Using Locality Information in Userlevel Scheduling TR-95-14

Martin Steckermeier[1] and Frank Bellosa[2]
University of Erlangen-Nürnberg
Computer Science Department – Operating Systems – IMMD IV
Martensstraße 1, 91058 Erlangen, Germany

December 23, 1995

Abstract

In the past few years, MIMD parallel computers have become important not only in the field of high performance scientific computing, but also as ordinary compute servers. Applications that can not be parallelized by an appropriate compiler get more and more parallelized by use of the threads programming model.

Especially for machines having large caches and/or non uniform memory access special care has to be taken for an efficient handling of applications with a huge number of threads. Locality of data has to be taken into consideration as well as the memory access behaviour of threads to assure high cache reusage.

This paper examines several techniques that are necessary for efficient thread management on userlevel. Algorithms and mechanisms for scheduling as well as for synchronization are analysed for their suitability in userlevel thread libraries and the importance of using locality information is pointed out. Measurements with a prototype show the superiority of these concepts.

[1]email: mstecker@informatik.uni-erlangen.de
[2]email: bellosa@informatik.uni-erlangen.de

1 Introduction

The steadily increasing need for more compute power in the past few years has led to an introduction of parallel computers into several fields. Not only scientific computing has changed from dedicated high performance computers, like vector machines to general purpose parallel MIMD machines, but also offices more and more use parallel systems as compute servers. For the reason of good scalability most of these systems use large processor caches to reduce bus contention, whereas in larger systems, especially in the field of scientific computing, NUMA[1]-machines are a promising approach.

In a similar fashion as the hardware has changed over the years, software has to be adopted to the new properties of this computers.

1.1 Problem

First attempts to parallelize applications using ordinary processes like those in UNIX were not satisfying as synchronization mechanisms and context switches between processes were to expensive to generate efficient parallelized programs.

To reduce the costs, the concept of processes was broken up into two different concepts. For example the operating system MACH knows the concepts of a *task* modeling the domain for a computation, and *threads* that can run in parallel in one task. As all threads share one common address space, communication, synchronization, and context switches are much more efficient than in former systems.

Although the costs had been reduced by one order of magnitude, they were still to high for fine grained parallizations as each operation was performed by the operating system. Therefore, instead of using these *middle-weight* kernelthreads, thread management has been moved out to the userlevel. Userlevel threads are scheduled on top of kernelthreads that act as virtual processors, completely by code residing on userlevel. Using different libraries for thread management, operations can be optimized for specific properties and therefore offer best performance.

First userlevel thread libraries used concepts known from the field of conventional operating systems like UNIX. But on parallel systems like NUMA architectures these concepts are no longer applicable. Although in an increasing order communication gets implemented in hardware, communication costs outweigh the actual computation costs. As the communication costs can be influenced by thread placement, thread management must use information about data locality and the affinity of threads to processor caches for thread placement and scheduling.

1.2 Contemporary work

The necessity to include locality information in scheduling has led to a lot of publications over the last years. Most of these papers deal with the problem how to assign kernelthreads

[1] **N**on **U**niform **M**emory **A**ccess

to processors or vice versa. Locality information is used as basis for decisions in static or dynamic *space-sharing* or *time-sharing* systems.

Vaswani and Zahorjan stated that additional affinity considerations had only neglectable influence on the runtime of their test applications ([VZ91]). But this might be caused by the fact that they used a spacesharing approach. Only if the number of processors assigned to the applicationchanged, they used affinity information for rescheduling.

In [Tuc93] and [TTG95], Tucker mentioned that affinity scheduling can lead to improvements of a few percents on small UMA machines. Although the improvements are small he concludes that its worth to include affinity considerations because of an increasing importance in fine grained applications.

The importance of thread placement and locality considerations has been showed by Markatos, e.g. in [Mar93] or [ML93] by measurements on different architectures. He concludes that new programming models will have to be used to efficiently exploit modern machines and locality information will have to be used for scheduling. Even in cases of an imbalanced load, these locality scheduling can lead to better results ([ML91]).

Squillante and Lazowska used a queuing model to examine the influence of locality and affinity scheduling on the performance of parallel systems. Their results show that only little information is necessary to improve performance ([SL89]).

1.3 Overview

In contrast to the papers mentioned above, our work concentrates on the special needs for scheduling of userlevel threads. After a look on general purpose thread libraries and the problems arising on moderen NUMA architectures we propose structures and algorithms for memory conscious scheduling (MCS) to include locality information into the scheduling process. Furthermore we investigate the influence of the scheduling order on the number of cachemisses and show methods to reduce these cachemisses as well as necessary properties for efficient userlevel synchronization mechanisms.

We close with measurements that show the influence of different strategies on the runtime of several applications.

2 Userlevel Threads and NUMA Architectures

The structure of typical userlevel thread libraries was adopted from older conventional operating systems. One of the most important goals of these systems was to balance load between the processors available to maximize machine utilization.

Figure 1 shows the typical components and the corresponding operations of such libraries. Threads which are created by the library function `create()` get placed on a central runqueue. If a processor needs a new thread it dequeues one of these runable threads. As long as the runqueue contains runable threads no processor will run idle. When a thread stops there are three possibilities:

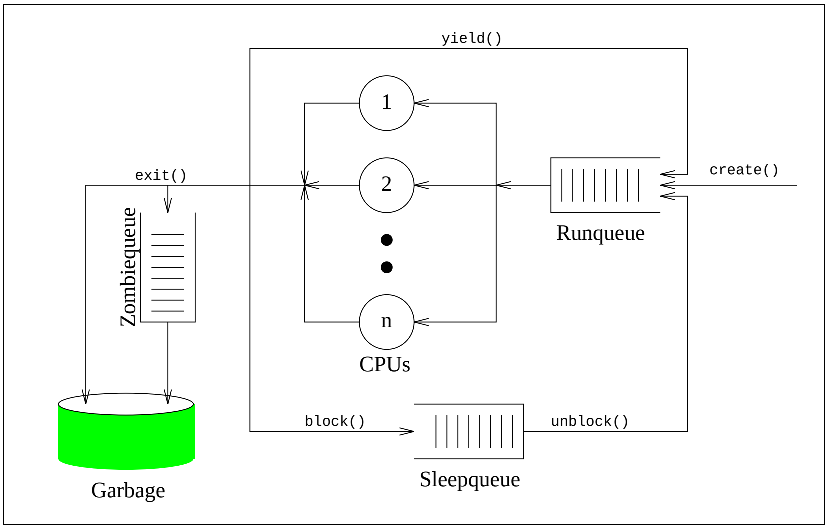


Figure 1: Structure of simple thread libraries

1. The thread yields the processor in favour of another runable thread and gets enqueued on the runqueue again(`yield()`).

2. The thread blocks due to some synchronization (`block()`) and gets enqueued on a central sleepqueue. When it is deblocked it will be dequeued from the sleepqueue and inserted into the runqueue again (`deblock()`).

3. When threads exit with `exit()`, *detached* threads are deleted at once whereas non-detached threads become zombies and get enqueued onto another central list.

This structures and strategies worked fine for smaller machines with few processors and UMA architecture. On larger machines however operating systems and userlevel libraries using these structures experienced a severe loss in performance. For example Markatos in [Mar93] shows that neglecting locality of data on current machines could lead to very low efficiency, especially on modern NUMA architectures.

To understand the reasons for the importance of including locality information into considerations on structures and mechanisms in userlevel thread libraries, one has to look on the attributes of NUMA architectures.

2.1 The Convex SPP – A Typical NUMA Architecture

Figure 2 shows the structure of the Convex SPP multiprocessor used for evaluating the concepts in the following sections. The machine is built up of up to 16 so-called hypernodes. Each of these hypernodes itself is a complete symmetric multiprocessor consisting of 4 CPU-blocks. These CPU-blocks consist of 2 HP PA-RISC processors, each having 1 MBytes virtually addressed instruction- and data cache, up to 512 MBytes local memory, an interface to CPU-blocks of other hypernodes, the so-called CTI[2], and access to a 5x5 crossbar switch. Across this switch each CPU-block has nonblocking access to the local memory of other CPU-blocks on the same hypernode and the I/O-interface.

