of a process are shown in figure 1 on page 11. Figure 4 on page 17 shows the resources of three threads within one process. These threads share most of the process resources with each other. Only some parts are unique for each of the threads.

This leads to a significant performance increase while using threads. To create a process, the memory for its virtual memory has to be reserved and the code has to be loaded. A thread is started in a process. So it can use the process virtual memory and the already loaded code. It just has to allocate memory for its stack.

The context switches of threads can also be implemented significantly faster. Two threads of the same process share the process resources. During a context switch between two threads the context of the surrounding process remains. The running thread has to save the register content on its stack and the register content of the next thread has to be restored. When two processes are switched, usually a large amount of memory has to be swapped, because each process has its unique virtual memory.

Processes communicate through pipes or special shared memory areas. These communications are system calls. Threads can use shared variables in the process memory which are accessible without OS protection.

Lets imagine a factory as an analogy for an process. In terms of the traditional single threaded process we have one employee working in the factory. He or she has to do all tasks and to use all tools to assemble products. If we have several employees (threads), they can share the tools and tasks. This would reduce the time to assemble a single product and the factory would have a higher output.

Communication within the factory (thread communication) is easy and fast, the employees can talk directly to each other. Interactions with another fac-

tory (interprocess communication) are not that easy, a telephone has to be used or a letter has to be sent.

Creation of a new factory (process creation) takes a long time, but hiring a new employee (thread creation) would be simple. An employee can easily do the task of another employee (context switch between threads). Changing the product line of the factory (context switch between processes) would obviously be a big effort. The tools for the old product line have to be dismantled and other tools have to be put up.

Concluding the comparison between threads and processes, an overview about the advantages and disadvantages of threads and processes is given in Table 2.

Table 2: Processes compared with threads

	Process	**Thread**
Creation	- allocate memory in the virtual address space - load code	+ allocate memory for the stack + code exists already
Switching	- includes loading and storing of all process' tables - large amount of memory might have to be swapped out	+ little context has to be saved + only the thread stack has to be swapped
Communication	- through pipes and shared memory with OS support	+ through shared memory of the process without OS support
Synchronization	+ implicitly by pipe communication	- explicitly by semaphores or other synchronization objects
Multiprocessors	- one single threaded process can run on only one processor	+ multiple threads in one process enable the use of multiprocessor machines
Application responsiveness	- applications appear unresponsive to the user while performing other functions (e.g. WWW browser Mosaic)	+ applications create separate threads, so that they have greater responsiveness (e.g. WWW browser Netscape)

3.0 *Process Migration*

As processes have a longer history than threads, the first attempts of dynamic load balancing have been done with processes. This chapter wants to give an overview over the basic methods, that are used.

3.1 PROCESS MIGRATION MECHANISMS

In the following the term *cluster* will be used often. A cluster is a set of one or more processors that share physical memory. With this definition, a Personal Computer, a workstation or a multiprocessor SMP will be called a cluster.

Process migration can be described as the act of moving (migrating) a process from one cluster to an other.

There are several problems that have to be solved to enable process migration. One problem is the process ID. The ID of the process, that should be migrated, could already be used on the destination cluster. Consequently a process migration system has to provide a logical process ID for all user operations requiring an ID. This ID maybe different from the one used by the OS.

If the OS uses a nonpreempitve scheduling strategy, the running process itself can only start its migration, because the process can not be interrupted by the OS. Therefore the process has explicitly call a function that invokes the migration.

It should be assumed that both the source and the destination cluster use a compatible processor architecture, so that the code can be used after migration without a transformation.

Basically, migrating a process from the source cluster to the destination cluster is done in the following steps:

1. Deschedule process and save its context.[3]
2. Copy the process context to the destination cluster.
3. Resume the process' execution on the destination cluster.

To restart a process, its complete context has to be copied. For processes the context includes the virtual memory (VM). So in a straightforward way all parts of the VM, that have been allocated, have to be copied after suspending the process.

This may become very inefficient, because the allocated parts of the VM are usually much larger than the rest of the context (the PCB). Also, not all parts of the VM will be referenced anymore. So the dominating cost of the process migration is the time spent in transferring the VM.

To enhance the performance of the straightforward process migration different schemes to transfer the VM are used:

- Pre copying of the virtual memory (interleaving step 1 and 2):
  Before the process is suspended on the source processor its complete VM is copied to the destination cluster. During the transfer some pages of the VM may have been changed. So these *dirty pages* are copied until a certain threshold of remaining pages is reached. Then the process is suspended and the current dirty pages are transferred.
- Copying on reference (interleaving steps 2 and 3):
  In this scheme the context of the process is copied without the VM. After this, the process is restarted on the destination cluster. Every time the process references a page, that does not reside in its memory, this page is copied from the source cluster. This scheme takes care of the fact that not all pages of the VM are actually used and copies only a minimum of the context.

3. If the process is running. Otherwise the context is already saved.

The pre copying scheme wants to reduce the freeze time during migration. The process can continue its execution during the transfer of the VM. Theimer [The86] has shown this reduction. The main disadvantage is that pages may be transferred more than one time and that unneeded pages are copied.

It has been shown that the copying on reference scheme reduces the cost of migration significantly. Zayas claims that the number of bytes exchanged between clusters have dropped by 58.2% and the cost of message handling has dropped by 47.8% on average using copy on reference strategies instead of the straight forward way [Zay87].

3.2 SYSTEMS WITH PROCESS MIGRATION

There are many existing distributed systems that have process migration facilities. The straightforward VM copying scheme is used by older systems like CHARLOTTE [Art89]. The V DISTRIBUTED SYSTEM [The86] copies the VM before the migration. The copying on reference scheme is used by ACCENT [Zay87] and SPRITE [Dou87].

4.0 *Thread Migration*

This chapter describes migration of threads and the known problems with it. Problems occur, because only a part of the thread context - the part that is not shared with other threads - is copied to the destination cluster during the thread migration. Some solutions to solve these problems will be shown.

4.1 THREAD MIGRATION MECHANISM

Some of the problems, that might occur with thread migration, are the same as with process migration (see section 3.1). The thread migration system has to provide a logical thread ID, because a thread with the same ID as the thread, that shall be migrated, might already be running on the destination machine.

If non preemptive threads are used, the migration can not be invoked by an external event. Therefore running threads can only be migrated by themselves. Threads of the run queue can be migrated without this restriction.

Another problem is that the thread has to be restarted in a different process after its migration to the source cluster. Therefore it is assumed that a process for the same program has been started on the source and the destination cluster and that the code is in the same virtual memory area on both clusters.

The basic steps to migrate a thread are the same as in the previous chapter:

1. Deschedule the thread and save its context.[4]
2. Copy the unique part of the thread context (stack and TCB) to the desti-

nation cluster.
3. Resume the thread execution on the destination cluster.

In contrast to process migration where copying the process context includes its VM, thread migration requires only the thread stack and its TCB to be copied. All shared resources of the process remain on the source cluster. As one result, a part of the thread context will still reside on the source cluster. Therefore a migrated thread can only continue the execution, if it does not access the shared resources or if it can still access the resources (e.g through a virtual shared memory (VSM), also called distributed shared memory (DSM[5])).

Another problem is that the thread stack might not be at the same memory location after its migration. This might be the case if the number of running threads on both clusters is different and if the threads allocate memory for their stacks when they are created.

To see what this means to thread migration, the following sections take a closer look at references in the thread stack.

4.2 THE HEAP POINTER PROBLEM

After copying the thread stack to the destination cluster, pointers referencing data in the process heap are in general no longer valid. Instead of pointing to the process heap on the source cluster they point to the process heap on the destination cluster.

The problem can be seen in figure 6a and figure 6b. Both figures show the virtual memory areas on the source and the destination cluster before and after the migration of a thread. The pointer referencing data in the heap points to an invalid memory location after copying the stack.

4. If the thread is currently running. Otherwise the context is already saved.
5. A distributed shared memory (DSM) is a memory area that is accessible by all clusters.

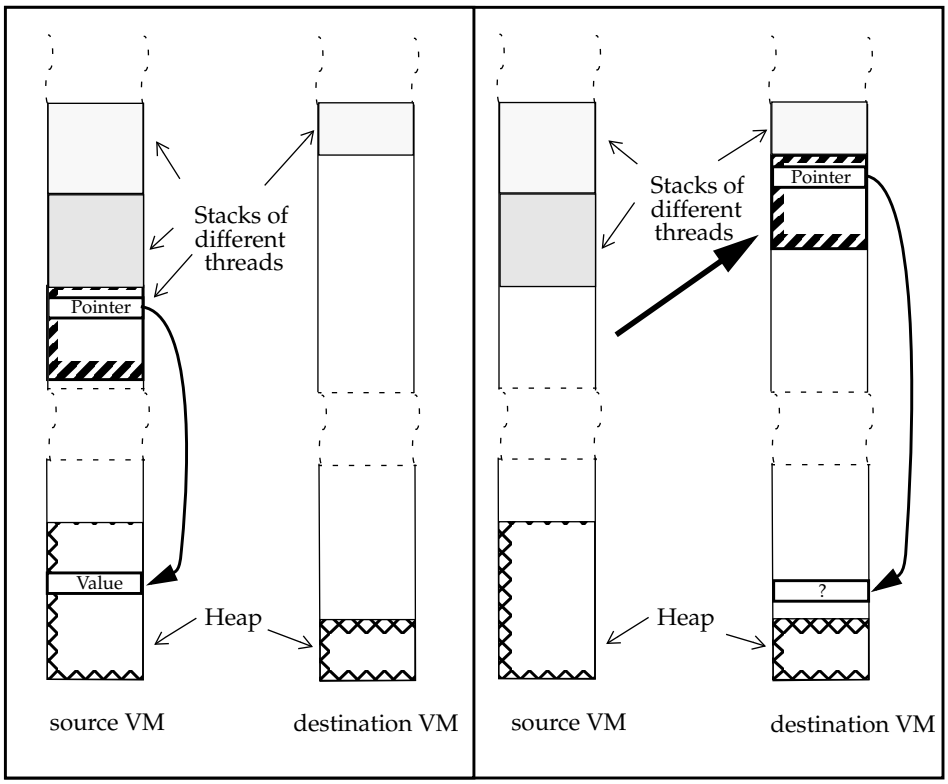

Figure 6a: Process VM before thread migration

Figure 6b: After migration

The pointer should actually reference the same data in the heap of the source cluster. To solve this problem, one can forbid to use the heap, but this is not acceptable for general programming. Another solution is to provide a virtual common heap for all clusters by a DSM/VSM. In this case every reference to an object at a remote cluster is replaced transparently by a remote access. The drawback of this solution is that the remote access might be an expensive operation and that every heap access operation has to be checked to distinguish local and remote accesses.

4.3 THE STACK POINTER PROBLEM

A similar problem as in Section 4.2 may occur to pointers in the thread stack referencing data in the stack itself. In general, it is not guaranteed that the stack base resides at the same address on the destination cluster after copying the stack. So these pointers become invalid.

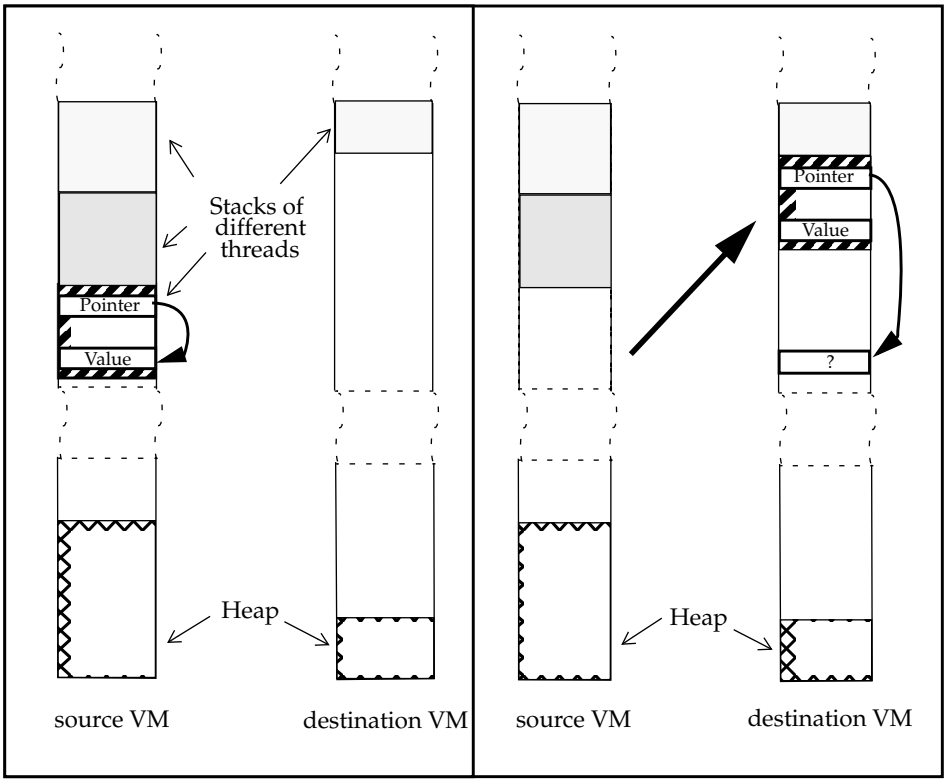

Figure 7a: Process VM before thread migration

Figure 7b: After migration

See figure 7a and figure 7b. After copying the thread stack the reference does not point to the object in the stack.

