Memory Conscious Scheduling and Processor Allocation on NUMA Architrchitectures

Frank Bellosa

June 1995 TR-I4-6-95

Technical Report

Institut für
Mathematische Maschinen
und Datenverarbeitung
der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl für Informatik IV
(Betriebssysteme)

Memory Conscious Scheduling and Processor Allocation on NUMA Architectures

Frank Bellosa
University Erlangen-Nürnberg
Department of Computer Science IV
Martensstr. 1, 91058 Erlangen, Germany
email:bellosa@informatik.uni-erlangen.de
Phone: +49 9131 85 7275
Fax: +49 9131 85 8732

Abstract

Operating system abstractions do not always meet the needs of a language or applications designer. A lack of efficiency and functionality in scheduling mechanisms can be filled by an application-specific runtime environment providing mechanisms for dynamic processor allocation and memory conscious scheduling. We believe that a synergistic approach that involves three components, the operating system, a user-level runtime system and a dynamic processor server can offer the best adaptivity to the needs of multiprogramming.

Especially on NUMA architectures data structures and policies of a scheduling architecture have to reflect the various levels of the memory hierarchy in order to achieve high data locality. While CPU utilization still determines scheduling decisions of contemporary schedulers, we propose novel scheduling policies basing on cache miss rates. An interface between user-level runtime system and application is essential to initiate a concurrent memory prefetching.The application is informed about scheduling decisions of the runtime system and can trigger prefetch operations. For the implementation of the runtime system we follow a two level approach: The lower level consists of assembler code for fast thread initialization and context switching. The upper level includes the user-level scheduler which provides load balancing and high cache reusage on top of kernel threads.

Because static processor sets, MACH cpu_servers and gang scheduling do not offer the required flexibility and efficiency in processor allocation and scheduling, a new approach to these topics had to be developed. The design of an adaptive dynamic processor server will be sketched. The decisions of this processor server base on processor requests and on information about memory locality of currently running applications. An interface between processor server and user-level scheduler allows the exchange of information to establish a dynamic partitioning of the processors among multiple parallel applications to achieve an optimum between throughput and fairness.

1 Introduction

Cache-coherent multiprocessors with **n**on **u**niform **m**emory **a**ccess (**NUMA** architectures) have become quite attractive as compute servers for parallel applications in the field of scientific computing. They combine scalability and the shared memory programming model, discharging the applications designer from data distribution and coherency maintenance. But still load balancing and scheduling are of crucial importance.

The parallelism expressed using “UNIX-like” heavy-weight processes and shared memory segments is coarse-grained and too inefficient for general purpose parallel programming, because all operations on processes like creation, deleting and context switch invoke complex kernel activities as well as costs associated with cache and TLB misses due to address space changes.
Contemporary operating systems (like SUN’s Solaris or MACH) offer middle-weight kernel-level threads decoupling address space and execution entities. Multiple kernel threads mapped to multiple processors can speed up a parallel application. But the potential benefit is limited by the kernel’s scheduling features, which cannot take into consideration the special needs of multiprogramming. Furthermore, kernel threads cannot offer a fine-grained programming model because thread management implies expensive protected system calls.
By moving thread management and synchronization into user-level the cost of thread management operations can be drastically reduced to one order of magnitude more than a procedure call [1]. Some advantages of user-level threads are:

- All scheduling operations belonging to a single application are handled inside the same address space. Cache and TLB misses can be reduced to a minimum.
- The scheduling algorithm and its interface can be designed with respect to the needs of a specific application, thus offering the optimum in performance and functionality. For example, preemptive- or priority-based scheduling of threads can be omitted, if this is not necessary for a specific application, to achieve low thread management overhead.
- Control structures for processes and threads are statically allocated in most kernels. Only the user-level offers the flexibility in adapting control and queue structures to the degree of parallelism inherent to an application ranging from several up to thousands of threads.