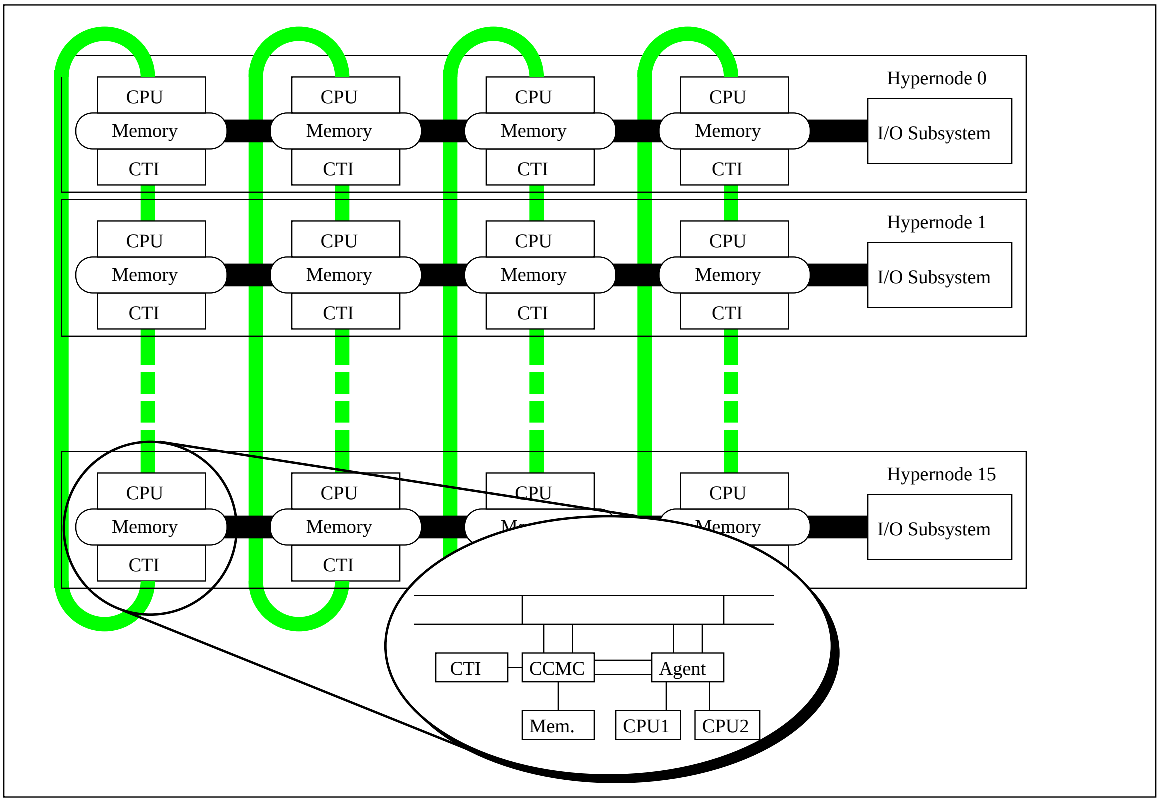


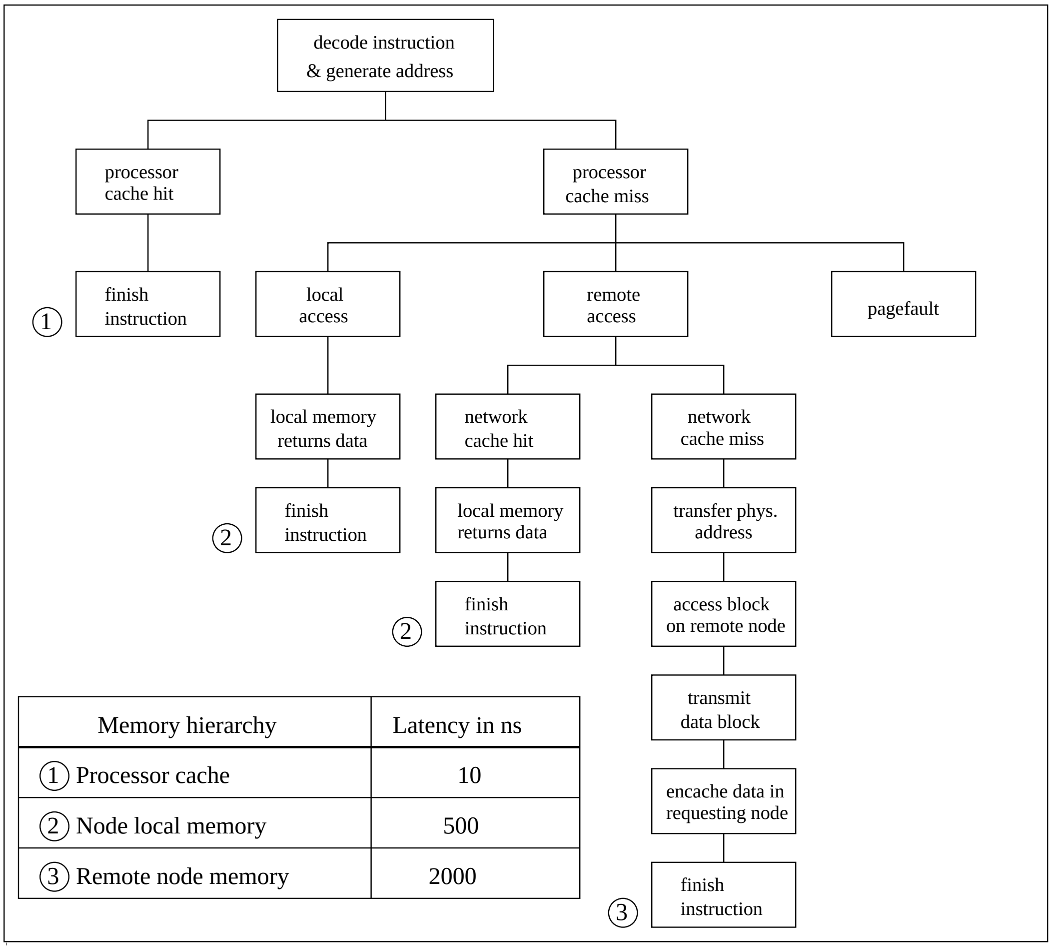
Figure 2: The Convex SPP Multiprocessor

All hypernodes are connected by 4 CTI-rings with a bandwidth of about 600MBytes/s. The resulting machine can be divided into several virtual machines, called subcomplexes. Each processor can access the whole memory in its subcomplex, but the memory latency depends on where the memory is actually located – either on the local hypernode or on a remote one. To reduce average access latencies, not only processor caches have been included, but also part of the local memory can be configured as network cache.

Figure 3 shows the various access paths and the resulting latencies on the Convex SPP. As can be seen, there are strong differences from 10ns in case of a processor cachehit, over 500ns in case of a local memory access up to 2000ns in the remote case. Compared to the processor's cycle time of about 10ns, it's obvious that the locality of data has a tremendous influence on the efficiency of computation.

In order to make efficient use of the machine's computational power, programs must be carefully designed to:

[2]**C**onvex **T**orodial **I**nterface



Memory hierarchy	Latency in ns
1 Processor cache	10
2 Node local memory	500
3 Remote node memory	2000

Figure 3: Memory Latencies on the Convex SPP

- reduce the average memory latency, i.e. try to access only local memory as far as possible, and to
- reduce the number of cachemisses.

Thread libraries structured as shown in figure 1 are not able to do this for several reasons:

- Accesses to central structures lead to frequent invalidation of processor and network caches, forcing other processors to fetch data from memory, which for most processors is located on remote hypernodes.
- Although the central runqueue guarantees perfect load balancing, there is no possibility to bind threads to specific processor. As a result threads migrate frequently and thereby loose their previous cache state. Each time they get rescheduled, they produce a lot of cache misses again.

- Moreover there are no means to locate threads near to their data. Therefore threads are usually urged to access remote memory, increasing the average memory access latency.

In this paper we propose two techniques that are essential for efficient userlevel scheduling and present a prototypical implementation of our ideas running on a CONVEX SPP 1000 multiprocessor.

3 Memory Conscious Scheduling

In contrast to UMA architectures on NUMA architectures like the Convex SPP there are different memory classes. Processors accessing memory experience latencies between 500ns for local memory and 2000ns for remote memory, although the whole mechanism for fetching memory across the CTI has been implemented in hardware.

To reduce the influence of this high access latencies in contrast to the processor cycle time of about 10ns on a SPP 1000, several caches have been introduced into the system. In addition to processor data and instruction caches, parts of the local memory can be configured as network cache to reduce the average latency when accessing remote memory. But nevertheless there is a gap between the processor's cycle time and this access time.

Several concepts have been developed to reduce the influence of cachemisses on the overall performance. On the one hand, there are special architectures like the HEP, MASE, Horizon, or Tera([AAC+92]). These architectures can deal with more than one instruction stream. If one instruction stream produces a cache miss, the processor can switch to another stream without having to wait for the end of the memory access. In addition, the Tera can have up to 8 outstanding memory operations, i.e. can start this memory accesses in advance to fetch data into the processor's caches.

On the other hand modern general purpose processors know a similar concept named *prefetching* and *poststoring*. Like the Tera processors, memory accesses can be started delayed or in advance. Despite possible main memory accesses the processor can continue as long as it doesn't really want to access the data.

Both, *prefetching* as well as *poststoring* depend on the compiler. It's the task of the compiler to generate code that exploits these capabilities. But even if the compiler supports this concepts the *Shared Memory* programming model in connection with heavy weight processes developed on UMA architectures couldn't efficiently be adopted to NUMA architectures. It's up to another programming model in connection with new scheduling strategies to weaken and/or solve the problems introduced by these architectures.

3.1 Userlevel Thread Libraries on NUMA Architectures

Compared to parallelization with heavy weight processes or kernel threads, userlevel threads offer new means to exploit information about the locality of data in the process of scheduling.

With coarse grain parallelization there was often no possibility to decide where to schedule a thread as it accessed memory almost everywhere in the system. As userlevel threads could be created for very short parts of computation that access only few memory regions the system has now the opportunity to place this threads near to their data and thereby reduce the average memory latency.

On the other hand, as userlevel threads usually need only small state information like tiny stacks and thread control blocks they are easily and cheaply migratable.The strategy to move threads to their data is in contrast to the traditional concept of bringing data near to processes by introducing several caches into the system.

Necessary for this migration is another property that e.g. on a Convex SPP cannot be fulfilled by ordinary kernel threads. In contrast to kernel threads, userlevel threads can usually be migrated to each node and each processor whereas kernelthreads are restricted to one node.

3.2 Using locality information for thread placement

To take profit from knowledge about the locality of a thread's data the user of a thread library, either programmer or compiler must have some possibility to assign threads to specific processors or nodes.

Such assignments can have two potential advantages:

1. If threads find most of their data in local memory instead of remote memory, the time for loading their working set will be reduced as each cachemiss will take less time.

2. If different threads will work with the same memory regions, it's beneficial to place them onto the same processor. As this increases the probability that a newly started thread will find some of its working set already loaded into the processor's cache, the average time of loading these working sets for this thread group will decline.

Necessary for this kind of placement is precise knowledge about the hardware structure, i.e. the number of processors and nodes and about the memory allocation policy applied by the operating system. For example on the Convex SPP a programmer can decide which memory class should be used. Based on this decision he can find out about the different locations and either he or the compiler can assign threads to this locations. This placement could either be done statically at compile time or dynamically on runtime.

3.2.1 Static placement at compile time

If the programmer or compiler know about the placement policy of the operating system they can predict where memory of a special class will be located. Especially for statically allocated memory, this will be possible. But also for dynamically allocated memory the location can usually be calculated, at least if the location of the allocating thread is known.

Based on this information the computation can be divided and distributed in a fashion that threads access only local memory.

Particularly for automatic parallelization of loops that use huge statically allocated, global data this kind of placement has proved to be very efficient([Mar93]). In case of manual parallelization the programmer has to calculate the memory locations by himself. But this evaluation can be very complex if the programmer uses dynamically allocated memory, and for memory classes like FAR_SHARED memory on the Convex SPP, which is spread over all hypernodes on a subcomplex. Therefore this strategy is not very useful, especially as all considerations have to be repeated each time the hardware structure changes.

3.2.2 Dynamic placement on runtime

With this strategy, locality calculations are done by the thread library instead of the programmer or compiler. Similar to the static assignment, the locations must be known. Either they can easily be calculated, based on the addresses of memory regions or the operating system must offer functions to translate addresses to locations. But still the information about addresses has to be made available, e.g. by the programmer.

The library's decisions now depend on whether the memory regions are already allocated when a thread is started or will be created later. If the memory regions are allocated in advance, the library can calculate the best original placement and start the thread there. This kind of assignment is particularly important for run-to-completion threads, because the thread library will have no further chance to reassign the thread to another processor.

If the memory regions are not known before the thread starts, if for example the thread will allocate this memory itself, it can only be started without locality considerations. But if it afterwards gets knowledge about data locality it can inform the thread library. The library can then decide whether it's worth to migrate the thread or not. Moreover this strategy can be applied if the thread's access behaviour changes, i.e. it starts to access other memory regions. As each migration imposes costs, e.g. in form of additional cachemisses, this strategy is only advantageous for threads that run for a long time and access lots of memory.

3.3 Prefetching of memory regions

As already mentioned, prefetching can be used to weaken the influence of memory latencies on the computation. Again information about data locality that has to be offered to the library is the basis for prefetching decisions. The library is then capable to prefetch memory regions before it actually switches to the thread. Besides the memory regions officially used by the thread, prefetching can also be applied to the thread's stack or its thread control block.

On NUMA machines like the Convex SPP prefetching can not only affect the processor's caches but also other parts, as for example the network caches. It depends on the cache coherency protocol whether prefetching into the network caches also affects processor caches. On the Convex SPP 1000 for example, prefetching into the network cache invalidates all copies of corresponding processor cache lines on all processors of the prefetching processor's hypernode. Better than distinct concepts for prefetching into different caches are mechanisms that allow prefetching over the complete memory hierarchy as implemented on the KSR1 of Kendall Square Research([Hau95]).

3.4 Binding threads together

Besides considering a thread's behaviour separately it's important to pay attention to communication between threads. If this communication is not implemented by messages but by shared memory, communicating threads should be located at least on the same hypernode. Otherwise even small amounts of communication memory, e.g. only a few bytes used for a mutex variable can cause a severe loss in performance.

This effect is due to the fact that the contents of communication memory continuously change and therefore the corresponding cachelines are frequently invalidated. As it is often not possible to reduce the number of invalidations without completely changing the algorithm the only influence a thread library can have is to collocate communicating threads near their communication memory. Thus, cachemisses are only to locale memory and the average access latency is reduced.

Similar to the placement strategies of the previous section, this collocation can either be implemented static or dynamic. With static binding, communicating threads are assigned to the same node or processor. But this binding gets lost, if due to load balancing one of the communicating processes is migrated to another location. This problem can be solved by dynamic binding. In this case threads are not bound to a specific location but are tied together, i.e. if one thread migrates, other threads will follow or the migrated thread will return.