While the heap pointer problem can be solved with a DSM, this is not a suitable solution for this problem, because the memory of stack on the source cluster can not be released and can not be reused. But more important is the fact that all local variables of the thread reside on the stack. So every access of a variable would include the overhead of a communication between

source and destination cluster. This is not acceptable, because the most referenced variables in general programming are local variables, so that the performance of the application might be reduced significantly.

In the following sections show two possible solutions are shown and their advantages and disadvantages are discussed.

4.3.1 *Pointer Manipulation*

One way to get out of the trouble is to update all pointers in the stack after it has been copied to the destination cluster. The stack base addresses on the source and the destination cluster are known. So by adding an offset given by the difference of the destination base address and the source base address, the pointers remain valid on the destination cluster. Figure 8 explains this solution.

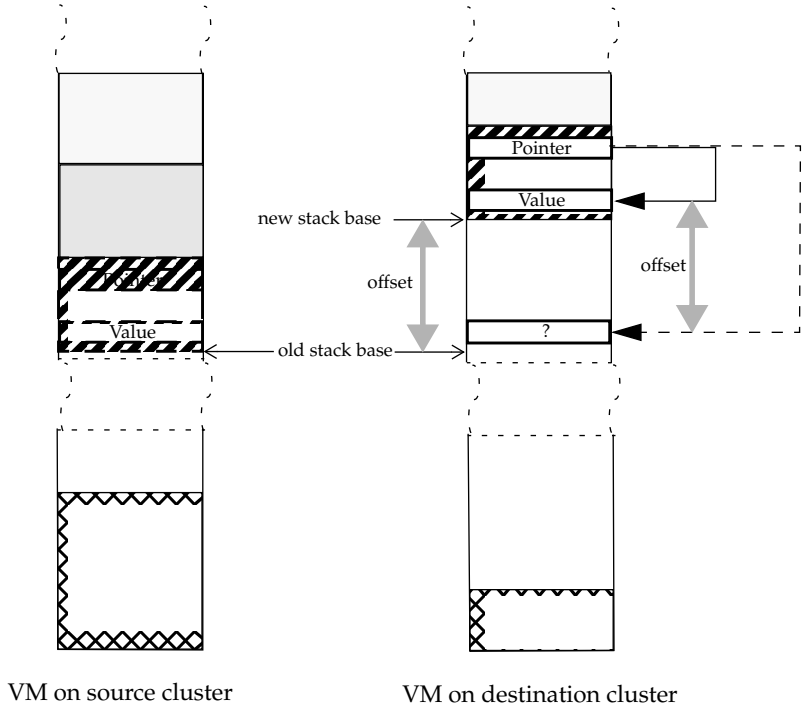

Figure 8: After manipulation

The main problem of this solution is to find all pointers in the stack. It can not been seen from the value of a memory cell, if it contains a stack pointer, a heap pointer or a value variable.

Image the stack given in figure 9. There are different possibilities how this stack could have been produced. The stack entry at address 10008 can be a pointer referencing the data at address 10004 or it can be a variable with the value 10004.

Although both of the above possibilities produce the same stack entries, both stacks have to be treated differently. In the first case, the stack entry at address 10008 has to be manipulated in the way described above, but in the second case, all entries remain unchanged when migrated.

As the stack itself does not include enough information about which entry has to be changed, additional information is needed. The only way to get along, is to know the memory locations of each pointer.

Cronk et al. introduce one solution [Cro97]. They built a runtime library that can run on top of existing languages. Their way is to register every new initialized pointer in a list. This list entry has to be released again when the pointer goes out of scope.

Once they have a list of all pointers, it is possible to update the stack after copying it to the destination cluster. They just have to check for all elements of the list, if the value of the pointer is in the range of the stack before the

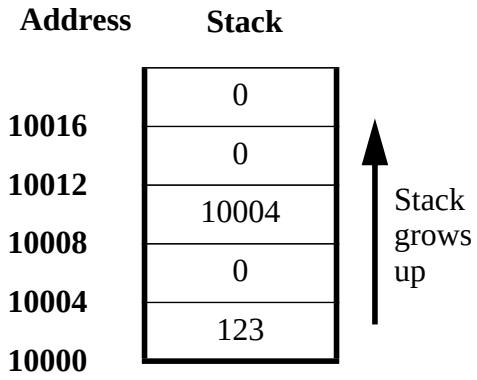

Figure 9: Given stack

migration. If this is the case, the pointer will be updated, otherwise it will remain unchanged.

Another problem that can not be solved with these lists are pointers, that are in registers. During the saving of the thread context before migrating, the register content can not simply be pushed on top of the stack. If this happened, the pointers would be loaded unchanged into the registers of the destination cluster while restarting the thread. The pointers may become invalid, because there is no entry in the list for pointers in the saved register content.

To change pointers in registers, additional information from the compiler is needed. If the pointer manipulation mechanism is used and if this information is not available, pointers can not be allowed to be in registers.

4.3.2 *Preventive stack reservation*

In the previous section we changed all pointers, that became invalid after migrating a thread. Now we will try the other direction. Instead of manipulating the pointers, the same pointer shall be valid at the source and the destination cluster. If this is guaranteed all pointers can remain unchanged.

For the previous solution an offset has been added to stack referencing pointers. All pointers are unchanged, if the offset is zero, which means that the stack base is the same at the source and the destination cluster.

In order to achieve that all pointers retain their meaning after the migration, the stack base should not change. The copying of the stack in this scenario is shown in figure 10a and figure 10b. After the migration the pointer references the right data item.

One way to achieve this, is to reserve a predefined memory area for every thread at all clusters.

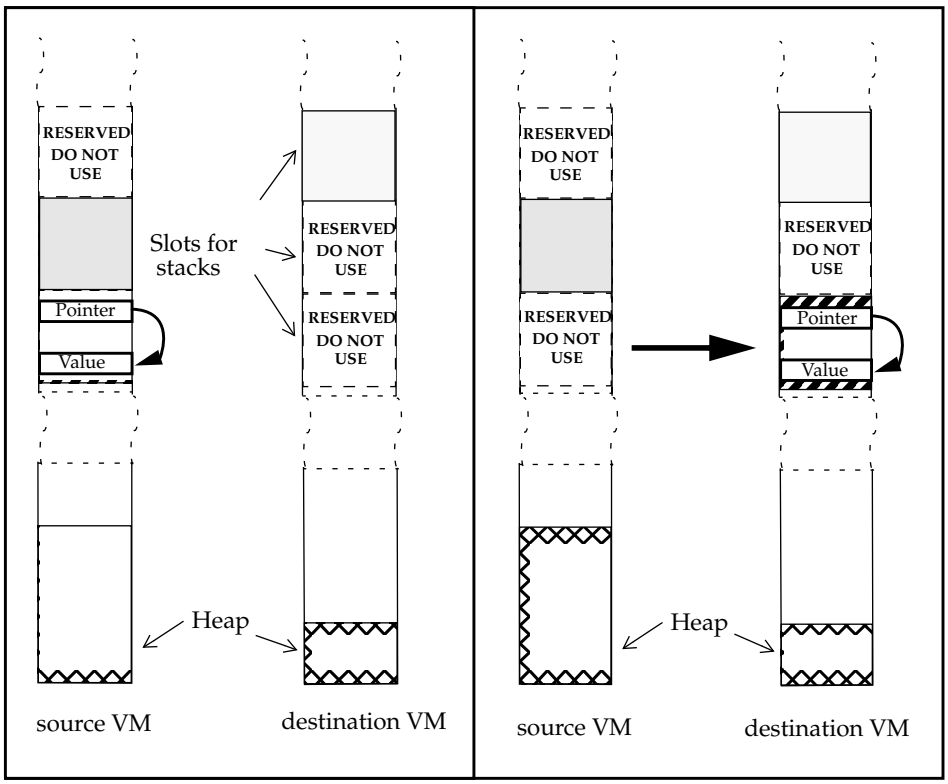

Figure 10a: Thread migration with preventive stack allocation

Figure 10b: After migration

4.3.3 *The pros and cons of both methods*

Both approaches introduced above have advantages and disadvantages. To decide which of them is the suitable solution, depends on the system used.

The pointer manipulating method is only possible with an extreme support from the compiler. One has to know all locations of pointers both in the stack and in registers. If this information is not available, some pointers may become invalid after the migration.

There is also a performance disadvantage, because of the overhead for managing lists with all pointer locations using the proposal of [Cro97]. There-

fore construction and destruction of pointers is significantly slower. It also takes more time to migrate a thread. After copying the stack, every pointer has to be checked, if it has to be updated.

The stack reservation approach does not have this performance overhead. It allows a faster thread execution and a faster migration. There is no need to know all pointers, so it is not necessary to have access to compiler internals.

Its main disadvantage is that the overall number of threads has a stricter limitation. A piece of the VM for every thread has to be reserved on all clusters. This memory slot can only be used by the thread, that has the corresponding logical thread ID. The slot has to be reserved on all clusters, but the slot will only be used on one cluster at a given time. Regardless how many clusters are used, the maximum number of threads, that can run on all clusters concurrently, is the same and equals:

$$maxthreads = \frac{maxmemory}{stacksize}$$

So this does not scale with an increasing number of clusters, but since the size of the VM usually is large enough, this restriction applies in rare cases only.

4.4 Systems using Thread Migration

There is a lot of research going on about thread migration. Most of the existing systems work with the pre-reservation of the stack memory, because these systems can get along without compiler support.

The following system update all pointers after thread migration:

- EMERALD:
  Emerald [Jul88] is a distributed object oriented language and runtime system. It has been one of the first systems using migration. Every object in Emerald contains a data object and a thread handling all operation on the data object. So moving one object includes the migration of the thread. As Emerald has complete control of the programming language, it is possible to update all pointers after the migration.

- ARIADNE:
  Ariadne [Mas95],[Mas96] is a user space thread library for parallel and distributed clusters. It provides a user level function to update pointers. No details are given how Ariadne identifies the stack references. Pointer in register are not updated, so the use of register for pointers is prohibited.

The more common approach is to reserve the same memory area for all threads on all clusters. Systems using this method are:

- AMBER:
  Amber [Cha89] is an object oriented DSM system. Objects in Amber consist similar to Emerald of a data object and a thread that handles the operations on the data object. Amber occupies the same address for each object and thread on all clusters to achieve the possibility to migrate threads and complex objects. It uses a so called global address space, in which references of objects have the same meaning, regardless to which cluster they are migrated.
- MILLIPEDE:
  Millipede [Itz97] is an environment for parallel programming over distributed clusters running under Windows NT. Millipede uses kernel threads although the migration mechanism is implemented in the user space. Millipede provides a DSM, so that references to the heap are valid after the migration of a thread.
- UPVM:
  UPVM [Cas94] is a thread package that supports migration for PVM (Parallel Virtual Machine) applications. It uses User Level Processes (ULPs) which similar to user threads. Additionally to the thread stack, UPVM also copies a private thread heap. So space for the stack and the heap has to be reserved on all clusters.

5.0 *Environment*

We have seen that choosing the migration mechanism depends on the system used. So this chapter will give a closer view of the thread and network library that has been used in this work.

This work for this thesis has been done in the Sather group of the ICSI. One goal was to built a thread migration mechanism for the object oriented programming language Sather and its parallel extension pSather.

Another goal was to extend the thread library with a thread migration mechanism to get an parallel C extension.

5.1 SATHER

Sather is a modern object-oriented programming language designed to be safe and efficient. Some of its major features are strong static type check, dynamic dispatch, multiple inheritance, parametrized classes, iteration abstraction, garbage collection, and exception handling.

Sather compiles to C. A compilation of a Sather program to an executable is divided into two steps (see figure 11):

1. Compilation of the Sather program. The output is C code.
2. Compilation of the C code with a C compiler. Linking with the C runtime libraries.

The advantage to use a two - step compilation is the resulting portability. Sather is currently available for several UNIX platforms and Windows NT.

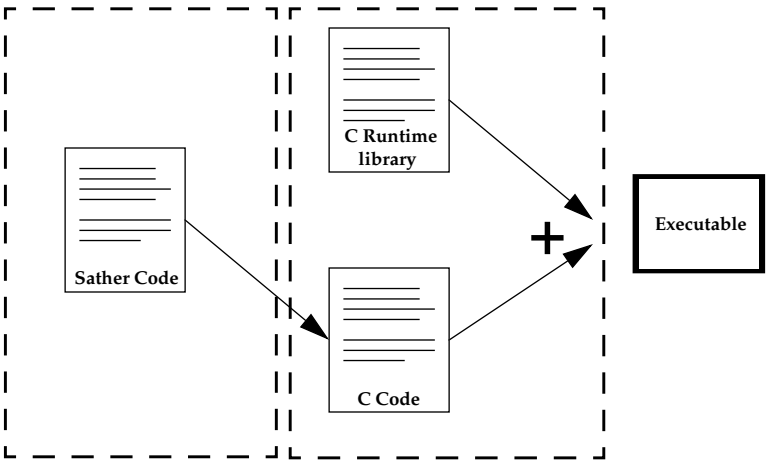

Figure 11: Compilation of a Sather program

Most of the Sather runtime system like the thread library are written in C. Therefore the thread migration support has also been written in C as an extension of the thread library. Therefore this extended library can also be used as parallel C extension.