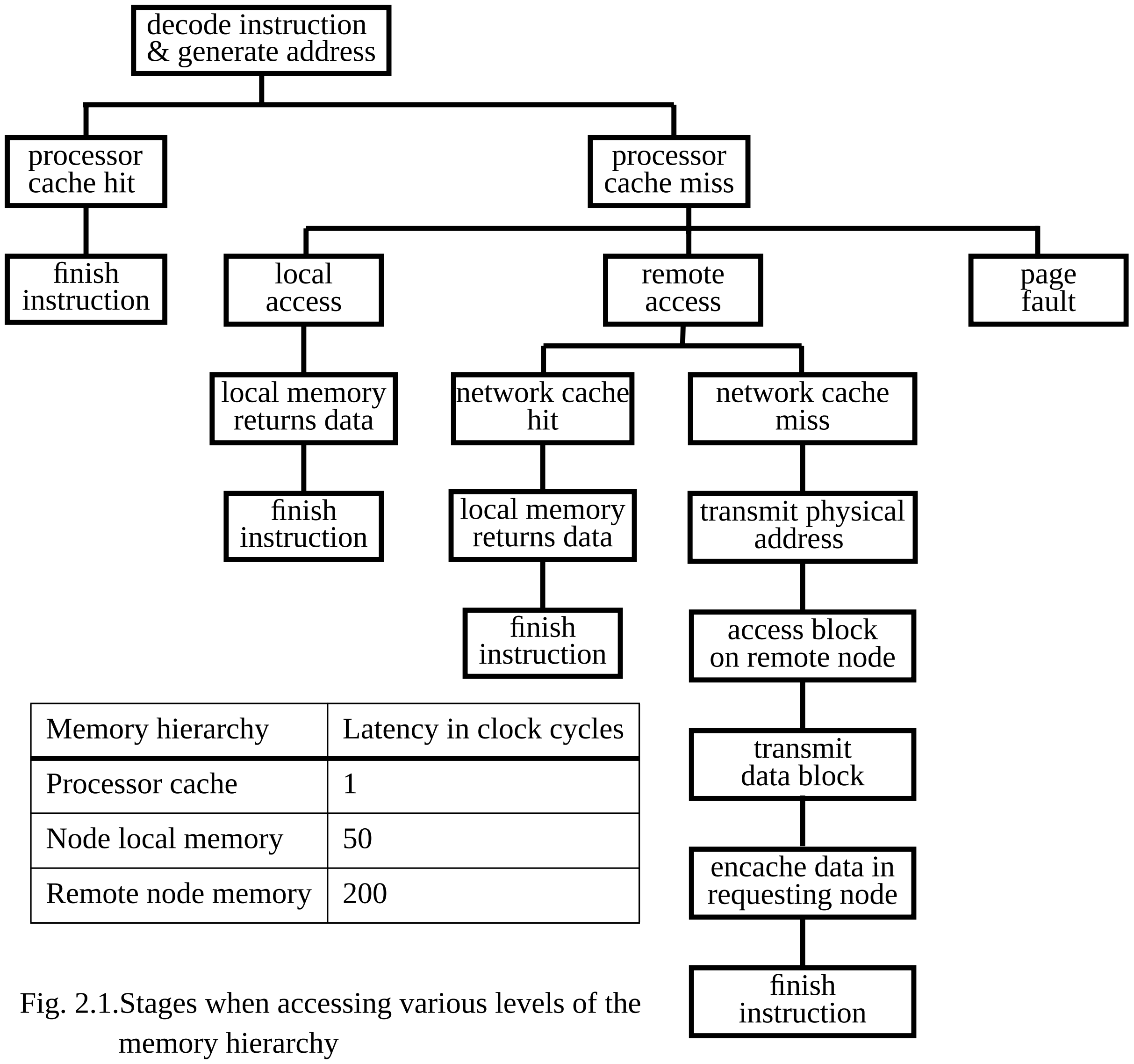
In general, light-weight user-level threads, managed by a runtime library, are executed by kernel threads, which again are mapped on the available physical processors by the kernel. Problems with this two-level scheduling arise from interference of scheduling policies on different levels of control without any coordination or communication.
On NUMA architectures with their discrepancy between computing and communication performance, memory conscious scheduling is essential to minimize the total completion time of an application by reducing inter-processor communication. One common representative of memory conscious scheduling is cache affinity scheduling proposed in [20]. The decisions of this type of scheduling base on the CPU utilization and the information about the processor where the most recent execution of a specific thread took place. Our approach to memory conscious scheduling goes beyond the usage of information about timing and execution location by using cache miss rates for each levels of the memory hierarchy.

In this paper we propose a non-preemptive user-level threads package with an application interface to invoke - based on scheduling decisions of the runtime system - prefetch operations to hide memory latencies. For keeping multiple applications near to their operating point along the speedup curve, an adaptive processor server will be sketched, which performs space sharing of a NUMA multiprocessor based on information about processor requests and cache misses.

The rest of the paper is organized as follows. We describe in section 2 the architecture of the CONVEX SPP, a cache coherent NUMA multiprocessor, which is the architecture for our prototype implementation. The design of a memory conscious runtime system with various strategies will be sketched in section 3 as well as an adaptive processor server . Finally we conclude in section 4.

2 Architecture of the CONVEX SPP

The Convex Exemplar Architecture [6] implemented in the Convex SPP Multiprocessor is a representative of cache coherent NUMA architectures. A symmetric multiprocessor called hypernode is the building block in the SPP architecture. Multiple hypernodes share a low-latency interconnect responsible for memory-address based cache coherency. Each hypernode consists of two to eight HPPA 7100 processors, each having 1 MB direct mapped instruction and data cache with a cache line size of 32 bytes. The processors of a single hypernode can access up to two GBytes of main memory over a non-blocking crossbar switch . The memory of remote hypernodes can be accessed over the interconnect. To reduce network traffic, a part of the memory is used as a network cache with a cache line size of 64 bytes. Load/store operations require various stages depending on the locality of the referenced memory region (see Figure 2.1).



Memory hierarchy	Latency in clock cycles
Processor cache	1
Node local memory	50
Remote node memory	200

Fig. 2.1.Stages when accessing various levels of the memory hierarchy

There are non blocking prefetch operations to concurrently fetch data regions from a remote node into the local network cache. These operations can be used to overlap computation and network traffic in order to hide latency.

Performance relevant events can be recorded by a performance monitor attached to each CPU. The performance and event monitor registers cache misses satisfied by the local or a remote hypernode as well as the time, the processor waits for a cache miss to be served. For high resolution time stamps several timers with various resolutions are available. There is also a system-wide clock with a precicion of 1µs.

The operating system is a MACH 3.0 microkernel with a HP/UX compatible Unix server on top. The system call interface from Hewlett-Packard's Unix and additionally, a system interface to create and control kernel threads are provided.