This migration can either be done automatically by the thread library or triggered by the program that calls a library function and this functions will migrate the thread. As it's nearly impossible to automatically estimate the costs of migrating some thread, the second possibility will probably be the better solution.

Although the concept looks like ordinary coscheduling, algorithms of that area can hardly be used. Coscheduling for example uses information about the communication behaviour based on measurements of the number of exchanged messages or TLB-entries[3]([SW95]). All of this measurements are to expensive compared with the short lifetime of userlevel threads, especially when considering a possibly huge number of threads.

Instead of using TLB information access patterns to the library's synchronization primitives, e.g. mutex locks could be used to find out about communities between threads. It is simple to let each mutex lock hold a reference to the last thread that accessed it. At the next access, the library can decide whether one of these threads, either the old or the new one should be migrated.

[3]**T**ranslation **L**ookaside **B**uffer

3.5 Exploiting cache affinity

When a thread is executed by a processor, data is fetched into the different caches. So if it is rescheduled it should be reassigned to its previous processor to take advantage of probably remained cachelines.

On the other hand, if a thread migrated due to load balancing between different processors and its affinity values for these processors are known, the thread library can decide whether it should migrate it back to a former one. By binding threads to the processor to which they have maximum affinity, cache utilization is improved dramatically. But the influence depends on other thread attributes, like the frequency a thread blocks or the length of each scheduling period. The longer each scheduling period lasts the smaller this influence is, as at most the cachemisses for loading the working set at the thread's start can be saved by this technique. On the other hand this is a reason for its importance in the field of userlevel thread programming, as these threads should usually support short scheduling periods.

4 Realizing MCS in Thread Libraries

As the previous sections showed, for realizing memory conscious scheduling there is the need to assign and bind threads to processors or nodes, or to tie them together. Two common approaches exist to implement these functions.

4.1 Priority Based Mechanisms

Originally, memory conscious scheduling was used to improve the standard scheduling mechanisms in ordinary operating systems. These systems mostly use central run- and sleepqueues. Without any further attempts, threads get randomly assigned to the available processors.

To implement MCS without having to rewrite wide areas of code the existing priority mechanism can be used. Instead of each thread having a fixed priority that's usually calculated based on its processor usage and the system load, now this priority also depends on the processor. The higher a thread's priority on one processor is, the tighter it is bound to this processor.

With this technique the user can specify very precise where his thread should be calculated and how tight this binding should be. Figure 4 shows different classes of binding that would be possible on a Convex SPP with 32 processors on 4 nodes:

- Threads 1 and 2 have very high priority to one specific processor. It's nearly impossible for them to migrate to another processor or node. This is especially advantageous if the user wants to tie two communicating threads together and assign them to one node.

- Thread 3 has equal priority on each processor of one node. Therefore it can freely migrate between processors on this node.

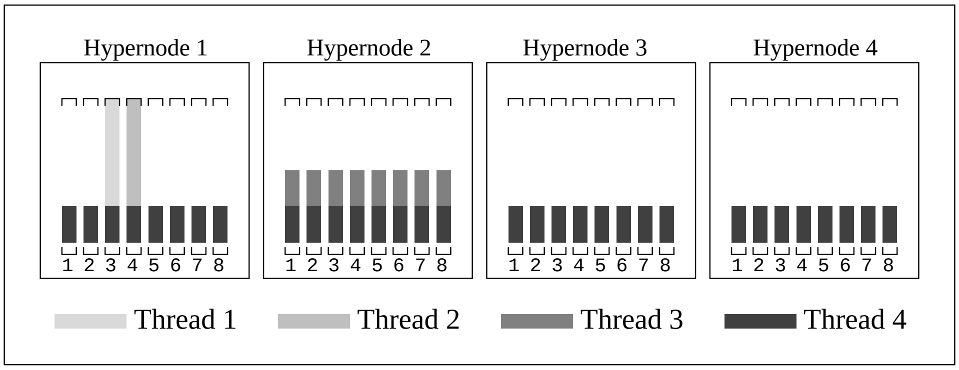


Figure 4: Different levels of binding

- Thread 4 has equal priority on all processors in is therefore not restricted in migration.

Tucker ([Tuc93]) used priorities to express a process's affinity to a processor by dynamically lifting the priority for the processor on which this process has been executed most recently. The last process that has been executed by a processor experiences an additional lifting. This leads to a simple binding of processes to processors and to an increase in the effective time slice for each process.

Because of its simplicity this mechanism can be added to current schedulers, but there are several reasons why it can't be used efficiently with userlevel threads or on modern NUMA architectures.

First, in these mechanism complex structures are necessary to find out about the thread with the highest priority without having to search and recalculate the priority of each available thread. Second, the usual implementation of having one central runqueue is a bottleneck in a multiprocessor. In connection with the long timeslices of processes in operating systems, these bottleneck had less influence. But in applications with lightweight threads that block frequently central structures have to be avoided as the probability for contention increases with the frequency of access.

4.2 Queuebased Mechanisms

To avoid the bottleneck of central structures, scheduling structures have to be distributed over the whole system. Thread libraries are a good means to dynamically adopt to the hardware structure of a machine. Figure 5 shows the distribution of queues according to the structure of a Convex SPP with 32 processors and 4 nodes. Local queues for each processor enable to bind threads to this specific location. This local queues offer two advantages:

1. As priorities are no longer necessary, simple structures can be used. Each access to the queue will therefore be fast compared to an access to the complex structures of a central priorityqueue.

2. Except for load balancing these structures are only accessed by the local processor, so there is no contention and only accesses to local memory or the processor's caches.

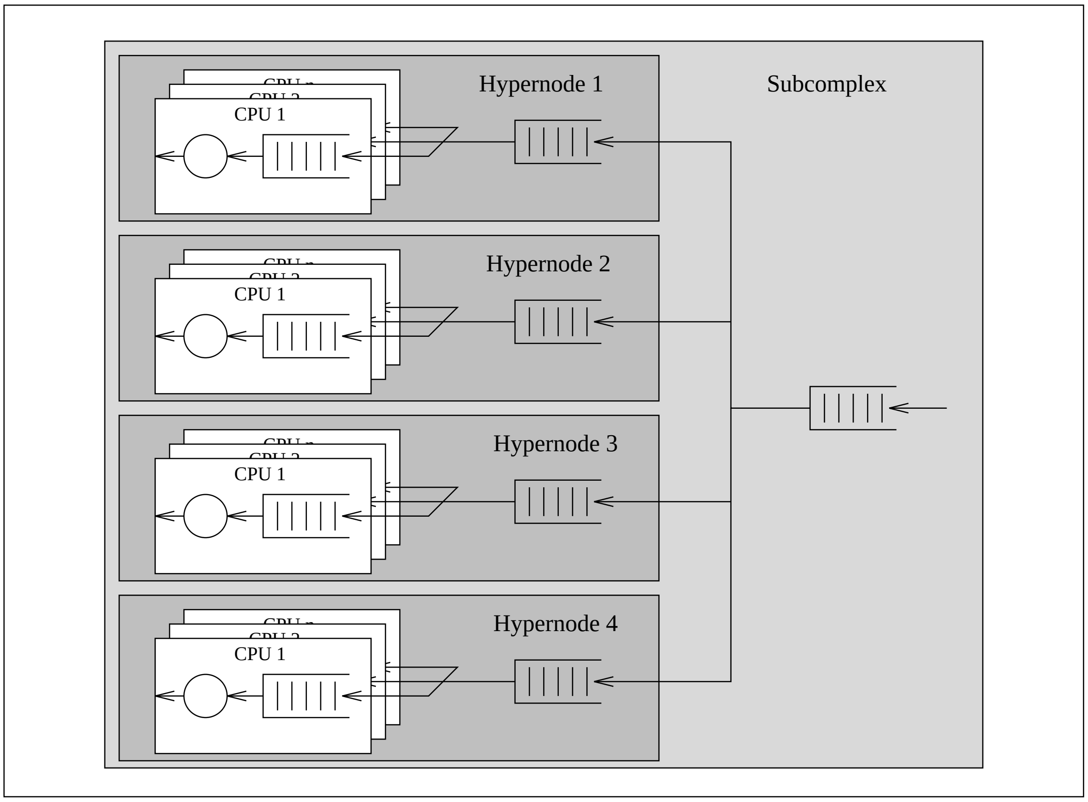


Figure 5: Distributed scheduling structures

If a processor finds its runqueue empty it will proceed with its node's runqueue, such preferring threads bound to the node to threads in the global subcomplex runqueue that have no binding.

These simple mechanism conserves assignment decisions based on locality information as far as possible and thereby a thread's affinity too. Each thread will be placed on the local runqueue of the processor on which it was executed most recently, resulting in a behaviour similar to that experienced by Tucker.

4.3 Local freelists

Similar to local runqueues other structures used in a userlevel thread library, e.g. scheduling structures or memory regions used for synchronization can be managed distributed. For example it may be useful to hold memory regions, e.g. for stacks in a local free list instead of freeing them after use. This not only saves time but also preserves the information that these structures are either allocated locally or at least have been used locally.

One drawback of local freelist is the necessity to balance between different list. Otherwise it would be possible that one processors always allocates new structures whereas other processors only collect unused structures. To avoid these situation usually a central pool is introduced, where processors can request for new structures or can give back unused structures. A common algorithm used for these central pools is:

- Each local freelist can hold up to M structures.
- If the number of structures in a local freelist exceeds this limit, $M/2$ structures are transfered to the central pool.
- If a processor request a structure and the local freelist is empty, it fetches $M/2$ structures from the central pool.

With the parameter M it is possible to adopt the mechanism to different situations from having equally balanced freelist at the cost of frequent balancing or vice versa.

4.4 Delayed thread start

Applications based on the thread programming model usually employ a huge number of threads. As these number normally exceeds the number of processors available, not all threads will be running simultaneously. But moreover not all threads created by the application must actually be started. As long as the computation doesn't depend on a newly created thread it's enough to store the information that is necessary for the real start. These technique has several advantages:

- In contrast to a completely instantiated thread the memory used to hold the parameters for the thread's start consist of only a few bytes.
- If the newly created thread will be assigned to another processor or will eventually migrate to another processor, it is preferable to let this processor allocate the memory for this thread as it's often not possible to create memory local to another processor.
- By distinguishing between *startable* and *runable* threads there is a clear distinction between threads having few or none affinity to a processor and threads having considerable affinity to a processor. This can significantly simplify load balancing.

Another advantage is given in connection with local freelists. If the thread library starts threads only if no other thread is runable, there is a high probability that another thread has exited or is currently exiting. Now the new thread can at least reuse the exited thread's structures and thereby adopt the affinity of these structures to the processor's caches. Certain thread packages even allow the new thread to use the structures of a currently exiting thread. This technique is known as *Continuation Passing* and usually assures lowest overhead and best cache reusage.

5 Affinity Scheduling

While memory conscious scheduling deals with the question of where to schedule threads, affinity scheduling tries to give an answer to the question when to schedule a thread. The goal of affinity scheduling is to find out in which order threads have to be scheduled to minimize the number of resulting cachemisses.

This section will present a few common methods used in this area as well as a novel approach to estimate a thread's affinity to the processor's caches and an approximation for the number of cachemisses when a thread gets restarted.

5.1 Influence of cache affinity on the runtime

Each cachemiss caused by a memory access results in an unnecessary delay to an application. As cachemisses take at least one order of magnitude more time than ordinary processor cycles on modern architectures it is obvious that the application programmer has to take care that his calculation causes as few cachemisses as possible. On the other side the scheduling algorithm has the responsibility to schedule threads at the right locations and in the right order.