5.2 pSATHER

Besides the serial language the Sather specification describes a parallel extension called pSather. In pSather the model of threads is used to express concurrency. This is different from pure object-oriented approaches to concurrency, e.g. the actor model [Agh86], which has been implemented by languages like Emerald [Jul88]. The thread concept being orthogonal to the object-oriented concept was chosen to gain efficiency. It can be seen as a compromise between a clean design and high performance, of which many pure object-oriented approaches are showing a lack. pSather also differs from languages like Java, where threads are represented as first-class (user-managed) objects. Instead, pSather threads are system-managed and handled by explicit language constructs for thread creation and synchronization. This choice is believed to lead to a more clear program structure and a better readability of code than user-managed threads.

Another compromise was made in the design of the memory model. To support a clear and efficient way of implementing parallel algorithms, an abstract shared memory model is offered, but in order to achieve high performance the mapping of objects and threads to clusters of a distributed platform is done explicitly by the programmer. A current approach to abstract from explicit mapping is the concept of zones [Sto97] allowing the programmer to express locality of objects and threads without explicitly placing them.

pSather is not as portable as the serial part of Sather, since for many operation systems threads are not available and a standard interface to threads and synchronization like the POSIX 1003.1c 1995 thread standard is not implemented completely for all systems claiming to support it. Currently, implementations of the pSather extension exist for shared memory and distributed memory platforms running the Solaris operating system (including Intel x86 platforms). These implementations are built on top of either Solaris threads or Active Threads (see Section 5.3), a portable thread package developed by Boris Weissman at the International Computer Science Institute in Berkeley.

5.3 THE ACTIVE THREAD MODEL

Originally pSather used the Sun Solaris thread library. But this library offers insufficient performance. An estimation of the costs of all thr ead operations leads to the result that they can be implemented much more efficiently . Another problem was that Solaris threads are not ported to other platforms except Sun Solaris. This resulted that pSather was only available for Sun Solaris systems. Therefore the Active Thread library was created to achieve better performance (see also section 8.1 for performance measurements) and to be portable (Active Threads are currently ported to SPARC, Intel 386 and higher, DEC Alpha AXP and HPPA systems).

The Active thread library (figure 12) uses so called *lightweight processes* (LWP), that are kernel threads according to section 2.2.2. These threads are the schedulable entity in the OS kernel and they are the underlying thread layer in the kernel space. The LWPs are scheduled by the OS. On top of this

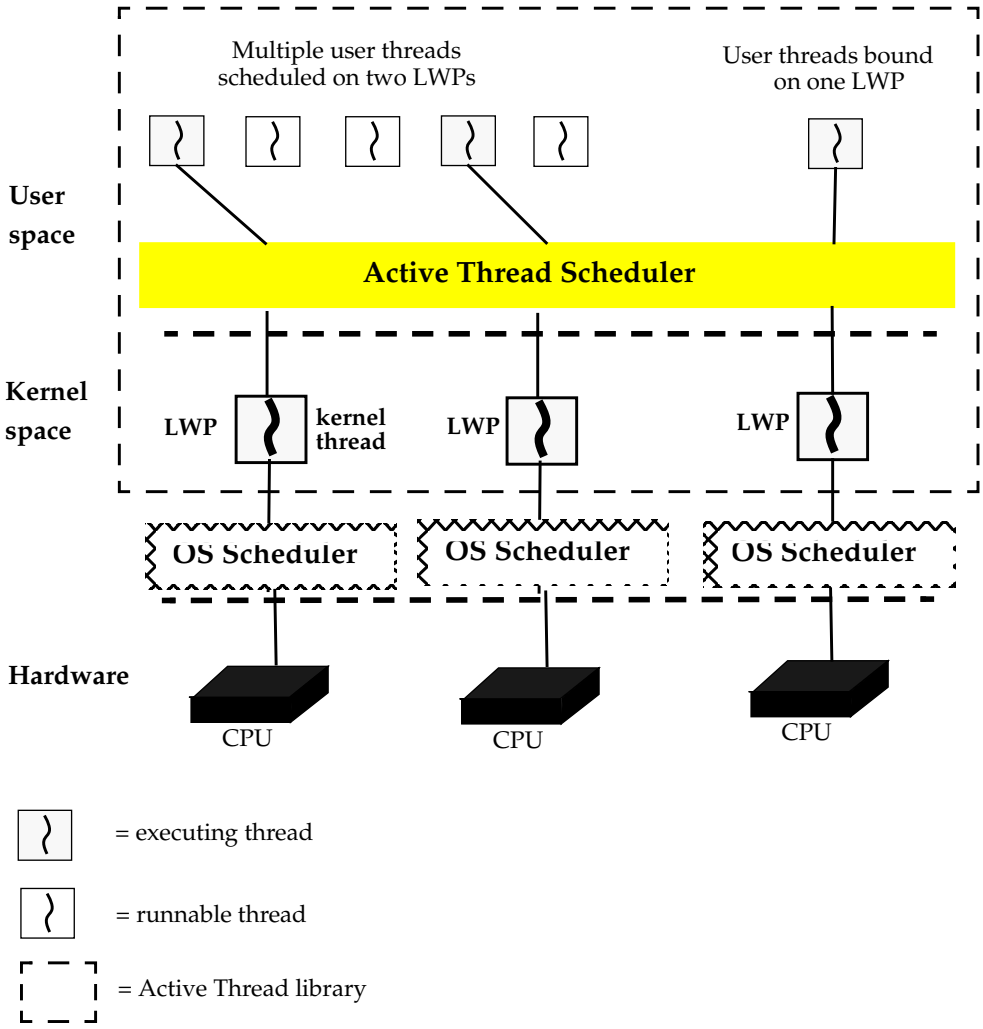

Figure 12: The Active Thread Model on a SMP

underlying layer is the runtime system, that schedules the runnable user threads of a queue (called *run queue*) within one LWP. The scheduling model is called the *Two-Level Model,* because a thread can be bound to a LWP or any number of threads are multiplexed onto some (smaller or equal) number of LWPs. The Active Thread library uses one LWP per available processor to enable the use of SMPs. No more than one LWP per processor is used to avoid unnecessary scheduling of the LWPs in the kernel space.

In this model one can think of LWPs as virtual processors for user threads. LWPs bridge the user space and the kernel space. All user threads are scheduled on top of a kernel thread.

Runnable user threads are stored in different *run queues*, that are used by the scheduler to get the next runnable thread. There are queues for bundles of semantically related threads (*bundle queues*), a queue for each LWP to achieve better spatial locality and to bound threads to LWPs (*local run queues*) and one global queue (*global run queue*), where unbound and unrelated threads are stored. Depending on the scheduling policy the different run queues are used.

The Active Thread library offers an interfaces to:

- create a thread,
- terminate a thread,
- stop the thread execution,
- bundle semantically related threads to share the same scheduling policy,
- synchronize threads with various synchronization objects like blocking and non blocking mutual exclusion locks and semaphores.

5.4 The Brahma Network Interface

The Brahma network interface is based on the Active Message model developed at the University of California at Berkeley. The interface has been extended at the ICSI and provides portable mechanisms for threads, synchronization and active messages.

The Brahma library includes function calls to:

- start threads on any other cluster of the network,
- copy parts of the memory between two cluster (copying can be both synchronous or asynchronous),
- synchronize threads on one cluster or on the complete network,
- exchange active messages between two clusters.

As Brahma is built on top of an active message layer, it is important to understand the active message model. An active message activates a request handler function on the cluster, that receives the message. Optionally the receiver can send a reply message, which invokes a reply handler function on the source cluster. The complete communication circle starts with the sending of a request handler function from the source cluster to the destination cluster and ends with the invocation of a reply handler function on the source cluster.

An active message contains of an address to the handler function and some data. The sending thread continues its computation after sending a message. The message will first be written into the output queue of the source cluster. It will then be transmitted across the network and will finally be written in the input buffer on the receiving cluster (see figure 13).

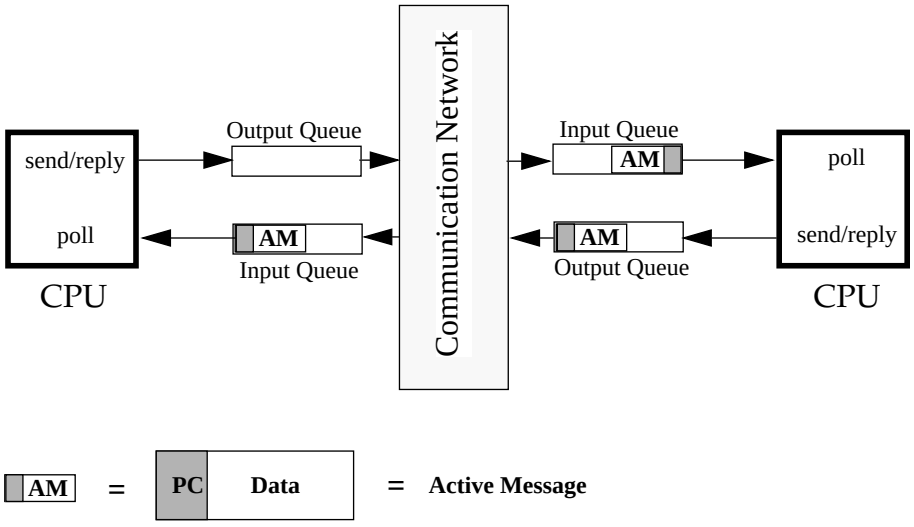

Figure 13: Active message model

The handler function will be invoked when an executing thread on the receiving cluster checks for waiting requests in the input queue. The thread starts the function with its arguments at the given address. After finishing the execution of the handler, the thread returns to the interrupted computation. Communication is therefore only guaranteed when input queue is checked frequently. Long running programs have to call explicitly a *BR_POLL()* from time to time, that checks the input queue. The input

queue is checked automatically during context switches and synchronizations.

A request handler function can explicitly call a reply function on the source cluster. The function invocation mechanism is the same as with a request handler. If no reply function is called explicitly, a simple acknowledge function is sent to the source cluster. An acknowledge has to be sent, because the size of the input queue is limited. If the queue is full, sending the next request handler will lead to a loss of an active message. Therefore the number of outstanding reply messages is counted. It will be increased sending a request handler and it will be decreased receiving a reply handler. The maximum number of outstanding reply messages is the size of an input queue.

For the same reason, request and reply calls should not been nested to avoid a deadlock. So a request handler is not allowed to send another request message to the calling cluster. It may only send a reply message. A reply handler is not allowed to send request or reply messages.

5.5 Distributed Shared Memory (DSM)

A distributed shared memory is a mechanism allowing the user to access data on a remote cluster without explicit communication. In other words, the goal of a DSM system is to make communication transparent to the user.

As we have seen in Section 4.2, this mechanism has to be available for thread migration without copying the heap of the surrounding process. A DSM is implemented in pSather and can be used to solve the heap pointer problem after thread migration. But for the parallel C extension no support for a DSM was available and therefore had to be built.

The next sections will show the DSM model that is used by pSather and the one that was implemented for the parallel C extension.

Figure 14: References in Sather

5.5.1 *pSather*

In pSather every heap reference is coded as a so called *far pointer*. Any reference to non local objects will be transformed transparent to the user into an remote access across the network.

The way references are coded in pSather is shown in figure 14. The upper bits are used addressing the cluster. The lower bits are used to code the memory location of the object. The lowest bit is set if the pointer references non local objects and is not set if the pointer references local objects. Local objects are objects in the stack of the process that are located in higher memory areas (in this example higher than 2^{26} bits). A result of using the lowest bit to distinguish between local and remote objects is that an object has to start at an even memory address.

This representation has some disadvantages. Before an object can be accessed, it has to be checked if the lowest bit is set or not. Depending on the result, the object will be accessed locally or through a remote access.

There is a trade-off between the maximum number of clusters and the memory area that can be accessed. Increasing the number of bits for encoding the clusters would reduce the size of the DSM.

An example of an access of a remote object is the following.

1. Check the lowest bit of the reference. It is set for a remote object.
2. Decode the number of the cluster where the object resides. Decode the memory location of the object. This memory location is locally valid on the remote cluster. The decoding of the cluster number is a shift operation, the decoding of the memory location is a masking of the higher bits and of the lowest bit.
3. Send a request handler to the remote cluster using the Brahma library that reads the value of the decoded memory location.
4. Receive explicit reply handler with the value of the object.