3 **E**rlangen **Li**ghtweight **T**hread **E**nvironment **(ELiTE)**

Most thread schedulers attempt to optimize load balance while reducing the costs for thread management including queue locking. This strategy is reasonable for bus-based shared memory architectures with an uniform memory access. The most valuable resource of these architectures is the computing power of the processor and the bandwidth of the bus system. Thus, scheduling policies focus on a high processor utilization while reducing bus contention.
The focus of thread scheduling has to move when we look at scalable shared memory architectures with non uniform memory access. Modern RISC-based processors are able to perform 1-2 operations per clock cycle while simultaneously performing a load/store operation to the processor cache. One can only take advantage of this immense computing power, if the processors can be supplied with data in time. Consequently, low memory latency is the key to high processor utilization. The bandwidth of interconnection networks is not the bottleneck for today's scalable parallel processors (e.g. the slotted ring interconnect of the Convex SPP has a bandwidth of 2.8 GBytes/s) whereas the time critical latency to access different levels in the memory hierarchy cannot be reduced in the next time because of technological reasons. Both switches and affordable dynamic memory cause a latency of about hundred nanoseconds, while processor cycles just need a few nanoseconds. The consequence of this discrepancy is that scheduling policies for NUMA architectures have to satisfy three essential design goals:

(1) **Memory Conscious Scheduling:** Threads are assigned to the processor, which is close to the data accessed by the thread. This policy tries to reduce processor waiting time due to cache misses. Fairness among threads of the same application is not necessary, as each optimal used processor cycle within an application helps to increase throughput.

(2) **Latency Hiding:** Prefetch Operations cause an overlapping of computation and communication.

(3) **Dynamic Processor Allocation**: Due to decreasing spatial locality, communication overhead and redundant work, the efficiency of a parallel application normally decreases as the number of processors is increased. In a multiprogrammed environment the overall throughput can be better compared to a single application environment, if all running applications run with a higher efficiency. This effect is called operating point effect [20]. The operation point effect can be achieved by a dynamic processor server partitioning the processors among multiple parallel applications.

As contemporary threads packages, developed for the use on shared memory multiprocessors with a modest number of processors, have design goals, which are not applicable for scalable NUMA multiprocessors with a high number of processors, novel scheduling architectures have to be designed.

3.1 Architecture

We present a scheduling architecture, which is intended for the use in fine-grained numerical applications or for the use by a compiler doing automatic parallelization.

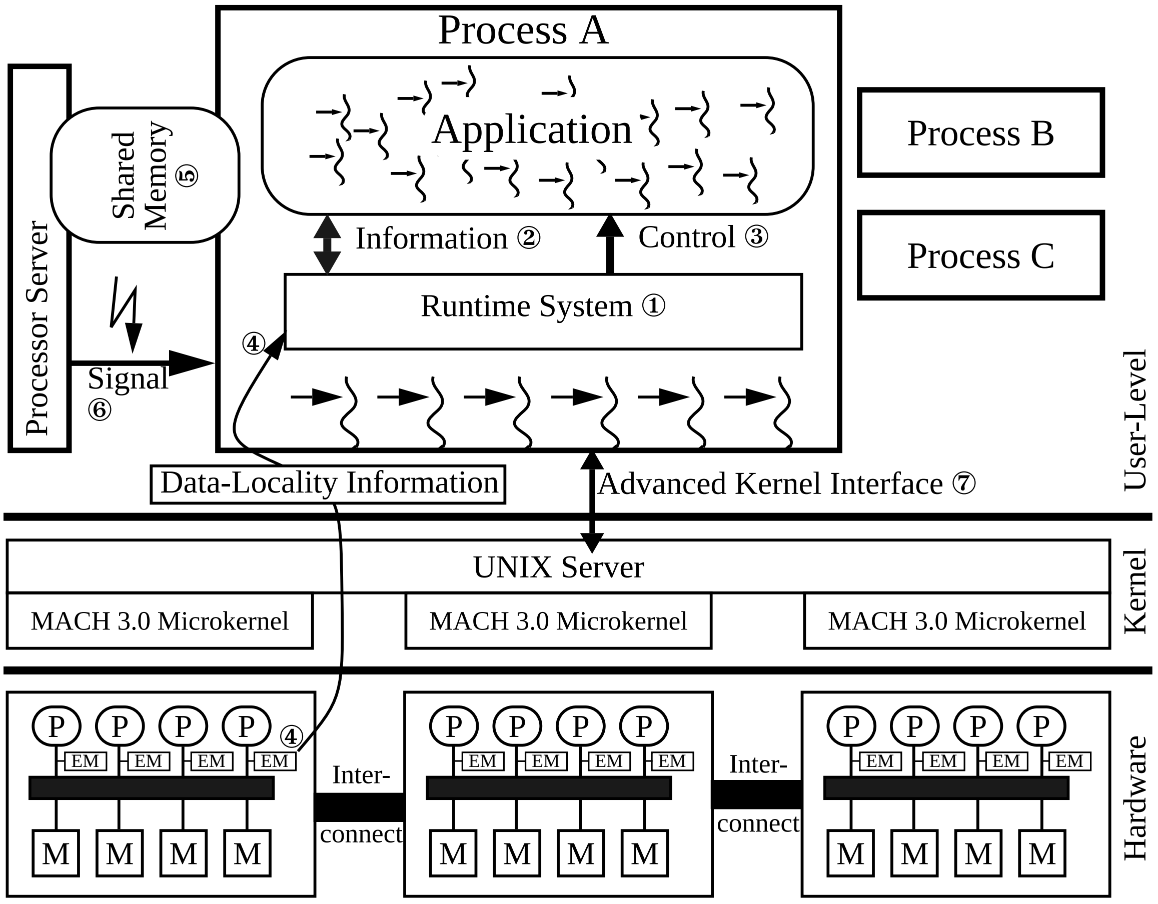


Fig. 3.1.Scheduling architecture of the Erlangen Lightweight Thread Environment (ELiTE)

The following architectural elements characterize the ELiTE architecture:

– Information about data locality of specific threads, provided by the event monitor unit (EM) of the hardware (see figure 3.1. ④) are gathered by the runtime system ① to influence scheduling on the base of priority queues. This is the key to memory conscious scheduling as presented in detail in the next subsection (refer to section 3.2).