Cachemisses can be divided into two different classes. First there are cachemisses necessary to load a thread's working set into the processor's cache. These working set consists of memory used for thread management like the thread control block and the memory used to hold the thread's data, like its stack or its data segments. The delay to load this working set happens each time a thread is started or restarted after it has been blocked. The shorter the thread will run after loading its working set, the higher the influence of this delay to the overall performance will be. As userlevel threads should support short scheduling periods the scheduler has to think of how it is possible to reduce this *reload transient*.

While a thread is running, usually additional cachemisses occur. These cachemisses are due to the invalidation of cachelines by other processors or to a change in memory access behaviour of the running thread. These cachemisses can't be predicted by the scheduling algorithm and therefore usually can't be avoided, although they can be used to find out about the new working set.

In contrast to this *transient* cachemisses, the first category can be influenced by a scheduling algorithm, especially for threads that block and continue frequently. If the scheduler knows that most of the working set of a currently deblocked thread is still resident in the processor cache it can prefer this thread by assigning it a higher priority. If it will be started as early as possible, it will experience fewer cachemisses than other threads having less cache affinity.

5.2 Affinity measures

In contrast to the information gathered for memory conscious scheduling, affinity information can't be calculated or accessed directly. Although more and more architectures allow cachemiss measurements there is currently no possibility to get information about the number of valid cachelines of an application's threads directly. Therefore one is restricted to approximate this cache state. Two different classes of affinity measures can usually be figured out:

1. Static measures: The affinity value and the resulting priority are calculated only when a thread blocks or deblocks, but the calculated value will remain constant until the thread will get restarted.

2. Dynamic, time dependent measures: Like static values, these values are calculated when a thread blocks or deblocks. But moreover these values age, either by periodically changing the value or the value's interpretation. The second method is especially interesting if the priority order in the runqueue remains constant over time, i.e. if for some time t the relation $P_i(t) < P_j(t)$ is true for two threads T_i and T_j, it will also hold for all $t + \Delta t$. In this case a thread's affinity value has only be interpreted and compared to the values of other threads in the runqueue when it gets inserted into this queue.

5.2.1 Timebased affinity models

The simplest models can be derived from timing information that is available on all architectures. These models don't need special hardware support and are therefore interesting for older or cheaper machines that have no possibility to gather cachemiss information.

Virtual time: While a thread is blocked other threads running on its previous processor will throw away parts of its cache state. The more other threads ran until the thread gets rescheduled, the more cachelines will have disappeared. Particularly for applications with equal threads, like those generated by automatic parallelized loops, the inverse number of intervening threads can give a good approximation for the number of valid cachelines in relation to other threads.

Each processor therefore gets a simple counter that is incremented on each scheduling decision. Every time a thread releases its processor it adopts its processor's virtual time and uses it as priority when it is placed on this processor's runqueue again. The higher this value is, the fewer intervening threads have been served by its processor.

Virtual time is a typical dynamic affinity model, as the interpretation of the affinity value changes over time but fortunately the priority order remains constant. As it requires no specialized hardware and only minimal software overhead but results in astonishing performance improvement, it's the first choice for machines with no or only time consuming possibilities to count cachemisses.

Real time On architectures offering high resolution hardware clocks the scheduler can use this time information instead of virtual time. But if the clocks aren't synchronized across the whole machine there is no increase in information. The priority order derived from this information will not differ from that based on virtual time. Only if all processors share the same global time, realtime offers higher potential as now threads of different processors can be compared. This can for example be necessary to find the thread with lowest affinity when load balancing is required.

Both methods, virtual time as well as real time are well suited for threads with equal attributes. On the other hand both methods ignore any differences, e.g. in runtime or access behaviour. In order to consider such differences real time can be used in a similar fashion as in ordinary operating systems. In addition to the time a thread is started or stopped, its runtime is evaluated. As threads that have run for a long time can be expected to have nearly all of their working set loaded into the processor cache, these threads should be preferred against others. But pure runtime is not enough, as it neglects that while the thread has been stopped parts of its working set will have been displaced by other threads. Therefore some aging strategy has to be introduced to take this time into consideration when calculating the thread's priority.

Aging strategies are already used in operating systems, like the Silicon Graphics IRIX. They assume a linear relation between a thread's runtime and its affinity. Only it the thread's runtime exceeds a minimum time, this affinity value is used for scheduling. On the other side, if the time a thread hasn't been running exceeds a certain limit, its binding to a processor is released([BB95]).

Another simple method is to divide a thread's runtime by the time it has already been blocked and use this value as its current priority, i.e. for a thread T_i, started at $t = start_i$ and stopped at $t = stop_i$, the priority at some time t evaluates to:

$$P_i(t) = \frac{stop_i - start_i}{t - stop_i} \tag{1}$$

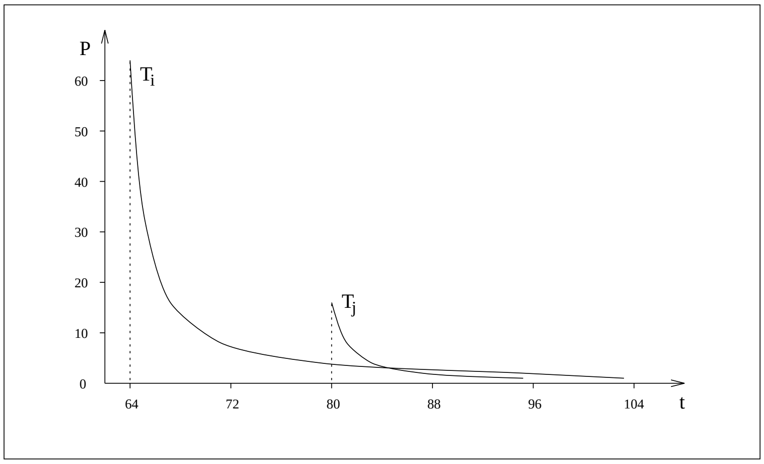


Figure 6: Hyperbolic aging of priority values

Figure 6 shows the priority of two threads. Thread T_i ran 64 time units followed by thread T_j which ran 16 time units. As the figure already suggests, this kind of aging strategy has a serious drawback. The two hyperbolas expressing the priorities can intersect. As the point of intersection t_s:

$$t_s = \frac{stop_j(stop_i - start_i) - stop_i(stop_j - start_j)}{(stop_i - start_j) - (stop_j - start_j)} \tag{2}$$

could be later than $stop_j$ the order induced by this priority value can change with time.

This effect can be avoided by another class of aging strategies. Here the scheduler uses exponential aging like

$$P_i(t) = \frac{stop_i - start_i}{K^{a(t - stop_i)}} \tag{3}$$

With the two parameters $K > 1$ and $a > 0$ different forms of aging can be implemented. Using exponential aging it's enough to calculated once whether a thread has a higher priority than another as this order couldn't change later on. Figure 7 shows the two

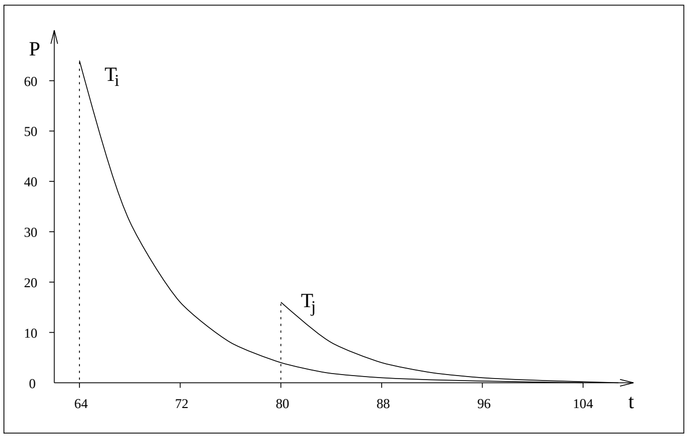


Figure 7: Exponential aging

threads again with exponential aging ($K = 2$, and $a = 0.25$). As the figure suggests the order can e.g. be evaluated at the time thread T_j stops and this order will remain constant for all time t:

$$\begin{aligned} P_i(t) < P_j(t) \quad &\leftrightarrow \quad \frac{stop_i - start_i}{K^{a(t - stop_i)}} < \frac{stop_j - start_j}{K^{a(t - stop_j)}} \\ &\leftrightarrow \quad \frac{stop_i - start_i}{K^{a(-stop_i)}} < \frac{stop_j - start_j}{K^{a(-stop_j)}} \\ &\leftrightarrow \quad \frac{stop_i - start_i}{K^{a(stop_j - stop_i)}} < stop_j - start_j \\ &\leftrightarrow \quad P_i(stop_j) < P_j(stop_j) \end{aligned} \tag{4}$$

Therefore, a simple priority queue is still sufficient to implement this aging strategy. When a thread has to be enqueued onto the runqueue and is compared with other threads in the runqueue the priorities of all threads that are concerned have to be recalculated. To improve performance a structure has to be used for the runqueue that needs as few comparisons as possible.

5.2.2 Simple cachemiss based affinity models

To get more information about cache usage modern architectures have been instrumented with *performance monitors* to measure the number of cachemisses and the average cachemiss latency. This can either be done by the processor or as with the Convex SPP 1000 by additional hardware.

Although the number of cachemisses is now available no information is accessible about the number of cachelines a thread owns in its processor's cache. Still more or less complex models have to be used to approximate this amount.

One idea is to use the number of cachemisses caused by a thread as measure for its future behaviour, i.e. a thread that experienced only few cachemisses is expected to produce few in its next scheduling period as most of its working set should still be resident in the processor's caches. In addition to using only the cachemisses of the thread's last scheduling period, the scheduler can also use some or all of the last runs to evaluate the thread's priority.

Although these strategy seems to handle threads correct, it is useless on a heavily loaded machine and for threads started for the first time, as it completely ignores the influence of other threads and the runtime of the observed thread. For example new threads will experience a huge number of cachemisses as they have to load their complete working set. Now the scheduler will assign a very low priority to these threads. As a result all of the working sets will be swapped out when the threads get rescheduled and a lot of cachemisses will be produced again.

Therefore one might intend to use the opposite order, i.e. to expect that threads that experienced a lot of cachemisses have loaded nearly all of their working set into the caches. But now the decision to give a thread low priority due to few cachemisses neglects that this might be the result of the fact that the thread has nearly all of its data in its processor's caches.

Instead of using only the number of cachemisses of the last scheduling periods, the scheduler can include a thread's history into its decisions. It can use the information gathered in former scheduling periods, possibly applying some additional aging strategies.

5.3 Mathematically modeling cache affinity

All models mentioned so far used intuitive assumptions how the number of cachemisses counts for a thread's affinity and should be converted into some priority. None of these models considered the influence of the cachesize, the number of threads, their access behaviour, etc.

In 1987 Thiebaut and Stone published a mathematical model for caches that enables to calculate the so-called *Reload Transient*, i.e. the time a rescheduled thread will have to wait until its working set has been completed again([TS87]). They used this working set, called a thread's *footprint* and calculated the reload transient assuming that cachemisses are independent and uniformly distributed. Using traces they found good correspondence between their restrictive model and the behaviour of real processes.

The drawback of using these investigation for scheduling, as has been suggested in [SL89] is, that the scheduler has to know about the size of footprints in advance.

Based on these two considerations we developed another cache model that enables us to get information about the valid number of cachemisses a thread owns in its processor's cache at any moment.

5.3.1 Increase of affinity while a thread is running

Our model has been tailored for the type of caches used in the Convex SPP 1000: *virtually indexed, direct mapped* caches. If a running thread causes a cachemiss, two different cases can occur:

1. The fetched data replaces another thread's cacheline and therefore increases the number of valid cachelines for the current thread by one.