5.5.2 *Parallel C Environment*

On the C programming level, no DSM model was implemented. Therefore thread migration would only be permitted if references from the stack to the heap had been prohibited. But this would mean to forbid the use of dynamic memory allocation and shared objects between threads, which is not acceptable for general programming.

To achieve the same capabilities as with pSather's DSM, new data types have been defined. This approach follows the object oriented software DSM model of pSather or PSO [Lüb95]. Additionally to the standard C types *int* and *float* the new types *shared_int* and *shared_float* have been written in C++. Created objects can be shared between threads on one cluster or between threads on all clusters. Choosing C++ allows a transparent intervention for operations on the types and allows the linking with the existing C run time library. An intervention is necessary for the following operations.

- Creation
  During creation memory for the object is allocated in the process heap. The pointer to this memory is encoded in the way shown in figure 14.
- Dereferencing (the * *operator*)
  Assigning a value to a shared object and requesting the value of a shared object, that does not reside on the local cluster, has to be replaced by a remote access. The access of a remote object is done in the same way that has been described in the previous section. If a value is assigned to a remote object the last two steps of the description change slightly. The request handler of step 3 has to write the value to the memory location and the explicit reply handler of step 4 can be omitted. If the shared object is local the unmasked memory location is accessed directly. The return type of **a*, where *a* is a *shared_int*, is *int*.
- Accessing the *address operator &*
  If the address operator & is used to access the address of a shared object, it has to return the encoded reference. These references are allowed to be on the thread stack during the thread migration, because these references are globally valid. The return type of *&a*, where *a* is a *shared_int*, is **int* (pointer to a *int* value). After assigning an *int** to *a*, the shared object *a* will reference the memory location given by *int**.

It is possible to create an object on every cluster, so that all objects reference the same data. The data will only reside on one specified cluster. To create these global objects from a shared object *a*, the method *a.create_global(cluster_nr)* has to be called on every cluster after the initialization of the migration library and after a shared object *a* has been created on every cluster. In the given example all shared objects will reference data on the cluster with the number *cluster_nr.*

If the new data types are used for dynamic memory allocation they return references that are valid on all clusters. If a shared object referencing data on a remote cluster is accessed, the access are transparently transformed into a remote access. Therefore the data types enable the use of dynamic memory allocation along with thread migration.

5.6 GLOBAL SYNCHRONIZATION OBJECTS

Working with multiple threads it becomes necessary to synchronize their execution. One basic synchronization mechanism is a *semaphore*. A semaphore consists of a counter and mechanism that protects the exclusive access to the counter for one thread called *lock*.

There are two operations on a semaphore, wait and signal. The wait operation on a semaphore checks to see if the counter is greater than 0. If so, it decrements the counter and just continues. If the value is 0, the thread will be blocked and has to wait for the semaphore. All operations on the counter have to be protected by the lock mechanism to ensure that only one thread per time accesses the counter.

The signal operation increments the counter of the semaphore. If threads are waiting for this semaphore, one will be unblocked and can continue its execution. After a signal on a semaphore with threads waiting for it the counter is still 0. If no thread is waiting for the semaphore the counter is in-

cremented. This can be seen as a saving of an unblocking of a thread for the future.

A problem related to the last section is the one of semaphores in the presence of thread migration. If two threads use a local semaphore to synchronize their execution it is normally not possible to migrate one of these threads, because the migrated thread can not access the semaphore on the other cluster. As synchronization is essential, a semaphore that is valid on all clusters was implemented.

The address of a global semaphore is also coded in the way shown in figure 14. Accesses to remote semaphores have the additional problem that the threads that synchronize on the semaphore can run on different clusters. If a thread has to wait for a remote semaphore this thread has to be woken up on its source cluster when it gets the semaphore. If a thread waits for a local semaphore its TCB is stored in a *block queue*, that is assigned to every semaphore. When it gets the semaphore the TCB is removed out of the block queue and is then put on the run queue to be restarted. In the remote case the reference to the TCB has to be encoded before it can be stored in the block queue. Otherwise it could not be determinated on which cluster a thread has to woken up.

The following example shows the wait operation on a remote semaphore.

1. Decode the cluster number and the address of the semaphore.
2. Send a request handler to get the counter of the semaphore (other threads are not allowed to access the semaphore after reading the counter).
3. Receive reply handler with the value of the counter.

4.a. If the counter is smaller than 1:
Send a request handler to store a encoded reference of the TCB in the semaphore block queue and decrement the counter.

4.b. If the counter is greater than 0:
Send a request handler to decrement the counter.

5.a. Block the thread that is waiting for the semaphore.

5.b. Continue the execution.

The following interfaces are provided for global semaphores.

- *void gl_sema_create(int count)*
  Create a semaphore and set counter to *count*.
- *void gl_sema_wait(gl_sema_t *sema)*
  Perform a wait operation on semaphore *sema*.
- *void gl_sema_signal(gl_sema_t *sema)*
  Perform a signal operation on semaphore *sema*.
- *void gl_sema_trywait(gl_sema_t *sema)*
  Perform a wait operation on semaphore *sema*. If the semaphore can be taken, decrement counter and continue, otherwise leave the counter unchanged and continue without blocking.
- *void make_sema_global(gl_sema_t **sema, BR_cluster_t cluster, int count)*
  The semaphore *sema* of all cluster is changed to reference a local semaphore on the cluster *cluster*. The counter of the local semaphore is set to *count*. Has to be called on all clusters of the network.

6.0 *Thread Migration with Active Threads*

The previous chapters have shown the basic mechanisms for thread migration and the given environment. This chapter describes the choice of the migration mechanism and its implementation.

6.1 CHOSEN MIGRATION MECHANISM

Section 4.3 gives two possible solutions for the stack pointer problem. One solution is the pointer manipulation, the other one is preventive stack reservation.

The pointer manipulation scheme of Section 4.3.1 needs detailed information about internals of the compiler, that produces the executable. This solution is not suitable for the given system. Although thread migration had to be implemented for the programming language Sather, no access to the needed internals has been available, because Sather uses the two step compilation (see figure 11 on page 39). The executable is produced by a C compiler and we could not get detailed information how the compiler works with pointers.

Therefore the only possible solution for the stack pointer problem was the preventive stack reservation.

The disadvantage of this solution is that the maximum number of threads is more restricted. But most applications with Active Threads will not reach

such a huge number. Active Threads are designed for small task and hence they get along with small stack sizes. The reservation of stack space for 10000 threads with a single stack of 8K byte results in an overall space of 78M bytes. The reservation of this space is only in the virtual memory of the process. A reservation of the physical memory is only done when the stack is used.

6.2 IMPLEMENTATION

A thread migration mechanism interacts with the different stages of the thread life. This section shows how the thread migration is implemented and which additional changes of the existing thread package had to be done.

6.2.1 *Start of the migration library*

At the initialization of the migration library a *pool* of threads with preallocated data structures for the TCBs is created. Pooling of the TCBs has the advantage that the creation of a thread is faster than without pooling, because the time to allocate memory is saved.

After the pool creation an area for the thread stacks has to be reserved in the virtual memory. As it has to be guaranteed that the memory locations are the same on all clusters, the only safe way is to compare the ranges of all clusters.

The chosen solution is to request the complete space for every thread stack at one time. This is done with the *mmap* command. Mmap establishes a mapping between the process virtual address space and a given file descriptor. The file descriptor can represent open disk files or any UNIX device. When mmap is used with the special device */dev/zero,* a unnamed memory region of a given length is created, initialized to 0 and mapped in the virtual address space of the process. The actual mapping of physical space is only done on demand when the memory region is referenced first.

After reserving the space in the virtual memory, all clusters have to synchronize and ensure that the area is the same on all clusters. If this is the case the area is divided into segments. Each segment is then uniquely assigned to a thread and is used as stack.

6.2.2 *Creation of a thread*

The migration library offers a pool of thread with pre allocated data structures for the TCBs. To start a new thread, one has to take an element out of the pool and set the program counter and the arguments of the called function. The thread can then be put on the run queue and wait for its execution.

The problem with the used migration mechanism is that not every thread can be taken out of the pool, because the thread with the same ID may already run on another cluster. Then it would not be possible to migrate this thread, because only one thread with a certain ID is allowed to run on all clusters.

One way to avoid this, would be to communicate across all clusters at the start of a thread to mark used threads. This would lead to an unacceptable overhead for every thread creation.

The better solution is to assign distinct subsets of threads to each cluster. A cluster is only allowed to start threads of its subset. This scheme is shown in figure 15. One can see the memory area for the thread stacks and its segmentation. There are slots in this area for threads that can be started on the cluster (three clusters are used in the example).

The information, which thread can be started, is now local and there is no overhead of a communication. This approach restricts the number of threads, which can be started on one cluster, more the one discussed above. When the maximum number of threads is started on one cluster, it is not possible to start a thread with an ID, that is assigned to another cluster but not started.

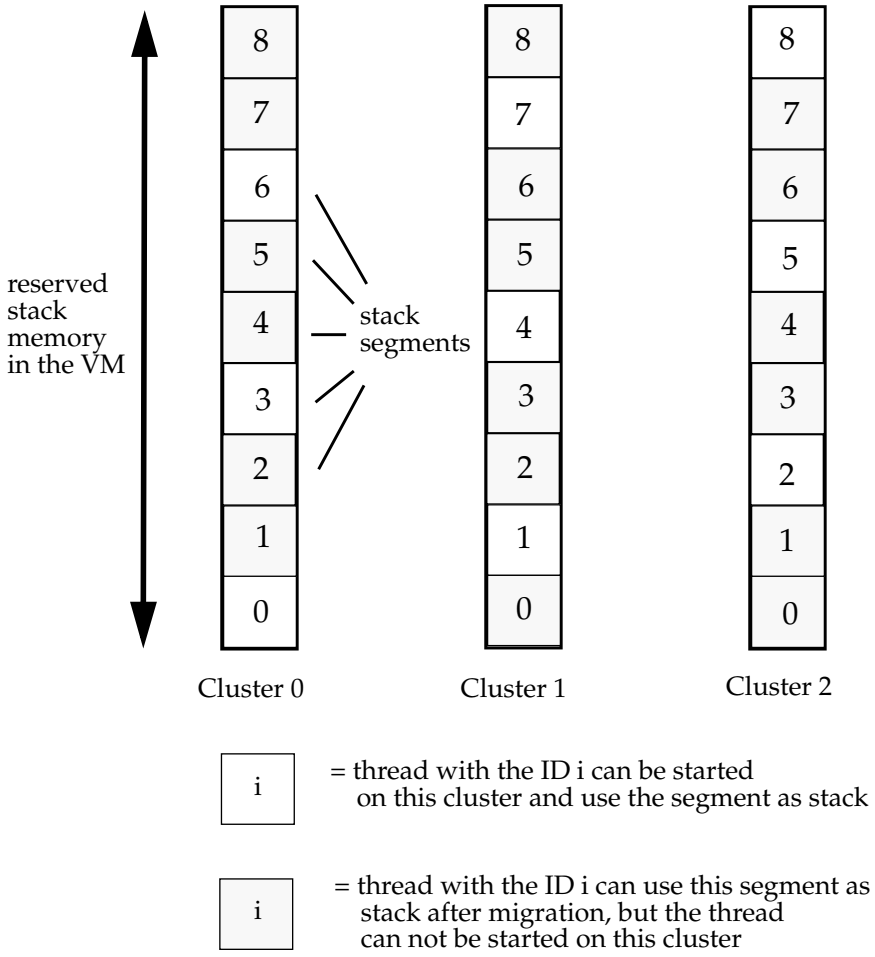

Figure 15: Segmentation of the stack memory

6.2.3 *Termination of a thread*

When a thread aborts on the cluster on which it has been started, nothing different to a termination without a thread migration mechanism has to be done. The TCB has to be put back into the thread pool.

Only if a thread aborts on a cluster different from the one started on, the handling changes. When a thread migrates from the source cluster to the destination cluster, the TCB is put back into the thread pool on the source

cluster. The thread with this ID should not been used again until the computation on the destination cluster finishes. Therefore the TCB has to be marked to not be reused.

If the thread aborts on the destination cluster this mark has to be reset, because the thread can be used again. So the abortion of a thread after its migration requires one communication step.

6.2.4 *Migrating a running thread*

The thread migration package offers a call to migrate a running thread. The running thread itself has to call this function, because it is nonpreemptive.

The migration of a running thread is divided into the following steps:

1. The thread calls the function to migrate itself.
2. The register content of the thread is saved on the stack.
3. The address of the TCB for the thread ID on the destination cluster is requested.
4. The TCB and its stack are copied to the destination cluster.

5.a. *Source*: The TCB is put back into the thread pool. It is marked to not be reused.

5.b. *Destination*: Thread is put on the run queue and can be restarted.

The user interface is the following:

void migrate_me(BR_cluster_t to),

where *to* is the cluster to which the thread shall be migrated.

6.2.5 *Get a thread from another cluster*

Instead of migrating a running thread, one cluster can request a runnable thread from another cluster. The only threads, that can be migrated from the destination cluster, are those in the run queue. It is not possible to get a currently executing thread, because it is non preemptive, or to get threads that are blocked.