– An interface between application and runtime system (see figure 3.1. ②) is essential to invoke - based on scheduling decisions of the runtime system - prefetch operations to hide memory latencies (refer to section 3.3).

- The mechanism for the context switch (see figure 3.1. ③) has to be optimized, to keep the costs for this operation to a minimum. Switches occur in the course of suspend operations and after the polling phase when using two-phase locking (refer to section 3.4).
- To achieve fairness between multiple applications and to keep multiple applications close to their efficient operating point an adaptive processor server performs space sharing of a NUMA multiprocessor, based on information about processor requests and cache misses (see figure 3.1. ⑤⑥) on different levels of the memory hierarchy (refer to section 3.5).
- An advanced kernel interface (see figure 3.1. ⑦) eases information exchange between kernel and runtime system to keep the number of running kernel threads used as virtual processors fixed and to prevent a loss of parallelism due to blocking system calls (refer to section 3.6).

3.2 Memory conscious user-level scheduling

Algorithmic optimizations of the application and scheduling mechanisms for the management of parallelism determine the overall throughput. The applications designer cannot be relieved from algorithmic considerations concerning memory locality. But he can take advantage of a scheduling, which makes a fine-grained architecture-independent programming style possible by its efficient memory conscious thread management.

In [1] the performance impact of different thread management alternatives for shared memory bus-based multiprocessors has been investigated. Advantages of per-processor ready queues and per-processor free lists could be demonstrated:

- Enqueueing and dequeueing threads can occur in parallel, with each processor using a different queue. To prevent idling of a processor with an empty queue, each ready queue has a separate lock, such that an idle processor can scan other queues for work, beginning with its own.
- To minimize memory allocations from central pools, control blocks and stacks should be stored in local free lists.

Both results should also apply to NUMA architectures. While the aim of locality in bus based multiprocessors is to prevent bus contention when accessing central locks, the focus has to be moved to reduction of the latency when referencing memory in NUMA machines.

To reduce the total completion time of an application and consequently the overall throughput, the number of cache misses has to be minimized. To address this problem, research has been done in the field of kernel level scheduling by modifying the Unix scheduler to perform a processor affinity scheduling of heavyweight processes [20] . Furthermore load balancing techniques for user-level threads have been evaluated where just those threads were allowed to migrate to an idle processor, when probably having few data in the cache [15].

One aim of this paper is to initiate a discussion about novel approaches in scheduling policies and data structures, reflecting the various levels in the memory hierarchy of NUMA architectures.

Basis of every scheduling is a measure of the resource of interest. The resource utilization of the CPU is measured by the consumed CPU cycles of a thread. Corresponding a measure of locality is the number of cache hits of a thread in its history. Most of the existing NUMA machines, as

Convex SPP or KSR1/2, provide data about cache misses. So we use dislocality as the basis of our scheduling considerations.
Dislocality is the average number of cache misses per time unit during the last runs of a thread.

In the next sections, we discuss the pros and cons of three scheduling policies (Total Locality Scheduling, Static Locality Scheduling and Dynamic Time-Locality Scheduling) which differ in complexity and completeness.

3.2.1 Total Locality Scheduling

The inverse priority of a thread is the product of its dislocality and the time between its last run and the moment of the enqueueing. The thread with the lowest inverse priority will be chosen for execution. Threads are stored in a *priority tree.* The distance from a node to the leafs corresponds to the disslocality.

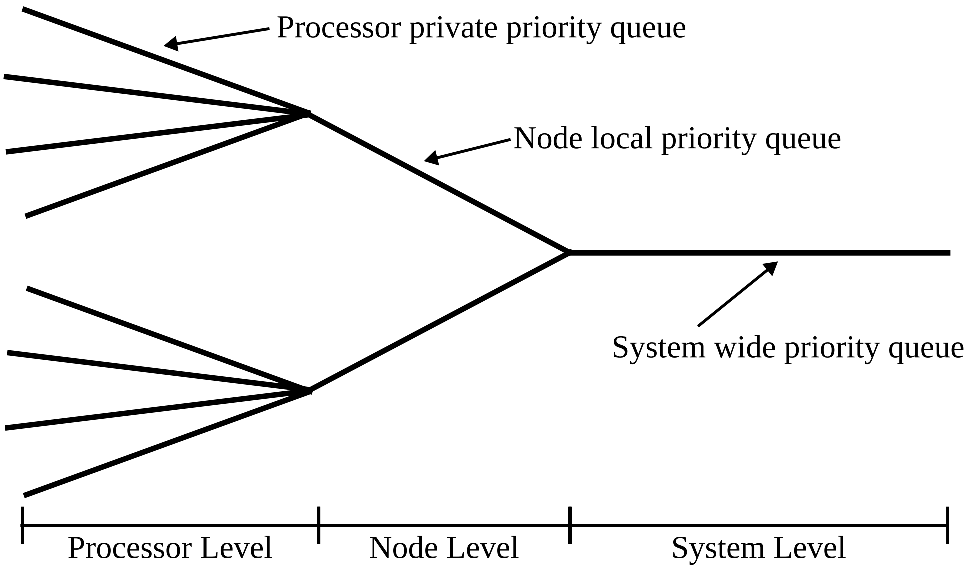

Processor private priority queue
Node local priority queue
System wide priority queue
Processor Level
Node Level
System Level

Fig. 3.2.Hierarchical priority tree to store thread control blocks according to their locality

The scheduling heuristic is based on the following assumptions:

- A thread which had a high data locality in its history, will run on with high data locality after assigment to its previous processor.
- A thread with a high dislocality will have a lot of cache misses, even when it would be reassigned to its previous processor.