2. The fetched data replaces a cacheline of the currently running thread. Its cache affinity remains constant.

With the same assumptions as Thiebaut and Stone and knowledge about the cachesize the cache state can easily be modeled by a Markov chain. Each number of valid cache lines corresponds to one state in this Markov chain. Figure 8 shows this chain with its transition probabilities. If a thread is in state C, the probability that one cachemisses increases the cachestate to $C+1$ is $(N-C)/N$ and the probability that it remains constant is C/N.

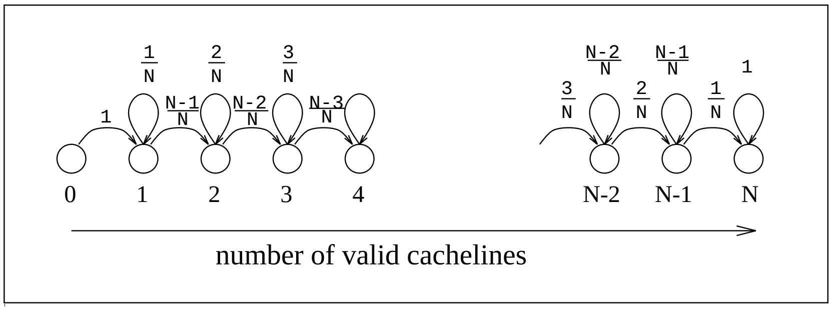


Figure 8: Increasing cachestate of a running thread

Now this Markov chain can be represented by its transition matrix M:

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & & 0 \\ 0 & \frac{1}{N} & \frac{N-1}{N} & 0 & \cdots & & 0 \\ 0 & 0 & \frac{2}{N} & \frac{N-2}{N} & \cdots & & 0 \\ \vdots & & & & \ddots & & \vdots \\ & & & & & \frac{N-1}{N} & \frac{1}{N} \\ 0 & & & & \cdots & 0 & 1 \end{pmatrix} \tag{5}$$

Each element $m_{i,j}$ is the probability that a cachemiss increases a thread's cachestate from i to j cachelines. Raising the matrix to the n-th power gives the probability for an increase from i to j after n cachemisses, where each element of M^n is of the form:

$$m^n_{i,j} = \begin{cases} 0 & : \; j < i \\ \frac{(N-i)!(-1)^{j+i}}{(N-j)!(j-i)!N^n} \sum\limits_{k=0}^{j-i} \binom{j-i}{k} (k+i)^n (-1)^k & : \; i \leq j \leq i+n \\ 0 & : \; i+n < j \end{cases} \tag{6}$$

We used this probabilities to evaluate the expected number of currently valid cachelines if we started with a known number i and experienced n cachemisses:

$$\begin{aligned} E^n_i &= \sum_{j=0}^{N} j m^n_{i,j} \\ &= \frac{(N-i)!(-1)^i}{N^n} \sum_{j=i}^{i+n} j \frac{(-1)^j}{(N-j)!(j-i)!} \sum_{k=0}^{j-i} \binom{j-i}{k} (k+i)^n (-1)^k \\ &\vdots \\ &= N - (N-i)\left(\frac{N-1}{N}\right)^n \end{aligned} \tag{7}$$

5.3.2 Decrease of cache affinity while a thread is waiting

A similar Markov chain can be used for the time a thread is waiting, either in the runqueue or in some sleepqueue. Figure 9 shows this model. The waiting thread looses cachelines every time the currently running thread replaces one of the waiting thread's cachelines with an own one.

In the same fashion as with the increase of cache state, this decrease can be calculated. The probability that a thread will have j remaining cachelines when it stopped with i cachelines and n cachemisses occurred since it stopped, is given by :

$$m^n_{i,j} = \begin{cases} 0 & : \; j < i-n \\ \binom{i}{j} \sum\limits_{k=j}^{i} \binom{i-j}{k-j} \left(\frac{N-k}{N}\right)^n (-1)^{k+j} & : \; i-n \leq j \leq i \\ 0 & : \; i < j \end{cases} \tag{8}$$

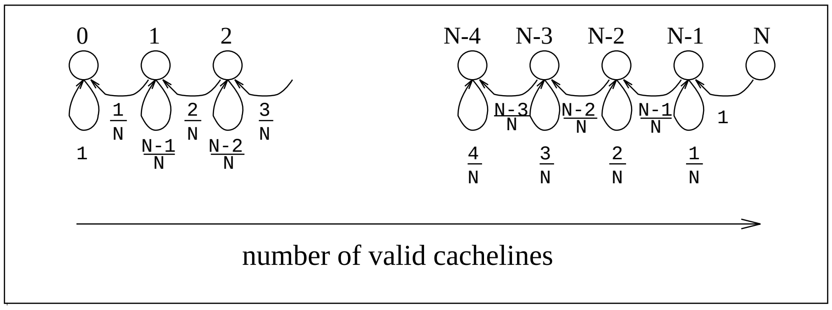


Figure 9: Decreasing cachestate of a waiting thread

Based on this result the expected number of valid cachelines can be expressed by:

$$E_i^n = i\left(\frac{N-1}{N}\right)^n \tag{9}$$

5.3.3 The reload transient

Similar to the methods used by Thiebaut and Stone both functions need to know how many valid cachelines a thread owns. But these values can be approximated by successively applying the two functions.

When a thread starts it is assumed to have no working set resident in its processor's caches. Starting from an empty cache state it will fill the caches until another thread gets scheduled. Based on function 7 its expected current cache state can be calculated. From now on in each moment its remaining cache state is given by function 9 with i set to the previously calculated value.

If the thread will be rescheduled it will probably complete its working set again, starting with the remainder of its last scheduling period. If we evaluated the footprint of thread i to be E_i after its last run and $n_i(t)$ cachemisses occurred up to the time it gets rescheduled, the number of cachemisses it will probably produce is of the form (using the constant $K = (N-1)/N$):

$$R(i, n_i(t)) = log_K \frac{N - E_i}{N - K^{n_i(t)} E_i} \tag{10}$$

The simplest form to use this reload transient is to give the highest priority to the thread with the lowest reload transient. This prefers threads that have most of their working set resident in the caches and will therefore produce few cachemisses. Moreover, preferring these threads, fewer cachelines of other threads will be replaced, i.e. other threads will get less disturbed as if we choose a thread with a higher reload transient. The priority relation between two thread therefore has the form:

$$P_i(t) > P_j(t) \leftrightarrow R(i, n_i(t)) < R(j, n_j(t)) \tag{11}$$

Again this is a dynamic affinity measure as the values change with time. Similar to the exponential aging strategies it can be shown that the order induced by this condition remains constant. The reload transient must therefore be calculated only once, e.g. if the thread gets placed on the runqueue. Important for the order in the runqueue is not the real reload transient that will arise at the time of rescheduling the thread, but only its relation to the current reload transients of all other threads in the runqueue.

5.4 Structures supporting affinity scheduling

The data structures presented next serve as a refinement of the coarse one shown in the section for memory conscious scheduling. They are intended to supplement the distributed structures to efficiently support affinity scheduling

In contrast to simple memory conscious scheduling for affinity scheduling we need priority queues again to express the intended influence of affinity on the scheduling order. These runqueues must have certain properties:

- For minimizing the necessary work when a new thread gets enqueued, the number of comparisons has to be minimized. This is crucial for dynamic affinity models as e.g. the reload transient.
- Finding and dequeuing the element with the highest or for load balancing lowest priority must be as cheap as possible.
- Elements with the same priority should perhaps be scheduled in a known fashion like FIFO or LIFO.
- Especially in systems with a changing number of virtual processors runqueues must be dividable and mergable in a simple and efficient way.

A number of structures and algorithms has been developed in recent years for implementing efficient priority queues. They can be coarsely divided into several classes:

List based structures: Although list based structures are inefficient for implementing large queues they can be useful for small queues. Brown showed that linear lists are superior to all other structures up 5 elements([Bro77]). If the queue exceeds this limit it can be divided into several sublists each of which responsible for another priority range. These sublists are generally used in UNIX systems. For affinity scheduling these sorted linear lists have the advantage of low overhead when accessing theh elements with highest or lowest priority.

Skiplists: While elements in sorted arrays can be found in $O(logN)$ using binary search, linear lists can't be used like that. To simulate this search method, skiplists can be used. Although searching is faster in skiplists compared to ordinary lists, their structure is fixed. Each time an element is deleted, the whole list has to be restructured resulting in high costs.

Tree based structures: These structures allow access to all elements with average costs of $O(logN)$. To guarantee this value care has to be taken to balance the tree. Otherwise the tree could degenerate to a sorted linear list. If trees are used as runqueues and only the first or the last element dequeued, simple restructuring techniques can be used to reduce the probability of degeneration.

Heapbased structures: In contrast to tree based structures, heap based structures cannot degenerate. They guarantee access costs of $O(logN)$. But only the element with the highest priority can be accessed directly. The opposite one can only be found by searching through the whole structure.

6 Synchronization mechanisms

Besides scheduling, lightweight synchronization mechanisms are a second important part of userlevel thread libraries. Again, kernel mechanisms are to time consuming, as each time the address space has to be changed, resulting in a loss of cachelines and TLB entries.

Despite the fact that synchronization mechanisms that are executed on userlevel are more efficient they usually have to communicate with the operating system in some form. Otherwise the operating system's scheduler could preempt one of the virtual processors, while the userlevel thread executed by this processor is inside a critical section and is therefore holding a lock.

A second problems becomes additionally important on large NUMA architectures. Synchronization algorithms as well as data structures used for them must be optimized for these architectures. Otherwise they could produce lots of cachemisses and significant delays.

6.1 Waiting mechanisms

All synchronization mechanism have in common that in certain cases threads have to wait on some condition. The implementation of this *waiting* is therefore important for all these mechanisms. Usually there are two different choices: either the thread spins actively until the condition is satisfied or it blocks. In practice, either these two or intermediate solutions are used for different reasons

6.1.1 Always Spin

This mechanism waits actively for some condition to be satisfied. As it can immediately continue after e.g. the lock has been released, it has the lowest latency of all possible mechanisms. Moreover, as the thread stays active, most of its cache state should remain unchanged. Nevertheless this mechanism has some drawbacks:

- Processor cycles are wasted.

- If the virtual processor serving the thread that holds e.g. some lock has been preempted by the system scheduler, all other threads waiting for this lock to be released would block the remaining virtual processors.

- If more userlevel threads exist than virtual processors a deadlock can occur, if the thread that holds the lock is not running, but all virtual processors are spinning.

- Depending on the architecture, the spinning can cause a high load on the system bus, eventually slowing down all other processors.

6.1.2 Always Block

This strategy is the opposite of always spin. Each thread that has to wait immediately blocks, i.e. it releases its processor and places itself on some sleepqueue. If the condition becomes true, it will be dequeued from this sleepqueue and placed on the runqueue or rescheduled at once. Although no processor cycles are wasted for spinning, two context switches are necessary and there is a higher probability for a loss of cachelines.

If e.g. a lock gets released, either one of the waiting threads or all can be deblocked and will concurrently try to acquire the lock. Even if only one thread is deblocked, it is not sure that it will be able to acquire the lock as long as other threads are running in parallel. In this case the thread will have to block again, resulting in additional context switches. Despite the costs for context switches some thread libraries don't use sleepqueues at all, but enqueue blocked threads immediately on the runqueue. With this technique, called *switch spinning* threads will try to acquire a lock again and again.

Instead of blocking a thread a second time, one might try to reserve the lock for a deblocked thread. But especially on system which could not guarantee that the thread will be started at once, i.e. without preemption, no other thread would be able to access this lock.