It is also more efficient to take a thread out of the run queue than to get a currently running thread. The threads in the run queue are ready to be restarted. Their register content is saved on the stack and can easily be restored.

The following steps have to be done:

1. Request to find a suitable (see the following user interface) thread of a run queue on the destination cluster. Get the address of the TCB and the thread ID.
2. Take the TCB for this ID out of the local thread pool.
3. Copy the TCB and its stack to the source cluster.

4.a. *Destination*: The TCB is put back into the thread pool. It has to be marked to not be reused.

4.b. *Source*: Put thread on a run queue.

The user interface is the following:

int migrate_to_local(BR_cluster_t from, int queue_type, int state), where*:*

- *from* is the cluster from which a thread shall be migrated,
- *queue_type* specifies in which queue of the Active Thread scheduler shall be searched (e.g global run queue, bundle queue, see section 5.3)
- *state* means the state of the thread which shall be migrated (ready to run, just initialized)

The function returns a *0* if no thread has been migrated or a *1* otherwise.

7.0 *Load Balancing*

The reason for thread migration is to achieve a better overall performance. The best performance will be achieved when the load of all clusters is nearly is the same. The load should be balanced across all clusters.

Static load balancing is usually not the best choice, because the load can not be predicted for all applications. The application can also run under a multiuser and multitasking OS. So the load of the other user on these clusters differs and interacts with the running application.

The better way to balance the load is dynamically. Threads should migrate away from high loaded clusters. If a thread works on remote objects, it is better to migrate it to the cluster where the data resides.

This thread migration package wants to give a possibility to establish dynamic load balancing. It offers one implemented dynamic load balancing algorithm. but it is flexible enough to implement different algorithms and policies. The choice which policy to follow, should be left to the user as far as possible.

This chapter gives an overview about what can be done and what can not be done. The package offers calls to migrate running threads and to get threads of a remote run queue. The abilities of both calls to establish load balancing are given in the next sections.

7.1 CLASSIFICATION OF LOAD BALANCING ALGORITHMS

This thesis will follow a classification that has been introduced by Lüling, Monien and Ramme [Lül91]. The given classification is very general and covers all distributed algorithms, but it does not use too many complex properties and points of view.

There are two basic parts in any dynamic distributed load balancing strategy. One is the decision part, the other is the migration part. During the decision part the decision is made whether or not to migrate. The base of the decision can be local information (e.g. load on the current cluster) or global information (e.g. other clusters have no work).

The actual migration is done in the migration part. After the decision to migrate, the only open question is where to migrate. If the migration only includes direct neighbors of the cluster, the strategy has a local migration space. Otherwise the strategy has global migration space.

Lüling et al. [Lül91] give a classification according to the decision base and the migration space. It distinguishes between local and global decision and migration activities. The initiator of the load balancing activity is also included. The initiator of the activity can either be the sender or the receiver of the load balancing unit.

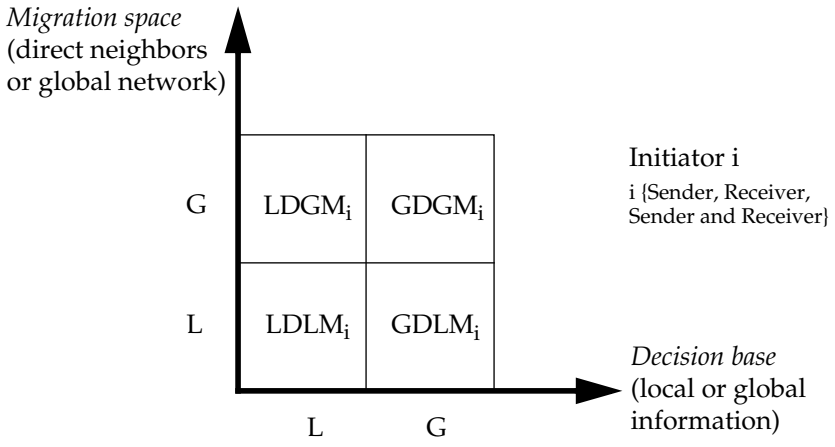

Figure 16: Load Balancing Classification

7.1.1 *Example Algorithms for the major Categories*

After giving a scheme to classify load balancing algorithms some known algorithms can be classified.

- $LDLM_{Sender}$ strategy
  An algorithm that uses a $LDLM_{Sender}$ strategy, is the local random algorithm. In fixed time intervals each cluster tests local information and decides whether to migrate or not. On a positive decision, it sends a load unit to a randomly chosen direct neighbor. The initiator of the load balancing activity is the sender of a load unit.
- $LDGM_{Sender}$ strategy
  If we modify the local random algorithm in the way, that the migration space is the complete network, we get a global random algorithm. A load unit can be sent, different to the local algorithm, to any cluster. Again, the initiator of the load balancing activity is the sender of a load unit.
- $GDLM_{Sender}$ strategy
  An algorithm with a $GDLM_{Sender}$ strategy is the gradient model method. The decision is based on gradients. The gradients are vectors that consist of load and distance information of all clusters. The local load of every cluster is classified in discr ete states. If one cluster is in the state of highest load, it sends a load unit to the direct neighbor, which is on the shortest path to a cluster with the lowest load. The sender of a load unit initiates the load balancing activity.
- $GDGM_{Receiver}$ strategy
  In the bidding algorithm, the decision to migrate a load unit is based on bids of clusters which are in a state with low load. Every cluster with high load gets bids from clusters, that want to receive a load unit. The load unit will be migrated to the cluster, that has a certain maximum distance to the sender and that sent the highest bid. The maximum distance will be increased, if the cluster gets not enough bids during a time interval. The receiver is the initiator of the load unit in this algorithm.

7.2 LOAD BALANCING WITH THREAD MIGRATION

The thread migration package offers two mechanisms to migrate threads. Different kinds of algorithms can be implemented with each of them. The next two sections will give their classification corresponding to the scheme described in Section 7.1.

7.2.1 *Migrating Running Threads*

The main restriction concerning the migration of running threads, is that the threads can not be preempted. So the thread itself is the only one to start the migration. It is the sender of a load unit.

The only possibility to balance the load with a running thread is that the thread calls explicitly a function, that decides whether or not to migrate. A good choice to call such a decision function is before a thread explicitly gives up its execution to another thread. The state of the running thread has to be saved regardless of the decision, if the thread should be migrated or not.

The basis of the decision is left to the user. The user has the freedom to implement its own decision function, that can work with local or global information. Various policies can be pursued. The implementation of a decision function is very easy. In the first part the function has to check some conditions. In the second part a *migrate_me(to)* is called or not based on the conditions of part one.

Reasons why a running thread should be migrated are:

- The load of the current cluster is high.
- The thread is working on data on a remote cluster. It may be better to reduce the network communications and migrate the thread to the remote cluster. It is very easy to determine if a data object resides on a remote cluster. Sather offers a explicit instruction to ask whether or not an object is local.

7.2.2 *Migrating threads from a remote run queue*

This mechanism does not have the restriction that only the thread, that is to be migrated, itself can initialize the migration. Any thread, that is in a run queue can be chosen. The decision which thread will be taken from which run queue is left to the user and can be specified in the user interface of *migrate_to_local(from,queue_type,state)*. One policy could be to choose a thread, that has been initialized, but never ran on the processor. The used part of the stack of these threads are small, so that the migration costs are low.

One good point to decide to migrate a remote thread is when all run queues on the local cluster are empty. If one can migrate a thread, the load of a remote cluster will be lower and the overall execution time may be reduced.If it is not possible to migrate a thread from a remote cluster, because its run queue is also empty, the overhead for the communication does not matter. This method is known as work stealing [Blu94].

An interface to enable work stealing to balance the load dynamically has been implemented. It includes the two functions:

- *void enable_work_stealing(int queue_type, int state),*
  to enable the load balancing, where *queue_type* specifies in which queue of the Active Thread scheduler shall be searched (e.g global run queue, bundle queue, see section 5.3) and *state* means the state of the thread which shall be migrated (ready to run, just initialized),
- *void disable_work_stealing()*,
  to disable the load balancing.

7.2.3 *Thread mechanisms classified*

The last two sections summarized the abilities of the thread migration mechanisms for load balancing. This section classifies them in the scheme of Section 7.1.

A running thread can only be migrated by itself. So it is the initiator of the load balancing activity that sends a load unit to a remote cluster. The decision base can be local information or global information. It is possible to send threads to any cluster in the network.

The second mechanism migrates threads from remote clusters to the local one. The initiator of the activity is the receiver of a load unit. The decision base can again be local or global information. Threads from all clusters of the network can be migrated.

Figure 17 classifies both mechanisms. Any mechanism covers the complete matrix for a given initiator. So using both mechanisms enables the implementation of any load balancing algorithm that can be classified with the used scheme.

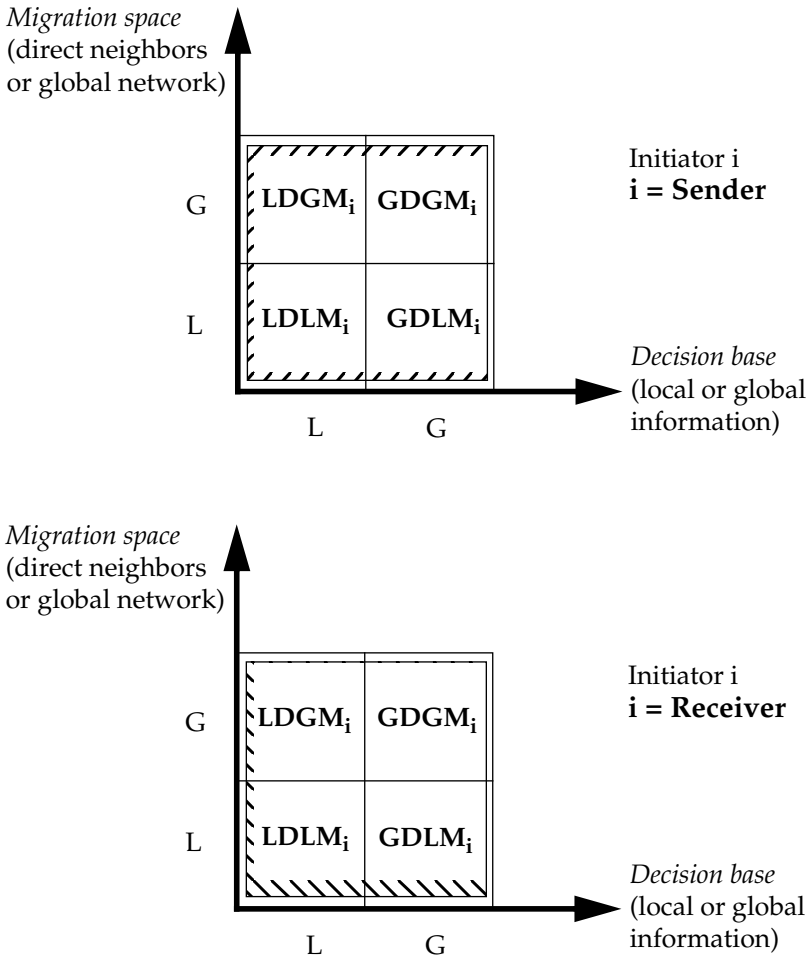

Figure 17: Classification of the thread migration mechanisms

8.0 *Performance Results and Examples*

In this chapter performance measurements of the thread migration system are presented. It starts with the Active Thread library and continues with the communication network and the network library. To make estimations if a thread migration is effective, an analytical derivation is given in the fourth section. The third section summarizes the times of the thread migration primitives. The chapter finishes with speedups, that applications have achieved when they used dynamic load balancing based on thread migration.

8.1 Active Threads

The design goal for Active Threads was to build a system for high-performance fine-grained pr ogramming. Fine grained programming helps to:

- express the logical concurrency of an application,
- port applications, because they are not very sensitive to the number of available processors,
- achieve better load balancing. The greater the number of available parallel tasks the higher the probability that processors are busy.

It is only possible to achieve good performance with multithreaded programs, if the overhead of the thread management is small. Commercial thread packages only offer threads, that are just lighter weighted processes.

The user has still to spend some thoughts choosing a reasonable partitioning of the application.

Most of the thread system overhead comes from:

- thread creation
- thread synchronization
- extra memory usage

Figure 18 compares times for several thread operations of Active Threads to times of commercial systems. The following operations are measured:

- *thread create*
  thread creation
- *null thread*
  entire runtime of a thread that performs a null call from creation to termination
- *context switch*
  context switch between two threads
- *uncontested mutex*[6]
  successful lock operation on a mutex without blocking
- *uncontested semaphore*
  successful wait operation on a semaphore without blocking
- *mutex try*
  non blocking try operation to lock a mutex that fails
- *semaphore try*
  non blocking try operation to get a semaphore that fails
- *mutex ping-pong*
  repeatedly synchronization of two threads on with a mutex
- *semaphore ping-pong*
  repeatedly synchronization of two threads on with a semaphore

Active Threads perform significantly better. E.g. the thread creation or the entire runtime of a thread from creation to termination (null thread) is orders of magnitude faster. Some of the times are summarized in table 3. Active Threads are compared on a 4 CPU SPARCstation 20 with Sun Solaris

6. A mutex is a synchronization object.

(version 2.5) threads and with Ariadne (see section 4.4). Ariadne is a user thread package recently developed at the Purdue University [Mas96] that supports thread migration and offers the same flexibility and portability as Active Threads. Compared to Solaris threads both Ariadne and Active Threads are significant faster. But Active Threads are still a factor of 3 to 10 faster than Ariadne.