Provided a non preemptive scheduling policy, a thread has to be enqueued during creation and after unblocking. As thread specific data will be initialized during creation, the new created thread should be enqueued with the highest priority (corresponding to the value 0 in the inverse priority range) in the local priority queue of the creator. For a resumed thread, the new inverse priority can be calculated as the product of dislocality and the time since the latest run. A thread with high locality not being executed in recent processor cycles can have the same priority as a thread with a poor locality, being executed most recently. Because the priority of threads with different locality changes over time, a cyclic recalculation of the priority of all runnable threads is necessary. Thresholds define, when a thread has to be enqueued in the processor local queue, the node local queue, and the system wide queue. The thresholds depend on the memory size of the levels in the hierarchy and the latency to access a specific level.

The Total Locality Strategy has some interesting advantages:

- Idle processors can look for new runnable threads in the local processor queue, then in the local node queue and finally in the system pool. Threashing of threads between processors will be avoided, because the time stamp and locality based queueing strategy with the re-evaluation of runnable threads according to their locality achieves a throttled movement of deblocked runnable threads from one queue level to the next as the waiting time increases.
- Local queues do not have to be locked, because only the local processor has access. Access to the node local or to the system queue can be reduced to a minimum by fetching multiple runnable threads from these priority queues into the local queue.

To resume a thread, the state in the thread control block should be set to runnable, but the enqueueing of this thread should be done by the processor recently executing this thread, because enqueueing implies accessing a lot of thread specific data. An enqueueing stack local to the processor within reach of all other processors has the advantage, that the stack has just to be locked for the time of pushing and popping the address of the unblocked thread.
Several data structures for priority queues exist [11], where Fibonacci heaps and relaxed heaps [7] just need O(log #threads) operations for the time critical 'find_and_delete_minimum'-operation, necessary to identify and extract the processes with minimal dislocality from the priority queue.

3.2.2 Static Locality Scheduling

Giving up the aging of enqueued threads and the total order within a local queue allows the use of simple hash queues. Like in the Total Locality Scheduling we have a hierarchy of hash queues according to the memory hierarchy. When unblocked threads will be enqueued, the thread control blocks (TCBs) get enqueued according to their dislocality in a hash queue. Again we have to find thresholds and hash functions which match the size and the latency of the caches.

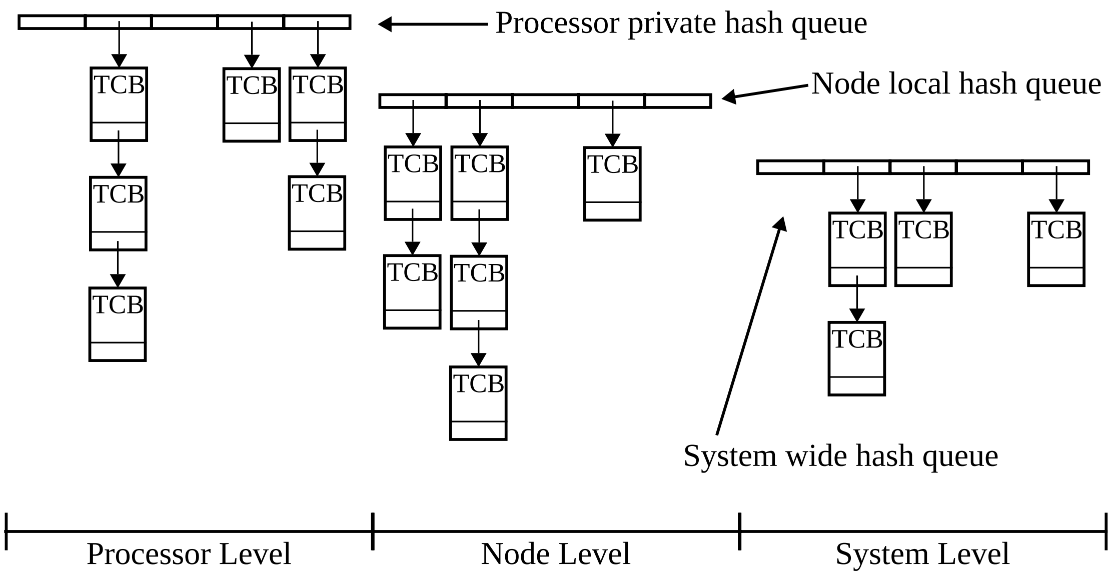


Fig. 3.3.Hierarchical hash queues without aging

A simple hash function can be the linear mapping of the range between thresholds of two memory levels onto the hash buckets. A threshold can be a percentage of available cache-lines. If the dislocality is greater than the threshold, we assume, that there is no more valid data in the corresponding cache. We currently evaluate thresholds corresponding to 20%-30% of the number of cache lines.