6.1.3 Spinblocking

Spinblocking combines the advantages of always spin and always block. If a thread has to wait for a lock it first spins for a while. After this period, if the condition is still not satisfied it blocks. Usually the time a thread spins actively is chosen in the order of time for a userlevel context switch.

If the time a thread will have to wait for the condition to become true is known in advance, a refined algorithm can be used. If the waiting time exceeds a certain limit, e.g. the time for a context switch, the thread will not spin but block immediately, saving the processor cycles that otherwise would have been wasted. On the other side, if the expected waiting time is lower than the limit the thread will spin, but at most for a certain time. If the condition hasn't been satisfied up to this moment it will block.

The expected time, that mostly depends on the purpose for which the synchronization mechanism is used, could often be approximated by the history. For example, the time a lock is held by threads is measured and used to calculate an average time. Using atomic instructions like `test_and_set` or `load_and_clear` a lock could then be implemented in the following manner:

```
void acquire_lock(lock_t *lock)
{
   int count;

   while(! TestAndSet(lock->addr))
   {
      if(lock->MeanTime > Threshold)
         block();
      else
      {
         count = 0;
         while((count < MaxSpins) && !TestAndSet(lock->addr))
            count++;
         if( count == MaxSpins)
            block();
         else
            break;
      }
   }
   lock->starttime = current_time();
   return;
}

void release_lock(lock_t *lock)
{
   lock->MeanTime = evaluate_meantime(lock->starttime, current_time());
   *lock->addr    = 1
}
```

6.2 Spinlocks for atomicity

All blocking mechanisms rely on the property that it is possible to check a condition and enqueue onto a sleepqueue in one atomic step. If the architecture doesn't provide instructions that are capable to do this, spinlocks have to be used. The drawback of wasted processor cycles is less important, as usually this type of critical sections is very short.

The implementation of spinlocks relies on atomic instructions like `ldcws`(load and clear word short) of the HPPA-RISC family, `ldstub`(load and store unsigned byte) of SPARC processors or the `xchg`(exchange) instruction of Intel processors. All of these instructions have in common that they are able to read and modify a memory cell in an indivisible step. If the datum resides in a cache or is fast accessible like on UMA architectures the exclusive access doesn't influence other processors. This is no longer true for NUMA architectures like the Convex SPP for several reasons:

- All atomic accesses to the same datum have to be serialized by the hardware, i.e. if a lot of processors compete for the lock, the only one that could release it might be delayed too. This can artificially lengthen the critical section.
- All of these instructions really change the datum, whether they are able to set the lock or not. As a result, all copies in the caches of other processors are invalidated and therefore all other processors will experience cachemisses.

As the problems mentioned so far led to unacceptable performance penalties even on architectures with a moderate number of processors, several improved techniques have been developed in the last few years([ALL89], [And90], [KLMO91], [MCS90]).

6.2.1 Simple spinning

Only for completeness, the following lines show the algorithm to set and release locks with simple spinning. This and all following algorithms rely on the following convention: locks that are set have the value zero, locks that are released an arbitrary other value.

```
/* acquire lock */
while(fetch_and_clear(lock_position)==0);

/* critical section */

/* release lock */
*lock_position = 1;
```

6.2.2 Spin on read, snooping locks

This method uses a `read` instruction for spinning. Only if the lock is free, the thread attempts to acquire it by an atomic instruction. On architectures with caches these read operations work locally, without disturbing other processors. Moreover as there is usually no need for a serialization of read operations, the release operation will not be delayed.

```
/* acquire lock */
while((*lock_position==0) || (fetch_and_clear(lock_position)==0));

/* critical section */

/* release lock */
*lock_position = 1;
```

Nevertheless, the method has drawbacks that makes it unacceptable for large machines. When the processor owning the lock releases it, it will write into the shared memory cell. All other processors will experience a cachemiss in their read-loop and will find the lock released. As a result all of them will try to acquire the lock with the same problems as with simple spinning. Although only one will succeed and all other will return to their read-loop, about $O(n^2)$ cachemisses will be caused by the atomic operations if n processors compete for the lock.

6.2.3 Backoff spinning

The problem of snooping locks was produced by the implicit synchronization caused by the cachemiss in the read-loop. *Backoff spinning* tries to reduce the probability for this synchronization by introducing artificial delays between consecutive read operations.

```
/* acquire lock */
while(fetch_and_clear(lock_position) == 0)
{
   while(1)
   {
      while(*lock_pos == 0);
      delay(waiting_period());
      if(*lock_pos != 0)
         break;
   }
}

/* critical section */

/* release lock */
*lock_position = 1;
```

The values used for the delay can either be calculated or chosen randomly. Practice showed that exponentially growing delays are well suited to reduce the number of cachemisses. To prevent very long delays, an additional upper limit can be introduced. Investigation, e.g. by Anderson([And90]) showed that exponential backoff spinning scales very well even on large machines.

6.2.4 Ticket locks

Especially on NUMA architectures each cachemiss can lead to an access to remote memory. One method that avoids unnecessary atomic operations and the cache invalidations caused by these operations are *ticket locks*. The lock now consists of two memory cells. One contains a ticket counter. Each thread that wants to acquire the lock draws a ticket by atomically incrementing this counter (`new_ticket`). Then it waits until the other memory cell (`served`) reaches the same value. That's the sign that it has successfully acquired the lock. To release the lock, the second memory cell is incremented:

```
/* acquire lock */
my_ticket = fetch_and_inc(new_ticket);
```

```
while(*served != my_ticket);

/* critical section */

/* release lock */
(*served)++;
```

6.2.5 Queued locks

All methods mentioned so far used one shared memory cell for spinning. But with this restriction there is no possibility to reduce the number of cachemisses further. Increasing the number of competing processors also at least linearly increases the number of cachemisses for each synchronization operation.

One technique that uses different memory cells for spinning is known as *queued locks* ([MCS90]). To implement this method the underlying architecture must provide more complex atomic instructions:

- `fetch_and_store(p,i)`: exchanges the value referred to by the pointer `p` by the value of `i` and returns the old value.
- `compare_and_store(p,c,i)`: compares `c` to the value referred to by the pointer `p` and if both are equal, replaces the old value of the memory cell by `i` and returns *true*. Otherwise the memory cell remains unchanged and the value *false* is returned.

Each attempt to acquire the lock is now represented by a separate structure `I`, built up of a pointer to the next of these structures in a queue of attempts and a private variable used for spinning. The lock itself is only a pointer (`Lock_ptr`) to such structures.

As the acquire and release operations are more complex they have been covered in two functions:

```
/* structure for locking attempts*/
struct
{
  int            wait;
  struct node_t *next;
}node_t;

void acquire_lock(struct node_t *Lock_ptr, struct node_t *I)
{
  node_t *old;

  I->next = NULL;
  old = fetch_and_store(Lock_ptr,I);
  if(old != NULL)                /* sleepqueue was not empty              */
  {
    I->wait   = 1;
```

```
    old->next = I;              /* enqueue I on sleepqueue             */
    while(I->wait);             /* wait until lock is released         */
  }
}

void release_lock(struct node_t *Lock_ptr, struct node_t *I)
{
  if(I->next == NULL)          /* there is no successor                */
  {
    if(compare_and_store(Lock_ptr, I, NULL)) /* dequeue structure      */
      return;
    else                       /* another thread is waiting for the lock */
      while(I->next == NULL);  /* wait until it is enqueued correctly    */
  }
  I->next->wait = 0;           /* release lock                         */
}
```

6.3 Datastructures for sleepqueues

Similar to runqueues in thread libraries sleepqueues have to be designed for low overhead, a high degree of possible parallelism and locality.

6.3.1 Central sleepqueues

In general purpose operating systems central sleepqueues are used for blocked threads. Each thread is characterized by a value that represents the synchronization mechanism that caused the thread to block. Using this value threads waiting for this mechanism can be found and deblocked. Figure 10 shows the structure of a system using one central sleepqueue.

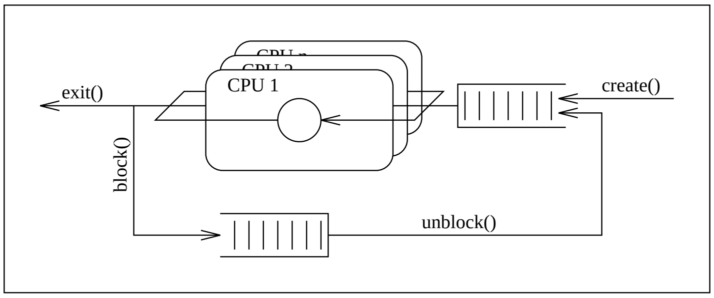


Figure 10: Using one central sleepqueue

Like listbased runqueues these sleepqueues are usually divided into several lists and threads are hashed into one of these lists. This improves parallelism as well as it reduces contention. Although this approach can be implemented very efficiently and with low overhead, on large systems especially NUMA architectures this central structures become bottlenecks again.

6.3.2 Local sleepqueues

To improve locality and parallelism sleepqueues could be distributed in a similar fashion as runqueues. Each processor would get its private sleepqueue and would be able to access it locally.

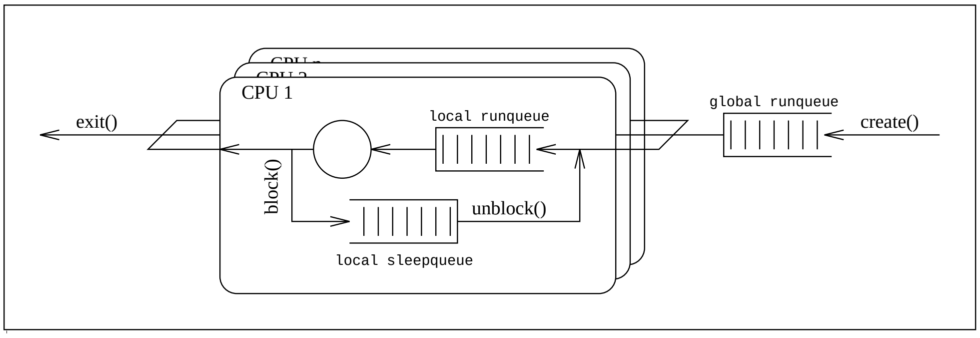


Figure 11: Using local sleepqueues for each processor

Although a structure like that shown in figure 11 would be possible it has a severe drawback. If threads on different processors or nodes would use the same synchronization mechanism, e.g. the same lock, the threads would be enqueued on different sleepqueues. If threads should be deblocked, a complex mechanism would have to be used to find the affected sleepqueues. On the one hand this would be very time consuming especially on large systems. On the other hand locality would be lost, as the deblocking processor would have to look at each sleepqueue, even on remote nodes.

6.3.3 Sleepqueues bound to synchronization mechanisms

Searching for threads that should be deblocked can be avoided completely if the sleepqueue for the mechanism is directly bound to it. Each mechanism has its own sleepqueue. This offers some interesting opportunities. Besides a high degree of possible parallelism, on machines like the Convex SPP with network caches, the mechanism with its sleepqueue seems to migrate to the node on which it is used. If only processors of this node access this mechanism it will have high locality. This is not possible with central runqueues, as all processors access this sleepqueue and therefore the structure would steadily migrate.

Figure 12 shows a system using this sleepqueues bound to synchronization mechanisms. Threads that get deblocked are usually enqueued onto the central runqueue or on the local runqueues of the processors where the thread has been running before it blocked.