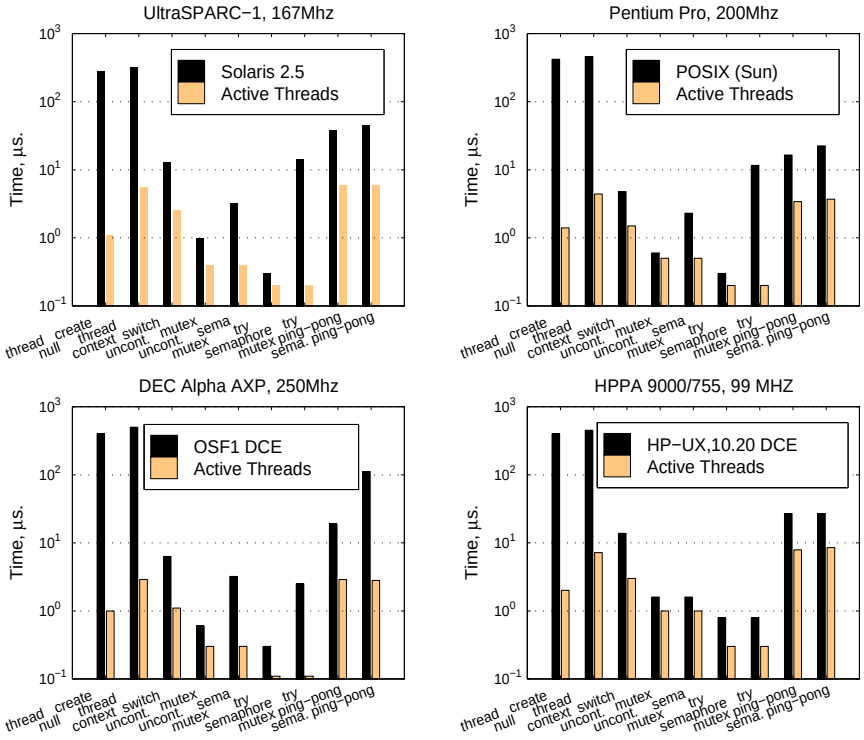

Figure 18: Active Threads versus proprietary thread system (measurements done by Boris Weissman)

.

Table 3: Comparisons with other systems, times in microseconds (measurements done by Boris Weissman

Operation	Solaris Threads	Ariadne	Active Threads
null thread	1715	40	14
thread create	1620	35	3.5
context switch	30	15	5.5

8.2 COMMUNICATION NETWORK

Section 6.2.4 and section 6.2.5 showed the steps to migrate threads. Basically they are:

- Save the thread context (only for running threads)
- Send the logical ID and copy the TCB to the destination cluster
- Copy the thread stack

The previous section has shown the costs of Active Thread operations. The other important cost factor migrating threads is the performance of the communication network.

The cluster used for this measurements have been connected by a Myrinet network. Myrinet is a Gigabit-per-second network. It is based on technology used for packet communication and switching within parallel supercomputers. Its characteristics are high bandwidth (up to 11 Mbyte/s between UNIX processes) and low latency (20 microseconds for short-packet transfer between UNIX processes) [Pak97].

The Myrinet was used with the Brahma library (see section 5.4). Brahma offers two interfaces to copy data between clusters: get and store. Store copies data from the local cluster to a remote clusters, get reads data on a remote cluster and copies it to the local cluster. The times to transfer an

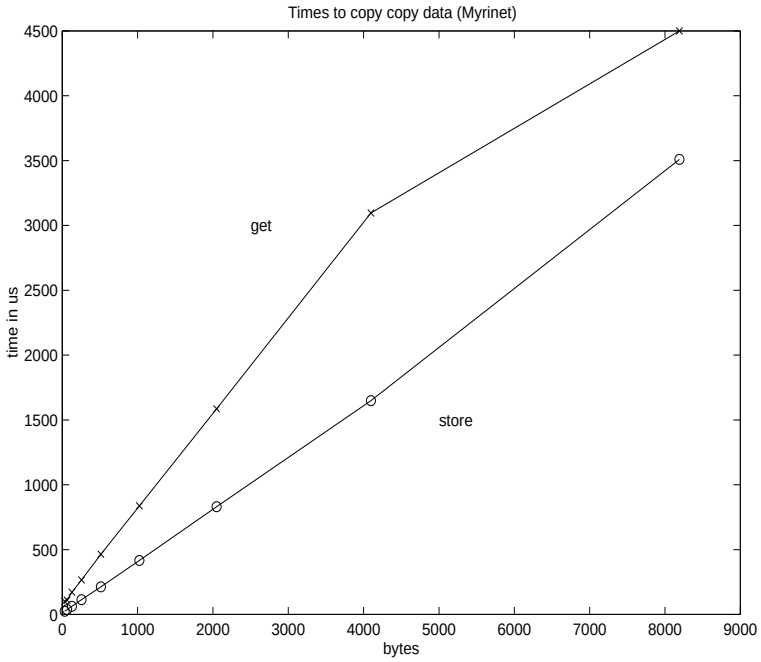

Figure 19: Performance of the Myrinet communication network

amount of data in the range of usual thread stack sizes are given for both interfaces in figure 19. Storing of data is more efficient, because the get is implemented as a store called on the remote cluster.

The times to copy data are orders of magnitude higher than the times for thread operations. It can therefore be expected that the costs of thread migration are closely related to the size of the thread stack, because the transfer time of the stack will dominate the time to migrate a thread.

8.3 THREAD MIGRATION PRIMITIVES

All measurements of thread migration times have been done on three SPARCstation 10, each with 4 hypersparc CPUs. The workstations were connected by a Myrinet network interface that is described in the previous section.

The times to migrate threads with the two interfaces given in section 6.2.4 and section 6.2.5 have been measured in the following way:

- Migrate the currently running thread:
  A thread has been started on one cluster that called *migrate_me(to)* in a way that the thread migrated ongoing to the next cluster of the network. The roundtrip across 3 clusters has been repeated 20000 times.
- Migrate a thread to the local cluster:
  1000 threads have been started on one cluster. After this all clusters of the network synchronized. Another cluster migrated all threads to itself.

The measurements of the migration costs prove the assumption that the time for the stack copying is dominating the migration costs. This can be seen in figure 20 and figure 21. Both figures show the times for thread migration and corresponding time transmitting the thread stack. The curves of the migration times nearly runs parallel to the curves of the transmission times. The offset between both curves is the time for thread operations and for one communication step.

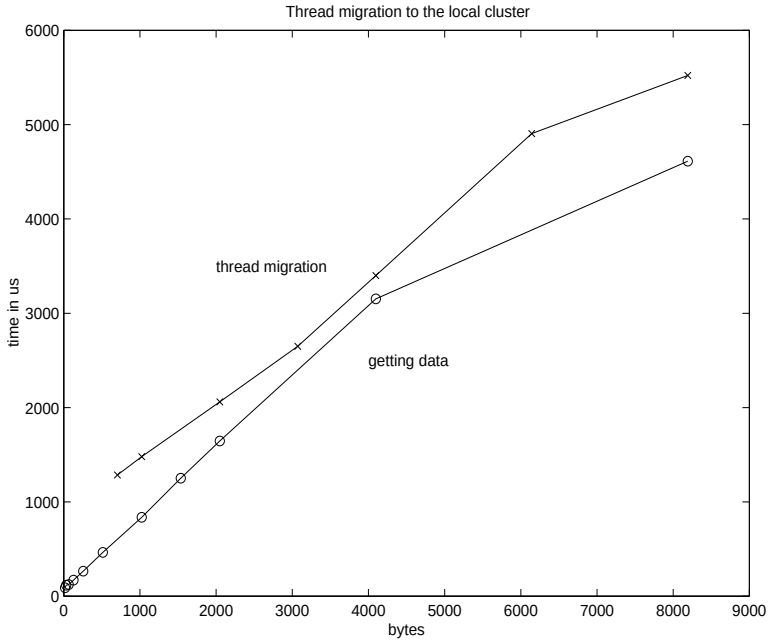

Figure 20: Get a thread from a remote cluster

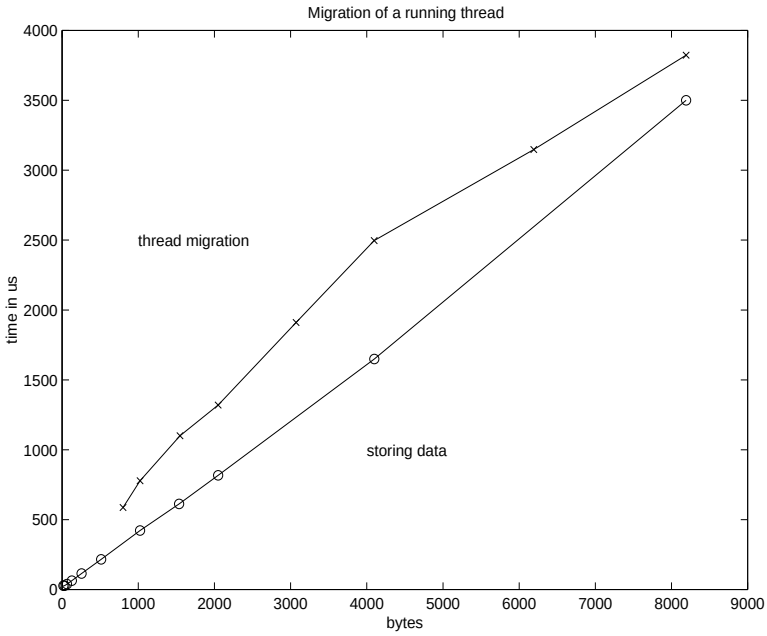

Figure 21: Migrate a running thread

Running threads can be migrated faster than threads from the run queue of a remote cluster, although the context of the running thread has to be saved before its migration. This is related to the different times to get or to store data between two clusters (see figure 19). It shows that the migration time is dominated by the transmission time of the thread stack.

Table 4 summarizes the time measurements of thread creation and thread migration and compares the thread migration times with the Ariadne system [Mas96]. The absolute times for thread migration with Ariadne and Active Threads are not directly comparable, because the times are dominated by the transfer time of the thread stack. The extreme difference of both times is therefore related to the used network technology. The times of thread migration without stack transfer times with Active Threads are half the times of the Ariadne system, although the Ariadne times have been measured on SPARCstation 5 that is usually faster than a SPARCstation 10. Another remarkable point is that migrating an Active Thread is more than 3 times faster than the thread creation of the Sun Solaris thread library. This comparison shows the efficient implementation of the migration primitives.

Table 4: Migration and related times on a Sun Sparcstation 10

Operation	Active Threads	Ariadne	Solaris 2.5
thread creation	**3.5 us**	**11 us**	**1620 us**
thread migration	**582 us**[i]	**11 310 us**[ii iii]	**-**
thread migration without transfer time of the stack	**250 us**	**510 us** [iii]	**-**

i. stack size: 800 bytes
ii. stack size: 1432 bytes
iii. On a Sun SPARCstation 5

8.4 More effective with Thread Migration

The question if an application gains a speedup migrating a thread depends on two things: the thread execution time and the time to migrate a thread.