Apart from serious trade-offs concerning aging and total queue order the Static Locality Scheduling has some advantages:

- Hashing needs just a fixed number of operations for enqueueing.
- The dequeueing needs a fixed number of operations, if we treat the linked lists in the hash table as a FIFO or LIFO queue.

3.2.3 Dynamic Time-Locality Scheduling

For applications with a huge number of threads with the same access patterns, the number of cache misses of an individual thread can be neglected and only the time since the last run of a thread is of interest. Having only this variable, the advantage of hashing can be combined with aging.

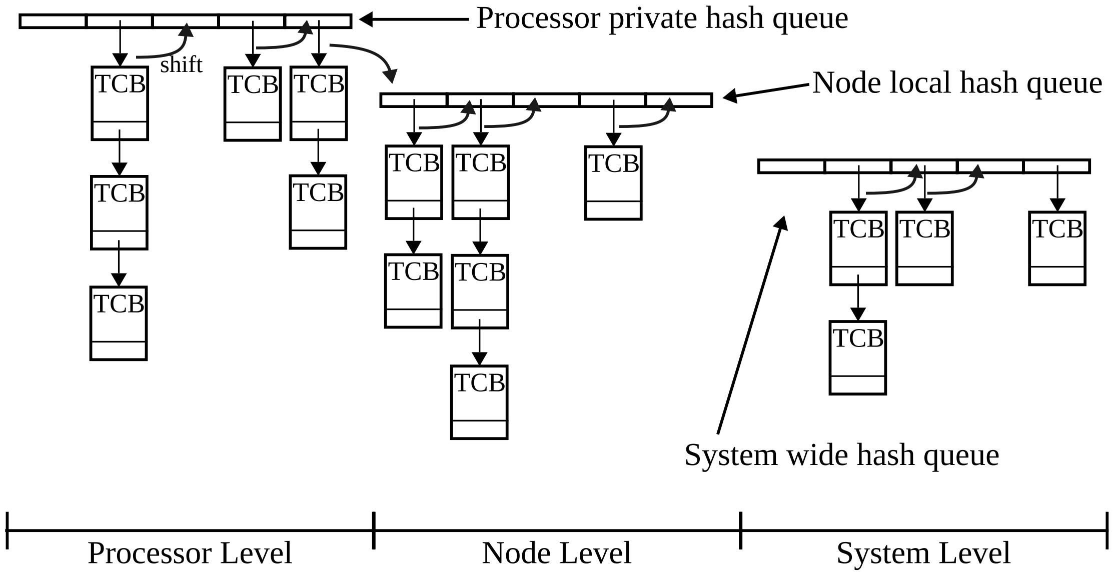


Fig. 3.4.Hierarchical hash queues with aging by shifting

The threads are filled into hash buckets according to the time of their last run. As this time proceeds, the hash fields have to be shifted in the way, that the time stamps of the enqueued threads match the hash field representing the corresponding time frame. If we use a hash field with 8 entries each with a pointer to a queue of TCBs, a shift operation can be done in 16 processor cycles, because all entries fit into a single cache line.

Provided that the locality of all threads is nearly the same, Dynamic Time-Locality Scheduling guarantees efficiency of the mechanism as well as time adaptivity by the mechanism of aging by shifting.

3.3 Scheduler Interface

Contemporary NUMA architectures like Convex SPP or KSR1/2 have non-blocking prefetch operations in their instruction set to concurrently fetch data regions from a remote node into the local network cache overlapping computation and network traffic and thus hiding latency. If thread specific data can be stored in a single block, a pointer to this block and its length can be stored in the thread control block. If there is an interface to the scheduler, a currently running thread can ask the runtime system to prefetch the data of the thread, which will run in the near future. This idea was motivated by implementations of adaptive numerical methods [19][3], where thousands of threads, each corresponding to a point of an adaptive grid, resume the threads representing the grid points in the neighborhood after calculating the local grid point before they suspend themselves. This numerical method, called *active threads strategy* can only run with high efficiency on NUMA architectures, if all thread specific data is resident in the data cache, before the context switch occurs.

3.4 Fast context switch and synchronization

A fast context switch free of race conditions is the basis of most synchronization mechanisms inside a runtime system. Lim and Agarwal [14] have investigated waiting algorithms for synchronization in large-scale multiprocessors. With increasing CPU numbers, the type of synchronization has a significant influence on the performance of fine-grained parallel applications. The proposed two-phased waiting algorithm combines the advantage of polling and signalling. After a fixed polling interval a thread blocks. The polling threshold depends on the overhead of blocking. While this algorithms shows good results for mutual exclusion, blocking always performs better for barrier synchronizations, because wait times at barriers are likely to be long. The results of [14] are of relevance for us, because the timing behavior of the investigated MIT Alewife multiprocessors has great resemblance to RISC-based scalable architectures like Convex SPP or KSR1/2. The proposed context reduction close to synchronization calls can be realized by compiler pragmas to reduce the costs of a context switch when blocking.

Context switching is delicate for race conditions on multiprocessor systems, because one processor could resume an enqueued thread while its stack is not yet completely frozen by the processor of its last run. To implement a context switching two switching models have been discussed:

- **Scheduler Threads**: During a switch, control is returned to a scheduler thread local to each processor. The scheduler thread enqueues a thread from the run queue and performs an additional switch to it. Races cannot occur because the freezing of a thread is performed on the stack of the scheduler. This simple and secure switch model however is very time expensive, as two context switches per thread switch are necessary.
- **Preswitch**: After saving the state of the old thread, the stack will be a switched to the stack of the new thread. Using this stack, the TCB of the old thread can be enqueued without the danger of a race condition. This mechanism assumes, that the next thread is known and existent, before the switch occurs and that the next thread already owns a stack, which makes lazy stack allocation difficult.