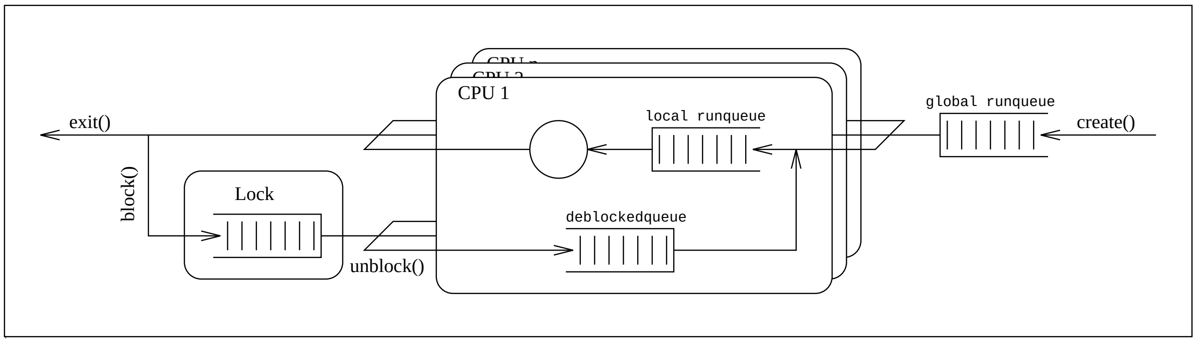


Figure 12: Sleepqueues bound to synchronization mechanisms

Particularly on system with priority queues as runqueues, each enqueue operation can be expensive. To reduce the overhead simple deblockedqueues can be used as shown in the figure for an intermediate store of deblocked threads. Each processor regularly scans its own deblockedqueues, calculates the deblocked threads' priorities and enqueues them on its local runqueue.

7 The *Mthreads* thread library

To evaluate our concepts we developed and implemented a userlevel thread library as part of the ELiTE[4] projects. This section will show the structure and results for some sample applications showing the possible improvements by the different mechanisms mentioned in the previous sections.

7.1 The internal structure

The *mthreads* library was specially designed for large scale NUMA architectures like the Convex SPP, which was used to evaluate our design.

The actual context switch mechanism is based on the *Quickthreads* package of David Keppel ([Kep93]). This package has been ported for the Convex SPP by Uwe Reeder ([Red95]).

Figure 13 shows the structure of our thread library that has been built on top of the quickthreads library. It's is obvious that the structure is similar to the hardware structure of the Convex SPP. For each level in the hardware hierarchy we have corresponding software structures that are connected on runtime. This enables us to dynamically adopt our thread library to different hardware structures, e.g. to a change in the number of processor or nodes.

[4] **E**rlangen **Li**ghtweight **T**hread **E**nvironment

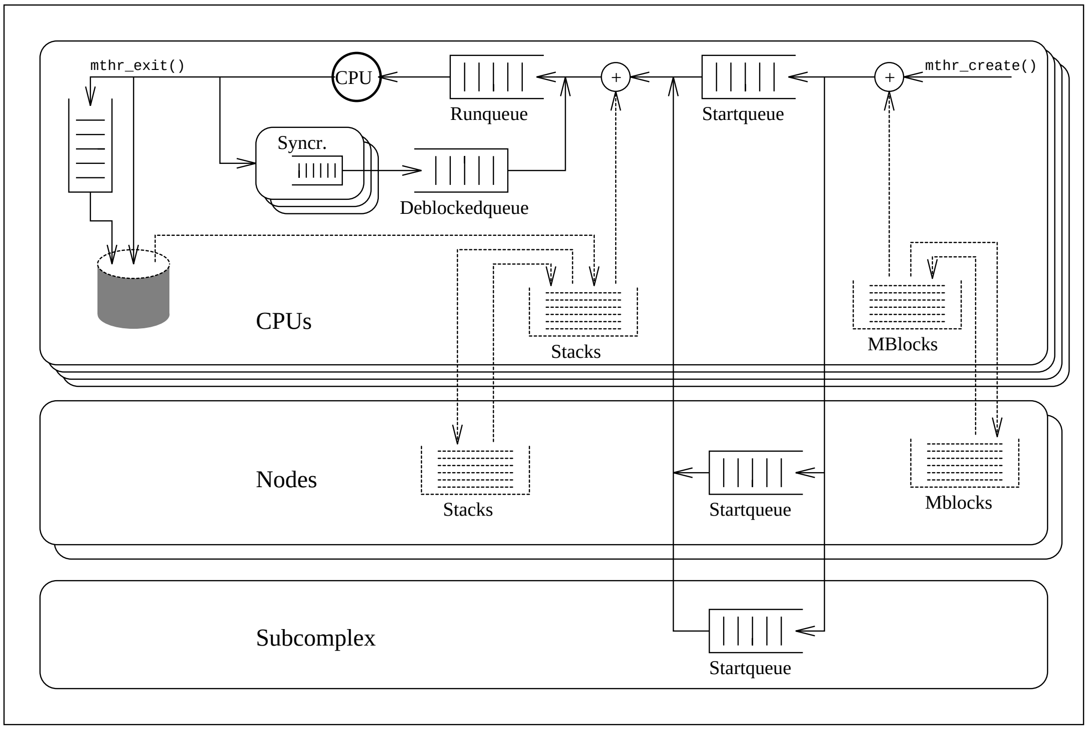


Figure 13: Structure of the *MThreads* thread library

7.1.1 Structures for thread management

Virtual processors: When the library is started it creates one kernel thread for each processor. Each of this kernelthreads is represented by a `vcpu_t` structure that holds references to local run-, start- and deblockedqueues, local pools etc.

Node structures: Similar to the processor structures these structures have references to a node local startqueue as well as to shared pools for stacks or memory blocks that can be accessed by all processors of this node. Moreover they know all processors of the node and their states.

The subcomplex structure The global subcomplex structure has information about all nodes and all processors and manages the global startqueue for threads that haven't been assign to special processors or nodes.

The runqueues: Each processor has its own runqueue. Because of the properties necessary for affinity scheduling like efficient support of priorities and operations with low overhead, we decided to use an ordinary binary tree. As it was known that dequeue operations work only at the front and back of the runqueue we could implement a simple balancing strategy and efficient dequeue operations.

The startqueues: Besides the usual thread states like blocked or runable, newly created *mthreads* are `STARTABLE` and get enqueued on separate startqueues. When threads are created, the programmer can assign them to one of the startqueues in the system, either to a specific processor's startqueue, to a special node's startqueue, or to the global startqueue. A processor looking for a new thread will traverse this hierarchy starting at its local startqueue. This results in a hierarchy of bindings of threads to locations.

Freelists: For all important structures that are frequently allocated and deleted, pools exist. These ensure optimal reusage of memory regions and furthermore simplify allocation as they guarantee correct alignment, This is necessary as HPPA-Risc processors in part need special alignment. For example memory cells used for locks have to be aligned on a 16 byte boundary.

As freelists are accessed in LIFO order, memory regions with the highest affinity to the different levels of caches are used first. If local pools fall empty or exceed a certain limit, elements are withdrawn from or deposited to the node's pools. Except the stackpool that transfers only one stack to and from the shared pool, all other freelists transfer a certain amount of elements at once to reduce contention at the central structure.

7.1.2 Synchronization

For synchronization the library offers mutex locks, semaphores, barriers and condition variables. All these mechanisms are based on a special form of condition variables with private sleepqueues. For blocking a two phase spinblocking algorithm is used. When a thread should be blocked the expected time the thread will be blocked can be presented to the `block` operation. Based on this information the algorithm decides whether to block at once or to spin before it blocks the thread. Mutexlocks for example evaluate the time the lock has been held and calculate the average time using a simple aging strategy.

To assure atomicity when enqueuing threads on the private sleepqueue spinlocks are used. These spinlocking operations, which are the most frequently called operations have been implemented in assembler. They have been realized as snooping locks, with exponential backoff and an upper limit for the delay between consecutive read operations.

7.1.3 Loadbalancing

If a processor has no runable thread on its own runqueue and doesn't find a startable thread on one of the startqueues in the hierarchy of startqueues, it switches to a private idlethread. These idlethread will perform the real load balancing. To avoid that – especially at the end of an application – lots of processors try to balance load, only one processor per node is allowed to do balancing simultaneously.

Load balancing works on two levels. In the first level it is tried to balance startable threads by moving startable threads from the higher levels in the hierarchy, i.e. from processor level down to the lower ones. If after this phase no startable thread is available for the processor, it knows that there is no startable thread in the whole system. In this case it proceeds with the second phase, i.e. it tries to migrate a runable thread. The search for a possible candidate is started at the local node. Therefore it looks at the threads with lowest priority on all runqueues of the local node and searches for the thread that leads to the highest improvement, by using information about the runtime of these threads, the load of all processors and the expected migration penalty. If no thread could be found the search continues on remote nodes. If no runable thread could be found at all, the virtual processor blocks.

7.1.4 Affinity evaluation

For being able to test different affinity models, affinity evaluation has been covered as an own module with a small interface that is used by the scheduling mechanism:

`aff_thread_stopped()` is called whenever a thread blocks or terminates to stop its affinity measurement.

`aff_thread_started()` is called subsequently to start the measurement for the successor thread.

`aff_greater()` is used to find out which thread has the higher affinity or priority. It is called on each comparison while a thread gets enqueued on some runqueue.

The Convex SPP 1000 used for our measurements allows either counting processor cachemisses or network cachemisses, but not both simultaneously. Moreover the measurement is very expensive compared to the processor cycle time as the mechanisms for counting are not part of the processor, but are implemented by additional hardware. Based on these possibilities we implemented and tested the following affinity models:

VTime: These model uses virtual time. Each time `aff_thread_stopped` gets called by the scheduler a processor private counter is incremented and stored as priority in the thread control block.

CMisses: This model assumes that a thread that produced a lot of cachemisses has loaded most of its working set into its processor's caches and for that reason should get high priority, i.e. the number of cachemisses of the last scheduling period is used as priority.

CmSum: Instead of using only the cachemisses of the last scheduling period, this model uses the sum of cachemisses of all the thread's scheduling periods as priority to include the thread's history into the priority calculation.

Reload: This model assigns higher priorities to threads that are expected to cause few cachemisses when restarted. Therefore the expected number of valid cachelines is calculated each time a thread starts or stops and is translated into the reload transient when the thread gets enqueued on the runqueue. Tables are used to simplify the calculation of footprint and reload transient values.

7.2 Results of different sample applications

To measure the influence of memory conscious scheduling and different affinity models on the runtime of application we parallelized numerical methods and additionally used several synthetic workloads that had a more predictable behaviour than numerical applications.

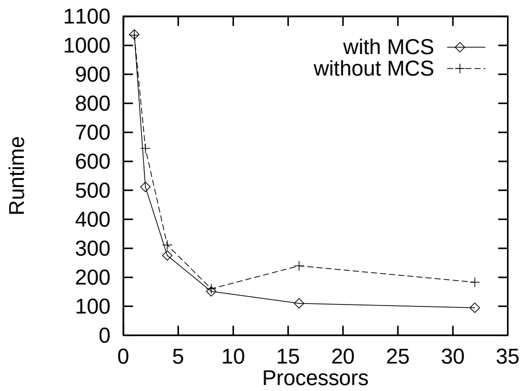
7.2.1 Sample applications

Synthetic workload: In this application a huge number of threads is created. All threads synchronize in couples resulting in a lot of context switches. Between two synchronizations no calculations or memory accesses are done. Therefore throughput depends on the efficiency of the mutexlocks used for synchronization and whether data structures like thread control blocks and stacks are resident in the processor cache.

Iterative numerical algorithm: The second application is an iterative algorithm used to solve large linear equations like those created by partial differential equations. The application uses the Jacobi algorithm and ran a few hundred iterations on a 2048x2048 matrix with 128 threads. As each thread accesses the same memory regions in each iteration this test is a good candidate for improvements by memory conscious and affinity scheduling.