This section gives a more detailed derivation of the question when a thread migration will be worth-while. The following symbols will be used (see also figure 22):

T_i = execution time of the application on cluster i
T'_i = execution time after migration
T_{ij} = execution time of thread j on cluster i
T'_{ij} = execution time of thread j after migration from cluster i
T_M = time to migrate a thread

First the overall execution time of the application on one cluster is the sum of the execution times of its threads

$$T_i = \sum_{j=1}^{N} T_{ij} \qquad \textbf{(EQ 1)}$$

After a migration of thread n from cluster j to cluster i, the execution time of the application on both clusters changed. After migration we get:

$$\begin{aligned} T'_i &= T_i + T_M + T'_{jn} \\ T'_j &= T_j + T_M - T_{jn} \end{aligned} \quad \textbf{(EQ 2)}$$

Without loss of generality it is assumed that $T_i < T_j$. The condition that thread migration reduces the overall execution time of the application is

$$\max(T'_i, T'_j) < T_j . \quad \textbf{(EQ 3)}$$

This lead to the equation system

$$\begin{aligned} T_i + T_M + T'_{jn} &< T_j \\ T_j + T_M - T_{jn} &< T_j \end{aligned}' \quad \textbf{(EQ 4)}$$

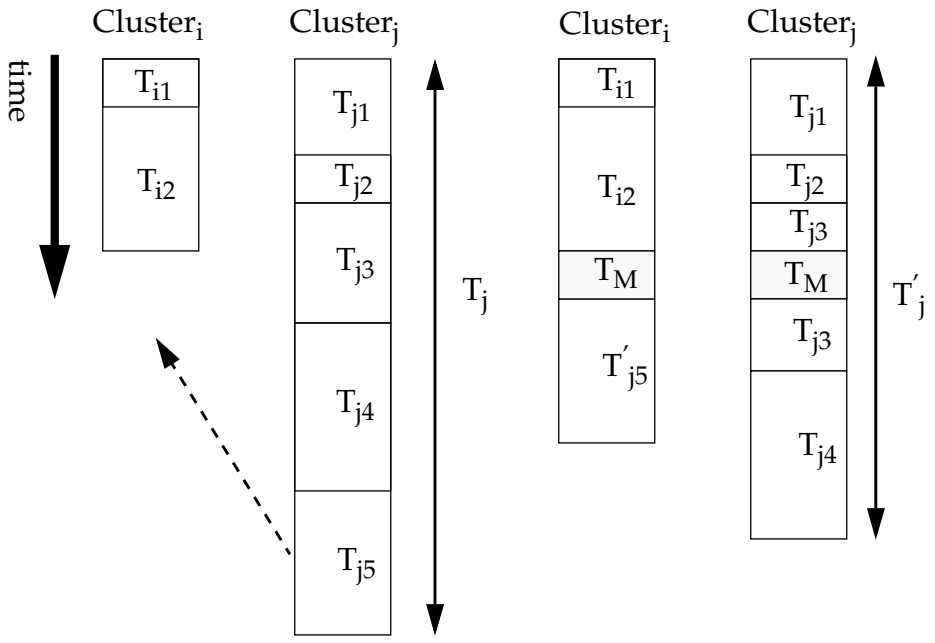

Figure 22: Execution time of an application on two clusters before and after migration

that can be reduced to

$$\begin{array}{c} T_M + T_{jn} < T_j - T_i \\ T_M < T'_{jn} \end{array} . \qquad \textbf{(EQ 5)}$$

The equitations above include two statements. $T_M < T'_{jn}$ says that the execution time of the thread that will we migrated should be greater than the migration time. This is obvious, because the migration time has to be added to the execution time of the application on both clusters.

The second statement is $T_M + T_{jn} < T_j - T_i$. It says that a thread shall only be migrated if the load unbalance between the two clusters is high enough. The difference between the execution time on both clusters has to be greater than the migration time and the execution time of the thread together.

From the analytical derivation we get a lower limit when migration will gain a speedup. The execution time of the thread has to be greater than the migration time. We also get a statement how unbalanced both clusters have to be.

8.5 Load Balancing Example

Equation 5 of the previous section gives a hint if a thread migration is effective to balance the load dynamically between two clusters. We measured the times for thread migration in section 8.3. The only open parameters are the execution times of the threads. But these execution times can not be predicted in the general case. Anyway, if we had the execution times, it would be more effective to balance the load statically placing all threads explicitly. Therefore an application used as example for load balancing based on thread migration should have a dynamic behavior during its run time.

The application chosen was adaptive quadrature. Adaptive quadrature is a flexible scheme for numerical integration. Classical schemes subdivide the domain of the integral into equal-sized subintervals and perform numerical integration like the Simpson rule on each subinterval. The results are combined to yield the required accuracy. In contrast to the classical scheme

adaptive integration subdivides the domain ongoing in as many subintervals as necessary to obtain a given accuracy.

The problem is to compute an estimate P, so that

$$\left| P - \int_a^b f(x)dx \right| < \delta, \text{ for a small and positive } \delta.$$

A recursive algorithm for adaptive integration is given in code example a).

a) Recursive algorithm for adaptive quadrature

```
void ad_quad(float *result, float a, float b, float err)
{
  float res_left, res_right, res_one, res_two, diff,middle, step;

  middle = (a+b)/2;
  step = b - a;

  res_one = 0.5 * step * (function(a) + function(b));
  res_two = 0.25 * step * (function(a) + 2*function(middle)+
                           function(b));
  diff = res_two - res_one;

  if(fabs(diff) < res_two)
  {
    /* Simpson rule */
    *result = res_two + diff/3;
  }
  else
  {
    ad_quad(&res_left,a,middle,err/2);
    ad_quad(&res_right,middle,b,err/2);
    *result = res_left + res_right;
  }
}
```

Adaptive quadrature has a dynamic behavior. Depending on the given accuracy and the function to integrate a different number of integration steps have to be made. The way to distribute the computational tasks was to divide the domain at the beginning in equal-sized subintervals and to start a thread to calculate each subinterval (see code example b). As result we got threads with differing execution times that could not be predicted in advance. This example was chosen as a representative for the class of problems that can be expressed using threads with differing execution times.

All threads have initially been started on one cluster. This is the worst case for load balancing, because the load is totally unbalanced at the beginning. The intention was that other clusters could migrate some load units and reduce the overall execution time. The other clusters followed a simple load balancing policy, called work stealing [Blu94] (see section 7.2.2). If one clus-

ter ran out of work, it tried to migrate a thread from the cluster, where all threads have been started. This is a LDGM strategy initiated by the receiver. The decision (processor has no work) is local, but the source cluster can be any cluster of the network.

b) Distributing the work

Input:

- r_border,l_border = domain of the integral
- error = accuracy
- nr_threads = number of threads

Output:

- result[0] = result of the integration

```
shared_float *result;
float a,b,step;
int i;

step = (r_border-l_border)/((float)nr_threads);
error  = error/((float)nr_threads);
result = new shared_float[nr_threads];
semaphore = (gl_sema_t**)
                at_malloc(nr_threads*sizeof(gl_sema_t*));
for(i=0;i<nr_threads;i++)
{
  semaphore[i] = gl_sema_create(0);
}
a = l_border;
b = l_border + step;
for(i=0;i<nr_threads;i++)
{
  at_migration_create_5(at_get_focus(),AT_UNBOUND,
                                ad_quad, (result_ptr+i),
                                a,b,error,(semaphore+i));
  a = b;
  b += step;
}
for(i=0;i<nr_threads;i++)
{
  gl_sema_wait(semaphore[i]);
}
for(i=1;i<nr_threads;i++)
{
  result[0] = result[0] + result[i];
}
```

The following pictures show the performance for different functions and different number of CPUs per cluster. In figure 23 the function $10 \cdot \sin(1/(0.00001 + 1000 \cdot \sin(20 \cdot x)))$ was integrated over the domain 0 - 2 with an accuracy of 0.00001. The execution times and speedups up to 4 processor are for a single cluster without dynamic load balancing. The execution times for 8 and 12 processors are measured with the work stealing load balancing strategy. The execution times increase with a higher number of created threads. The reason for this is obvious. The function has distinct areas, where many adaptive integration steps and where only few steps have to be done to achieve the same accuracy. The higher the number of threads the

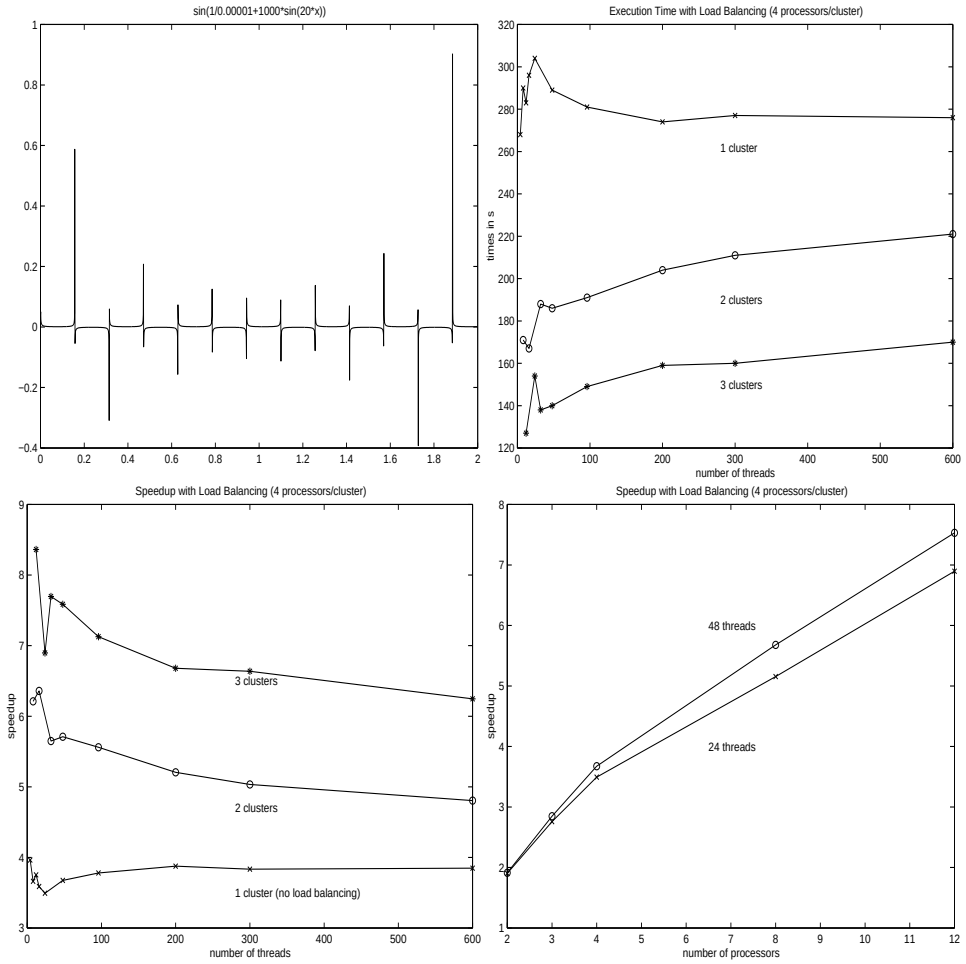

Figure 23: Benchmarks with load balancing

greater the probability to migrate threads with small execution times, so that the ratio between migration and execution time is not balanced.

An other example can be seen in figure 24. This time the function $123 \cdot \sin(1/x) - 134 \cdot \sin(20/(x-2)) + 120 \cdot \sin(3000 \cdot x^2)$ was integrated over the domain]0;2[. This time only one processor per cluster was used. On the borders of the domain the number of recursive integration steps increases. The execution times of the created threads are therefore varying a lot.

Figure 25 shows an example, where all threads have nearly the same computational work (the function $\sin(20000 \cdot x)$ was integrated). This avoids the

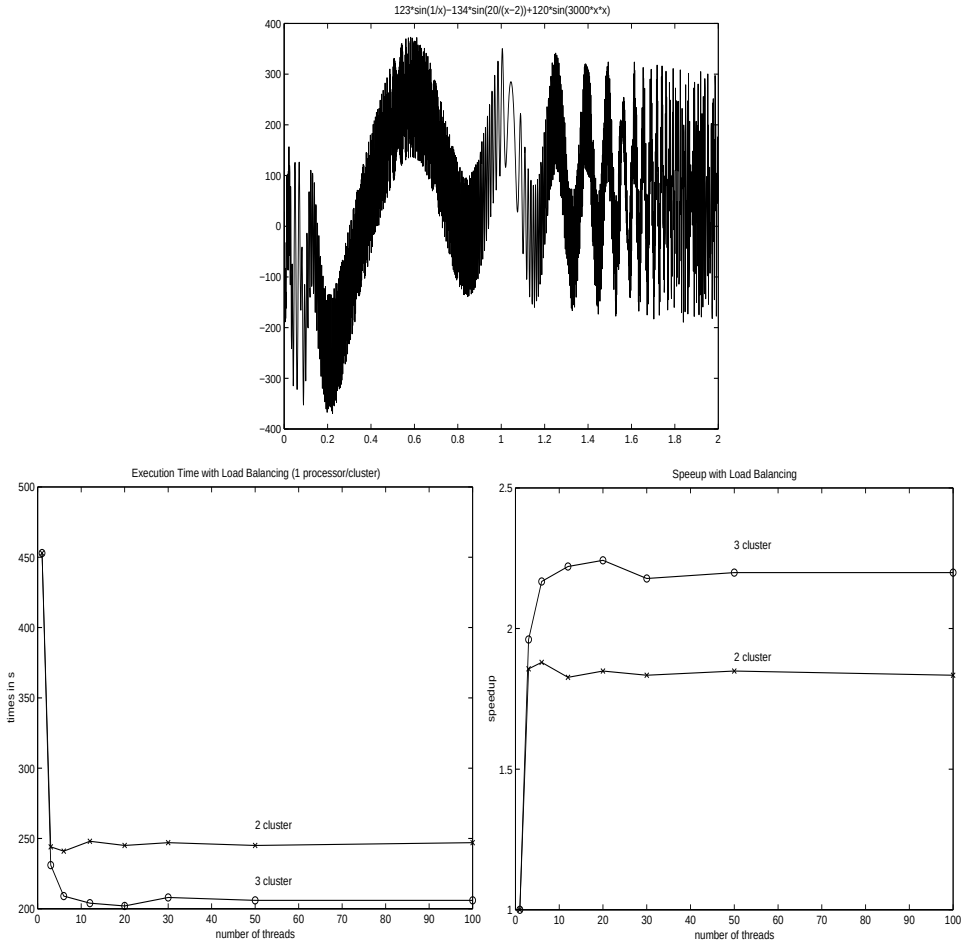

Figure 24: Load balancing with 1 processor per cluster

migration of threads with small computational work and should show the best possible speedups. It can be seen that the maximum speedups nearly reach the number of processors used.