As switching efficiency is essential for a fast runtime system, preswitching will be used in ELiTE. Based on the QuickThreads package of the University of Washington [10] providing the preswitch model for various processor architectures, we have ported the package to the HP PA-RISC processors architecture [18]. On a CONVEX SPP using this type of processor, the following times for a context switch can be reported:

Operation	Clock Cycles
Context Switch within Cache	153
Context Switch within Node	1122
Context Switch between Nodes	1805

The proportion for a context switch with thread control blocks in the three levels of the memory hierarchy is 153/1122/1805 = 1/7/12. These are exactly the proportions, we expected calculating with a memory latency of 1/50/200 cycles and 32/(64) Bytes (network-) cache lines. Most of the time is spent for saving and restoring the callee-saves registers. The consequence is that switching can only be optimized by reducing the number of registers, which have to be saved. These are the callee-saves registers, regulated by the calling conventions (e.g. by the HP PA-RISC calling conventions). As context saving and restoring for most contemporary RISC processors (an exception is the SUN SPARC processor with its register windows) is a sequence of machine instructions and no part of the instruction set, a change in the calling conventions could make context switching much more efficient by increasing the caller-saves registers and reducing the callee-saves registers.

3.5 Dynamic Processor Server

Space sharing of a multiprocessor is beside gang scheduling [17] one mechanism to prevent a decrease in performance due to cache data corruption because of context switches and preemption of kernel threads inside critical sections. The parallelism as well as the data locality inherent to an application determins the optimal operating point, where an application can run with its best efficiency. Running multiple applications on their operating point improves the overall throughput compared to sequential runs of applications, which cannot use the whole machine efficiently. This operating point changes over time due to swaying requests of applications and because of multiprogramming. A processor server proposed in [4] suffers from its static allocation policy.
The dynamic processor server proposed in [5] divides the processors among multithreaded applications according to processor requests and processor usage, thus focusing on high processor usage. But CPU usage is not the resource of interest concerning NUMA architectures. Our approach partitions the processors based on requests and cache misses.

A dynamic processor server can be implemented as a normal process exchanging data with the runtime system of multiple applications via shared memory (e.g. SYS V shared memory segments or memory mapped files) and signalling to divide up the available processors.

– Each application is entitled to at least one kernel thread.

– If an application requests an additional processor, the processor is granted, if there are more processors than kernel threads. Otherwise the dislocality (average number of cache misses per processor in a given time unit) of all running applications is evaluated. If the requesting application is not the one with the poorest locality, a processor is taken away from the application with the poorest locality having more than two kernel threads. The spatial locality of both application has to be taken into consideration to reduce the number of used hypernodes for each application to a minimum. Taking away a processor means, suspending a kernel thread, which cannot serve as virtual processor after suspension. Assigning a processor to an application means, resuming a kernel thread bound to a processor.

Information about locality and processor requests have to be sent to the processor server in the one direction. On the other way, the demand to suspend or to resume a kernel thread has to be sent back to the application. Research has been done in efficient communication mechanisms to address this problem [20]. It is the question of signalling or polling (see figure 3.1. ⑤⑥).

The high overhead associated with signal handling can only be justified, if there is an idle processor waiting for new work. This is the case, if the server signals the runtime system to resume a kernel thread.
Suspension is only allowed when a kernel thread reaches a safe suspension point. A suspension point is safe, if all kernel thread specific data like local priority queues and wait channels has been moved to node local or system global data structures. As a consequence, suspension should only occur, if the application voluntarily makes a blocking call to the runtime system e.g. because of a locking operation. Inside the runtime system, the kernel thread can then poll for information and clean up all its data if its suspension is demanded. There is no loss in overall throughput, because as long as a kernel thread dedicated for suspension runs useful work will be done and no processor is idle.
Data concerning locality can be placed in the shared memory region of the runtime system and the server without any costs. If a processor is requested, this data can be evaluated to decide which kernel thread has to suspend or is allowed to resume.

3.6 Kernel Interface

User-level runtime systems use kernel threads as virtual processors, assuming an equivalence of physical and virtual processor, which cannot be maintained because of events like I/O, page faults, and blocking system calls. To keep up this equivalence, the user-level has to be notified to react adequate to these events.
Scheduler Activations, proposed in [2], use kernel threads to upcall the runtime system. This strategy suffers from the fact, that you need a free processor to run the kernel thread upcalling the user-level. But in the case of a request for suspension of a virtual-processor, there is no free processor. The consequence is an expensive context switch on kernel level causing TLB misses and data cache corruption.
In [13] and [20] communication mechanisms between the kernel and a user-level thread library are proposed to retort the performance losses when threads block in the kernel or are preempted in critical sections. The kernel and the threads package communicate using shared memory whenever possible to avoid the need for synchronization interaction. Softwareinterrupts signal the thread package whenever a scheduling decision may be required. For example, polling of shared memory in a safe suspension point is used to inform the runtime system to suspend a thread, while signalling will be used to inform the runtime system, that a thread can be resumed or a new kernel thread can be created. Signalling is used to prevent idling a processor while information exchange over shared memory is used whenever quick response to events is not so important.