7.2.2 Influence of MCS on runtime

To figure out the influence of MCS on the speedup that can be attained for the different applications we implemented a central version of our thread library that uses shared start- and runqueues as well as central pools. This library was compared to that shown in figure 13 with distributed structures.



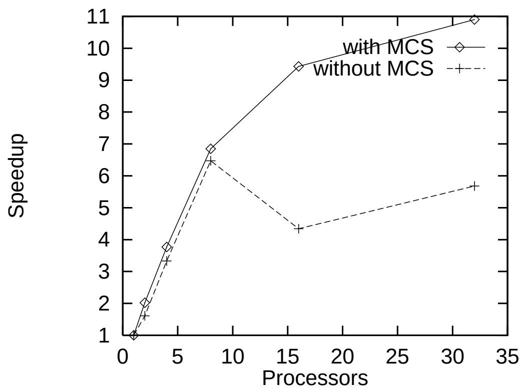


Figure 14: The iterative algorithm

Figure 14 shows the differences in runtime both of the central and of the distributed approach with MCS for the iterative application. Both versions scale very well for one node, but if the number of processors exceeds this border the version with MCS is still able to increase speedup whereas the central version's speedup drops to a value below that of 8 processors. On the one hand this decrease in speedup is due to the bottleneck of the central structures, but on the other hand it's caused by the growing communication costs. If the computation crosses hypernode borders, calculations of iterative processes are no longer efficiently possible without locality considerations.

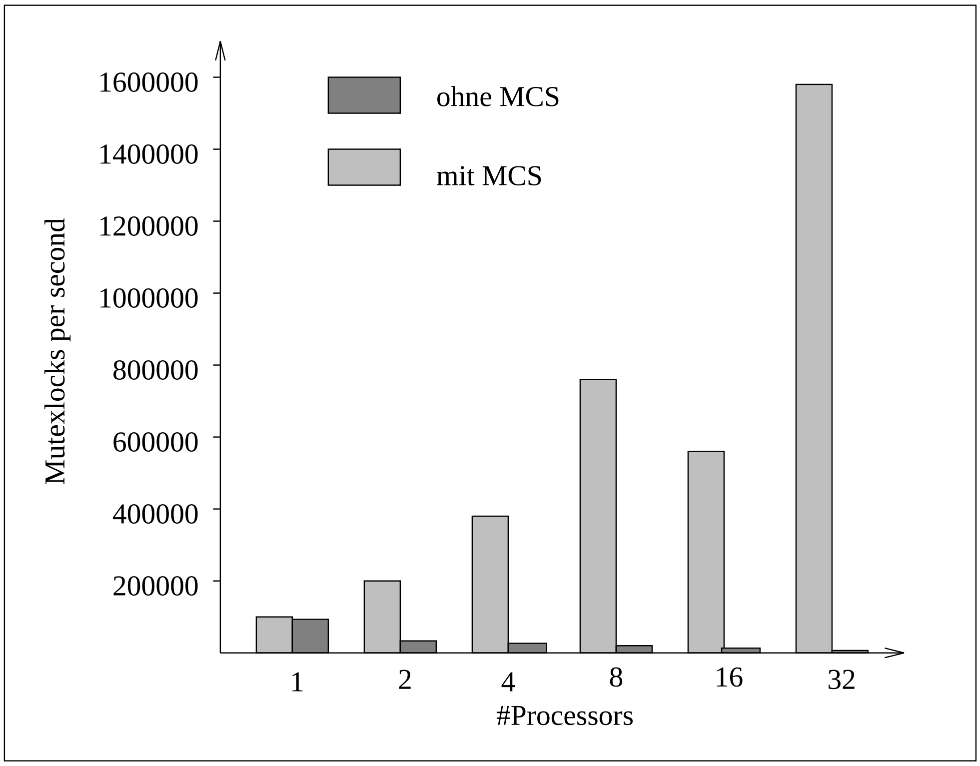


Figure 15: Synchronization speed

Figure 15 shows the number of mutex locks 4096 threads were able to acquire per second in our synthetic application. In the version without MCS this number decreases very fast for several reasons:

- The central scheduling structures become a bottleneck.
- As there is no possibility to collocate communicating threads, two threads accessing the same lock may be placed on different hypernodes, leading to high costs for each locking operation.
- Threads block and migrate frequently and thereby loose their complete cache state.

In constrast to the centralized version the version using MCS scales linearly with the number of processors up to 8 processors. Although the number of synchronizations decreases when the application uses more than 8 processors due to higher synchronization costs across different hypernodes, afterwards the overall performance continues to increase with the number of processors.

7.2.3 Influence of affinity scheduling on the runtime

In contrast to the synthetic workload, that has only neglectable working sets and therefore shows nearly no improvement by affinity considerations, applications like the iterative algorithm take benefit from thread reordering.

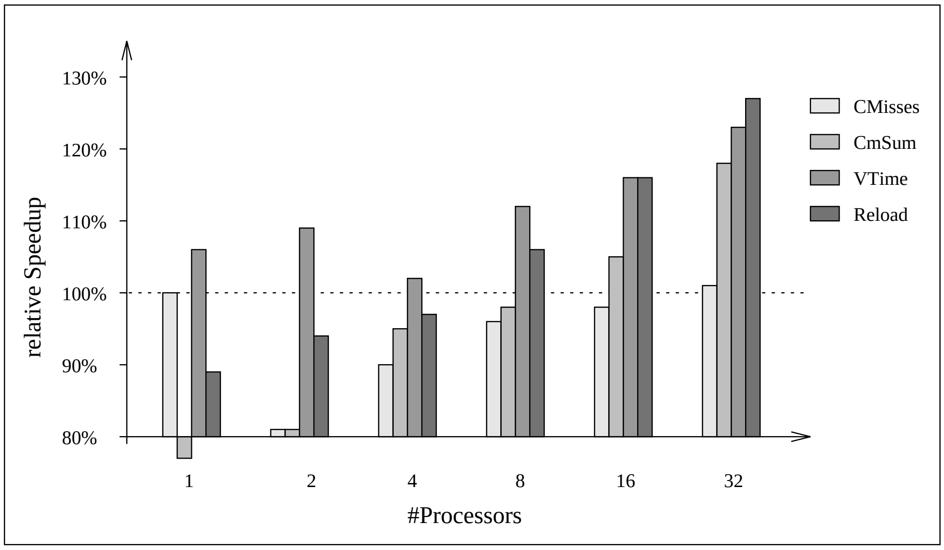


Figure 16: Influence of affinity models on the Jacobi algorithm

Figure 16 shows the improvements of different strategies in relation to **NoAff**, i.e. how much faster the application ran in contrast to using only MCS without additional affinity information:

CMisses: This model was not able to estimate cache affinity correctly as the results gathered by looking at the number of cachemisses the thread produced in its last scheduling period are no longer valid for the next scheduling period due to the huge number of cachemisses in the meantime.

CmSum: CmSum struggles with the same problem as CMisses. Nevertheless with more processors, i.e. more caches it can improve performance compared to NoAff.

VTime: VTime is able to estimate the cache state of threads fairly good. For all number of processors it leads to an improvement. Especially because of its low overhead and the fact that it needs no special hardware it's the first choice for small machines.

Moreover, using VTime we experienced an acceleration of about 25% for our fine-grained parallelization on one processor compared to the usual solution using one process, despite the overhead caused by thread switches etc.

Reload: This model leads to the highest speedup, but has to deal with the high costs for reading cachemiss counters on the Convex SPP and calculating the priority. Therefore these costs could only be outweighted if communication gets expensive, i.e. the computation crosses hypernode borders.

8 Summary and Conclusion

This paper showed that threads are not only an important means for manual parallelization, but can also help to improve the performance of parallel applications.

Two different approaches to improve userlevel thread scheduling have been figured out. Memory conscious scheduling uses locality information based on hints by the programmer or compiler to distribute threads on the available processors in a fashion that reduces communication costs and increases cache reusage. Moreover, the distributed structures used for MCS help to reduce the probability for bottlenecks. The second approach, called affinity scheduling refines the methods of MCS. Threads are scheduled in an order that leads to a better cache reusage by either applying simple intuitive methods or using a more complex mathematical model.

As a second topic we examined strategies and structures for efficient and scalable synchronization, even on large NUMA architectures. Once more, distributed structures had to be used to reduce contention and remote accesses that would otherwise cause a severe loss in performance.

The results of the implementation with MCS compared to that without MCS show the superiority of the distributed scheduling approach. Especially for frequently blocking applications running on a real NUMA architecture, i.e. on a machine with more than one node, distributed structures are crucial. Less contention is responsible for the investigated improvements as well as more locality by considering the affinity to caches. But the results of the synthetic work load showed that the ideas are also applicable to smaller machines, like those comparable to one node of a Convex SPP.

On the other hand, affinity scheduling offers additional improvements. Although simple models using cachemiss information couldn't satisfy, the more complex method using the reload transient showed the potential of these considerations. Especially on upcoming machines that are able to count cachemisses on the processor and can therefore efficiently evaluate these measures, these models will become more important. For machines that have no hardware support to count cachemisses or for small machines, virtual time can be used instead.

Our further research will concentrate on adopting our ideas to other, new architectures. Moreover we will refine our models by using the properties of these machines, e.g. to cover the influence of the different memory classes into our models.

References

[AAC+92] Gail Alverson, Robert Alverson, David Callahan, et al. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. *6th ACM International Converence on Supercomputing*, pages 188–197, July 1992.

[ALL89] T.E. Anderson, E.D. Lazowska, and H.M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12), December 1989.

[And90] T.E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[BB95] James M. Barton and Nawaf Bitar. A scalable multi-discipline, multiple-processor scheduling framework for irix. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949, pages 45–69. Springer, 1995.

[Bro77] Mark Robbin Brown. *The Analysis of a Practical and Nearly Optimal Priority Queue*. PhD thesis, Stanford University, February 1977.

[Hau95] Andrea Hauth. Vergleichende Untersuchung von Threadbibliotheken auf ihre Eignung für numerische und systemnahe Anwendungen. Studienarbeit, IMMD IV ,Friedrich-Alexander Universität Erlangen-Nürnberg, April 1995.

[Kep93] D. Keppel. Tools and techniques for building fast portable threads packages. Technical report, University of Washington, May 1993.

[KLMO91] A.R. Karlin, Kai Li, M.S. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 41–55, October 1991.

[Mar93] Evangelos Markatos. *Scheduling for Locality in Shared-Memory Multiprocessors*. PhD thesis, University of Rochester, Rochester, New York, 1993.

[MCS90] J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. Technical Report 342, University of Rochester, Department of Computer Science, 1990.

[ML91] E.P. Markatos and T.J. LeBlanc. Load balancing vs. locality management in shared-memory multiprocessors. Technical Report 399, University of Rochester, Computer Science Department, Rochester New York 14627, October 1991.

[ML93] Evangelos Markatos and Thomas J. LeBlanc. Locality-based scheduling for shared-memory multiprocessors. Technical report, Computer Science Department, Rochester , New York, 1993.

[Red95] U. Reder. Implementierung eines effizienten Prozeßumschalters auf Benutzerebene. Studienarbeit, IMMD IV, Friedrich Alexander Universität Erlangen-Nürnberg, 1995.

[SL89] M. Squillante and E.D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. Technical report, Department of Computer Science, University of Washington, 1989.

[SW95] Patrick G. Sobalvarro and William E. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 107–126. Springer, 1995.

[TS87] D. Thiebaut and H.S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5:305–329, 1987.

[TTG95] J Torrellas, A. Tucker, and A. Gupta. Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, February 1995.

[Tuc93] A. Tucker. *Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors*. Dissertation, Stanford University, 1993.

[VZ91] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 26–40, October 1991.