The work stealing strategy helps to gain speedups for applications. On the other hand it can produce an overhead, if threads are requested from a cluster on which all processors are busy and on which no additional threads are in the run queue. The request will not lead to a migration of a thread, but the ongoing execution is interrupted during the handling of the request. The network handling is therefore overhead for the overall execution time of the application.

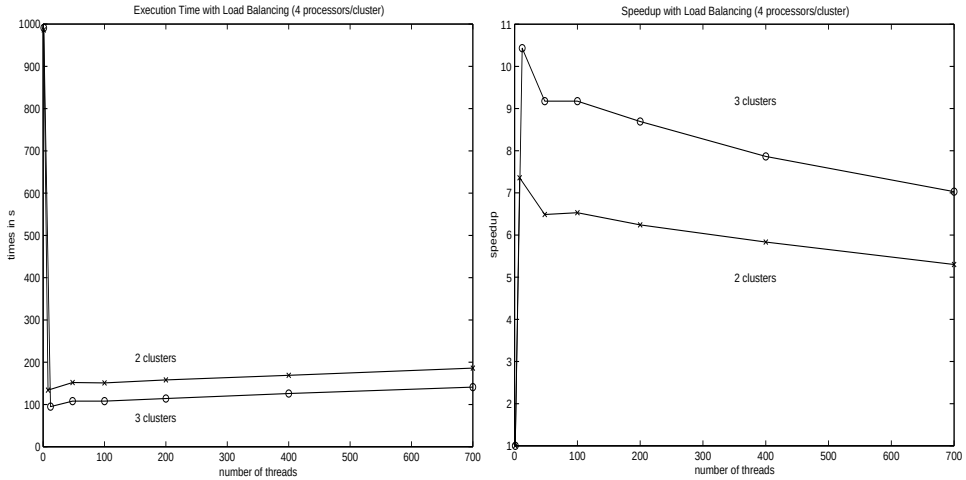

Figure 25: All threads have the same computational work

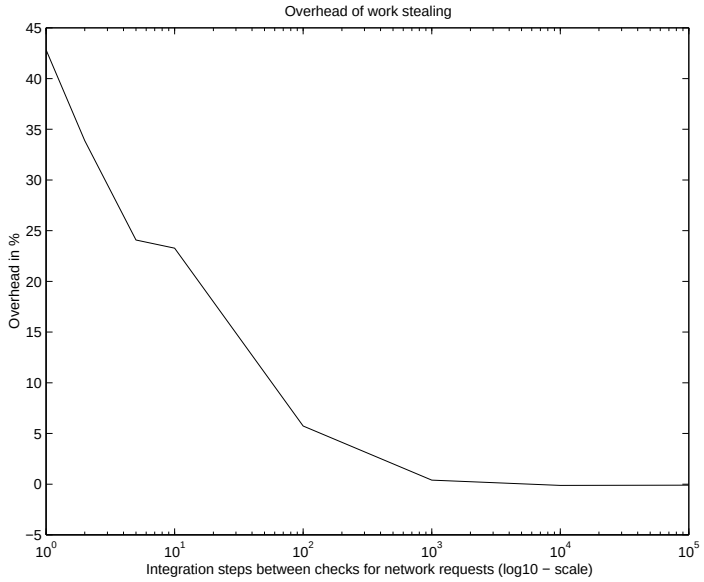

Figure 26: Overhead of the work stealing strategy when no threads can be migrated

In order to measure this overhead the adaptive quadrature was started initially with 4 threads on a 4 processor cluster. Two other clusters tried to steal threads unsuccessfully from the first cluster. Figure 26 shows the overhead of execution times with work stealing to the execution time without work stealing expressed as a percentage. The overhead was measured with

different numbers of integration steps between the next handling of a request to migrate a thread. The overhead is smaller than 5% for a reasonable ratio of integration steps to network handling greater than 100. For the previous examples a ratio of 100000 has been used. Therefore the work stealing strategy does not produce a significant overhead when used in a situation with unfavorable load distribution.

Speedups could be achieved in all examples, although the initial distribution of the load could not be more unbalanced. This leads to the conclusion that thread migration with Active Threads is suitable to balance the load of applications that have dynamic run time behavior.

Even if no application has been used that created threads dynamically during the run time, the method is also applicable for applications of that class. The method just considers the current load of the clusters and does not make any assumptions about the creation time of a thread.

The examples show that the presented method can be used to balance the load of badly placed applications with dynamic run time behavior. Further it can be used to avoid explicit placement. All threads of the application can be started on one cluster and the load will be automatically balanced. This makes applications more portable, because the application can dynamically adapt to changing number of clusters and does not have to be rewritten. It can also be used without the risk to produce a significant overhead when the load balancing algorithm can not successfully migrate threads.

9.0 *Conclusions*

This thesis introduced thread migration as a tool to ease parallel programming with multiple SMPs connected by fast networks. Simple dynamic load balancing strategies have been implemented that automatically migrate threads between cluster.

It has been shown that applications could improve their performance using a very simple load balancing strategy. Even for the worst initial distribution of the application, applications gained speedup up to the number of overall processors used. The improvements could be achieved for different problems and different numbers of processors.

These performance measurements show that load balancing eases the placement problem of parallel applications on multiple SMPs. If the initial distribution of the application is unfavorable, the unbalanced load can be balanced effectively. Even further, applications do not have to care for the placement. Speedups are achieved if all threads of the application are started on one cluster.

Active Threads offer a flexible event handler mechanism that makes it possible to implement even more flexible load balancing policies with thread migration than the one used in this work. This might gain in further improvements. One can think of migrating bundles of semantically related threads. One can also implement mechanisms to migrate data to improve the locality of the execution.

References

Agh86 G. Agha: *ACTORS: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, Massachusetts, 1986.

Art89 Yeshayahu Artsy, Raphael Finkel: *Designing a Process Migration Facility*, IEEE Computer Magazine 22(9), pp 47-56, September 1989

Blu94 Robert D. Blumofe, Charles E. Leiserson: S*cheduling Multithreaded Computations by Work Stealing*, in: Proceedings of the 35th Anual IEEE Conference on Foundations of Computer Science (FOCS'94), Santa Fe, New Mexico, November 20 - 22 1994
WWW: http://theory.lcs.mit.edu/~cilk/

Cas94 Jeremy Casa, Ravi Konuru, Steve W. Otto, Robert Prouty, Jonathan Walpole: *Adaptive Migration systems for PVM*, in Proceedings of Supercomputing '94, pp 390 - 399, Washington D.C., November 1994
WWW: http://info.computer.org:80/conferen/sc94/otto.ps

Cha89 Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, Richard J. Littlefield: *The Amber System: Parallel Programming on a Network of Multiprocessors*, in ACM Symposium on Operating System Priciples, December 1989

Cro97 David Cronk, Matthew Haines, Piyush Mehrotra: *Thread Migration in the Presence of Pointers*, to appear in: Proceedings of the Mini_track on Multithreaded Systems, 30th Hawaii Internationa;l Conference on System Science, January 1997,
WWW: http://www.cs.uwyo.edu/~haines/research/chant/hicss97.ps

Dou87 Fred Douglis: *Process Migration in the Sprite Operating System*, in Proceedings of the 7 th ACM International Conference on Distributed Computer Systems, 1987

Hen96 John L. Hennessy, David A. Patterson: *Computer Architecure: A Quantitative Approach, 2nd Edition*, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996

Itz97 Ayal Itzkovitz, Assaf Schuster, Lea Wolfovich: *Thread Migration and its Applications in Distributed Shared Memory Systems,* to appear in: The Journal of Systems and Software, 1997
WWW: http://www.cs.technion.ac.il/Laps/Millipede

Jul88 Eric Jul, Henry Levy, Norman Hutchinson, Andrew Black: *Fine-Grained Mobility in the Emerald System*, in: ACM Transactions on Computer Systems, Vol. 6, No. 1, pp 109 - 133, February 1988

Lüb95 Stefan Lübke, Jürgen W. Quittek, Torsten Wiese: *Public Shared Objects: Shared Symbol Space for Multicomputer Architectures,* in: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications 1995 (PDPTA'95), pp. 626-635, Athens, Georgia, Nov. 3-4 1995
WWW:ftp://ftp.iuk.tu-harburg.de/pub/tech-reports/TR-403-95-004.ps.gz

Lül91 R. Lüling, B. Monien, F. Ramme: *Load Balancing in Large Networks: A Comparative Study*, in: Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing, pp 686 - 689, IEEE, 1991

Mas95 Edward Mascarenhas, Vermon Rego: *Migrant Threads on Process Farms: Parallel Programming with Ariadne*, Technical report TR 95-081, Department of Computer Sciences, Purdue University, December 1995.
WWW: **http://www.cs.purdue.edu/homes/edm/papers/aria2.ps.Z**

Mas96 Edward Mascarenhas, Vermon Rego: *Ariadne: Architecure of a Portable Threads System Supporting Thread Migration*, in Software - Practice and Experience, Vol 26(3), pp 327 - 356, March 1996
also as: Technical report TR 95-017, Department of Computer

Sciences, Purdue University, 1995
WWW: http://www.cs.purdue.edu/homes/edm/papers/aria1.ps.Z

Pak97 Scott Pakin, Vijay Karamcheti, Andrew A. Chien: Fast Messages: Efficient, Portable Communication for W orkstation Clusters and MPPs, IEEE CONCURRENCY Vol. 5, No. 2, pp 60-73, APRIL-JUNE 1997

Sto97 D. Stoutamire, *Zones: Portable, Modular Expression of Locality,* Ph.D. thesis, University of California at Berkeley, 1997.

Tan92 Andrew S. Tanenbaum: *Modern Operating Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1992

The86 Marvin Theimer: *Preemptable Remote Execution Facilities for Loosely - Coupled Distributed Systems*, PhD thesis, Stanford University, 1986

Zay87 Edward R. Zayas: *Attacking the Process Migration Bottleneck*, in Proceedings of the 22th ACM Symposium on Operating System Principles, pp13 - 24, ACM Press, New York, New York, 1987

Index

A
Active message. *See* Brahma, active message model
Active Thread 40
  bundle 42
  lightweight process 40
  migration cost 70
  performance 65
  run queue 42
  thread pool 52
  two level scheduling model 41

B
Brahma
  active message model 42, 43
  handler function 43
  network interface 42

C
C
  parallel extension 39

D
dev/zero 51
DSM. *See* Shared memory, distributed shared memory

H
Heap 10

K
Kernel mode 14
Kernel space. *See* Operating system, kernel space
Kernel thread. *See* Thread, kernel thread

L
Load balancing 56–61
  algorithm 58
  classification 57
  decision base 57
  dynamic 56
  example 73
  initiator 57
  migration space 57
  performance 75, 76, 77
  thread migration. *See* Thread migration, load balancing
  user interface 60

M
mmap. *See* Virtual memory, mapping

O
Operating system
  kernel space 17
  multitasking 11
  process table 10
  user space 17

P
PCB. *See* Process, process control block
Pool. *See* Active Thread, thread pool
Process 9–15
  communication 22
  context 11
  context switch 12, 15
  creation 22
  definition 9
  mode 14
  operation 14
  process control block 10
  resources 11
  state 13
Process migration
  logical process ID 25
  mechanism 26
  pre copying 26
pSather. *See* Sather, parallel extension
PVM 37

R
Run queue 42

S
Sather 38
  compilation 39
  parallel extension 39
Scheduling
  non preemptive 12
  preemptive 12, 13
Semaphore 47
  global 48
Shared memory
  distributed shared memory 29, 30, 44

far pointer 45
parallel C extension 46
pSather 45
PSO 46
virtual shared memory. *See* Shared memory, distributed shared memory
SMP 19
Stack 10
System call
blocking system call 19
wrapping 19

T
TCB. *See* Thread, thread control block
Thread 15–20
communication 22
context 16
context switch 22
creation 22
definition 16
kernel thread 17, 18, 41
resources 17
thread control block 16
user thread 17, 18, 41
Thread migration
heap pointer problem 30
implementation 50–55
load balancing 59
logical thread ID 28
mechanism 28
migrate running threads 54
migrate threads to the local cluster 54
migration cost 67
migration point 59, 60
pointer identification 33
pointer in register 34
pointer manipulation 32
preventive stack reservation 34, 35
stack 53
stack pointer problem 31
start of the library 51
thread creation 52
thread termination 53
user interface 54, 55
Trap 14

U
User mode 14
User space. *See* Operating system, user space
User thread. *See* Thread, user thread

V
Virtual memory 10
code area 10
global data area 10
heap. *See* Heap
mapping 51
stack. *See* Stack
VM. *See* virtual memory
VSM
virtual shared memory. *See* Shared memory, distributed shared memory

W
Work stealing 60