A strategy offering fast response to blocking events is proposed in [12] and will be used in ELiTE. The runtime system parks spare kernel threads in the kernel. In the case of a blocking call, the kernel deblocks a parked thread to maintain a fixed number of running kernel threads. When the blocking request is resolved, the kernel informs the runtime system about the deblocking via a shared page or shared memory segment. If this deblocked user-level thread is selected for execution, the corresponding kernel thread initiate a system call to park in kernel again and to release the blocked kernel thread. The system is in the same state as before the blocking call.

4 Summary and Conclusions

The overhead associated with fine-grained threads goes beyond the cost of thread management, because of memory transfers between the various levels of the memory hierarchy. As maximum throughput is the goal of our efforts, we have presented the architecture of the Erlangen Lightweight Thread Environment (ELiTE). The focus of this scheduling architecture lies on the reduction of cache misses and on the hiding of latencies while preserving fairness among multiple applications running on a cache coherent NUMA multiprocessor.
Short time memory conscious scheduling has to be done by a runtime system. As contemporary threads packages do not reflect the memory hierarchy of a NUMA multiprocessor in their data structures and policies, we have studied three queueing systems with processor and node local structures, offering a locality scheduling based on data about thread locality and timing, gathered at runtime. The proposed strategies range from hierarchical hash queues with aging strategies to a tree structure of priority queues. Preswitching has to be evaluated as an efficient switching technique avoiding race conditions on multiprocessors.
Because scheduling decisions are known at the application level, data of the thread scheduled to run after the next switch, can be fetched concurrently into the local cache. Triggering asynchronous prefetch operations helps to overlap computation and communication and therefore to hide memory latency.
All this efforts cannot be successful, if the kernel threads, used as virtual processors of the runtime system, are scheduled by the kernel without knowledge of special scheduling needs of an application. We have sketched a processor server dividing the available processors among multiple applications according to locality information and processor requests. Because only one kernel thread runs on top of each processor using this policy, an advanced kernel interface for an information exchange between runtime system and kernel prevents a loss in parallelism if threads block inside the kernel.
The implications of memory locality in scheduling policies for NUMA architectures are in the focus of our efforts; we currently investigate and implement novel approaches in data structures and interaction among kernel, processor server and runtime system using the Convex SPP hardware and softare.

5 Acknowledgments

We would like to thank Thomas Eirich, Matthias Gente, Franz Hauck, Fridolin Hofmann, Christoph Koppe and Armin Rueth for many useful and insightful comments. We would also like to thank the Convex operating system group for providing us with their Convex SPP software and their background in OS development . The research project in cooperation with Convex Computer Corp. is supported by the Bavarian Consortium for High Performance Scientific Computing (FORTWIHR).

6 References

[1] T. Anderson; E. Lazowska; H. Levy: The Performance Implication of Thread Management Alternatives for Shared-Memory Multiprocessors. In ACM Trans. on Computers, 38(12), Dec. 1989, pp. 1631-1644

[2] T. E. Anderson; et al.: Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In ACM Transactions on Computer Systems, 10(1), Feb. 1992, pp. 53-79

[3] F. Bellosa: Parallele leichtgewichtige Prozesse zur Implementierung adaptiver numerischer Verfahren, Universität Erlangen-Nürnberg, IMMD IV, no. TR-I4-2-94, Jan. 1994

[4] D. L. Black: Scheduling and Resource Management Techniques for Multiprocessors, Phd Thesis, Carnegie-Mellon University, July 1990

[5] C. McCann; R. Vaswani; J. Zahorjan: A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors. In ACM Trans. on Comp. Sys., 11(2), May 1993, pp. 146-178

[6] Convex: Exemplar Architecure, Convex Press, Nov. 1994

[7] J. Driscoll; H. Gabow; R. Shrairman; R. Tarjan: Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation, Communications of the ACM, 32:1343-1354, 1988

[10] D. Keppel: Tools and Techniques for Building Fast Portable Threads Packages, University of Washington, TR UWCSE 93-05-06, May 1993

[11] D. Knuth: The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Mass. , 1973

[12] C. Koppe: Sleeping Threads: A Kernel Mechanism for Support of Efficient User Level Threads, submitted to the 7th IASTED -ISMM International Conference: Parallel and Distributed Computing and Systems (PDCS'95), Oct 95

[13] T. J. LeBlanc; et al.: First-Class User-Level Threads. In Operating Systems Review, 25(5), 1991, pp. 110-121

[14] B. Lim; A. Agarwal: Waiting Algorithms for Synchronizations in Large-Scale Multiprocessors. In ACM Transactions on Computer Systems, 11(1), Aug. 1993, pp. 253-297

[15] E. Markatos; T. LeBlanc: Locality-Based Scheduling for Shared-Memory Multiprocessors, Phd Thesis, Computer Science Department, Univ. Rochester, 1993

[17] J. K. Ousterhout: Scheduling Techniques for Concurrent Systems. In Third International Conference on Distributed Computing Systems, 1982, pp. 22-30

[18] U. Reder: Implementierung eines effizienten Prozeßumschalters auf Benutzerebene, Studienarbeit am IMMD IV, University Erlangen, 3/1995

[19] Ulrich Rüde: On the multilevel adaptive iterative method, SIAM Journal on Scientific and Statistical Computing, Vol. 15, 1994

[20] A. Tucker: Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors, Phd Thesis, Department of Computer Science, Stanford University, CS-TN-94-4, Dec. 1993