Locality-Information-Based Scheduling in Shared-Memory Multiprocessors

Frank Bellosa

University of Erlangen-Nürnberg
Computer Science Department - Operating Systems - IMMD IV
Martensstraße. 1, 91058 Erlangen, Germany
email: bellosa@informatik.uni-erlangen.de

Abstract

Lightweight threads have become a common abstraction in the field of programming languages and operating systems. This paper examines the performance implications of locality information usage in thread scheduling algorithms for scalable shared-memory multiprocessors. The elements of a distributed scheduler using all available locality information as well as experimental measurements are presented.
Most shared-memory multiprocessors use multiple stages of caches to hide latency. 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. Per-processor data structures avoid lock contention and help to reduce memory traffic. While CPU utilization of processes still determines scheduling decisions of contemporary schedulers, we propose novel scheduling policies based on cache miss rates and information about synchronization. All data gathered at runtime are transformed into affinity values inside a metric space, so that threads migrate near to their (sub)optimal operation points defined by location and timing of execution. The distribution of data structures and the usage of locality information characterizes the proposed memory-conscious scheduling architecture. A prototype implementation shows that a locality-conscious scheduler outperforms centralized and distributed approaches ignoring locality information.

Keywords: affinity scheduling, cache-miss information, NUMA architectures, operating systems.

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, relieving the application designer of data distribution and coherency maintenance. But cache locality, load balancing and scheduling are still of crucial importance.

Large caches used in scalable shared-memory architectures can avoid high memory access time only if data is referenced within the address scope of the cache. Consequently, locality is the key issue in multiprocessor performance. One goal of software development is a high degree of locality from the system up to the application level. Even if application designers develop code with high locality, the impact of caches is reduced when scheduling policies ignore locality information. Disregarding locality, the scheduler will initiate switches into uncached process contexts. The consequences are cache and TLB misses for the processor in question and cache line invalidations in caches of other processors.

NUMA architectures like KSR or Convex SPP already provide locality information gathered by special processor monitors or by the cache coherence hardware. The latest processor generations - e.g. HPPA 8000, MIPS R10000 or Ultra SPARC - include a monitoring unit. A processor monitor can count events like read/write cache misses and processor stall cycles due to load and store operations.
Locality information about each process/thread, such as the duration of the last active period, the cache miss rate, the processor stall time, and the processor of last execution, can be used to calculate an affinity value. Furthermore, cooperating threads can be identified by synchronization events and collocated at the same processor, whereas a trade-off between collocation and load balance has to be provided.

The parallelism expressed by "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, deletion and context switching invoke complex kernel activities and imply 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 kernel threads only offer a middle-grained programming model, because thread management implies expen-

This work was supported in part by the Bavarian Consortium for High Performance Scientific Computing (FORTWIHR) and Convex Computer Corporation.

sive protected system calls. The potential benefit of using locality information increases with the frequency of scheduling decisions, because the scheduler is the instance evaluating locality information. Consequently, the benefit of using locality information in the kernel will be limited by the low frequency of scheduling decisions.

By moving thread management and synchronization to the 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 are 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 to achieve low thread management overhead, if a lean scheduler is sufficient for an application.
- Data structures for processes and threads are deeply rooted in most kernels. Only the user level offers the necessary flexibility in adapting data structures to the degree of parallelism inherent in an application ranging from several 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. Efficient user-level threads are predestined for fine-grained parallel applications. User-level schedulers make frequent context switches affordable and therefore draw most profit from the use of locality information if the lifetime of cachelines exceeds scheduling cycles.

Problems with this two-level scheduling arise from the interference of scheduling policies on different levels of control without any coordination or communication. A loss of parallelism and the occurrence of a deadlock situation is possible due to blocking system calls invoked by user-level threads. Solutions to these problems are discussed in [11].

In this paper we propose a non-preemptive user-level threads package with an application interface to trigger prefetch operations to hide memory latencies based on scheduling decisions of the runtime system. We outline several scheduling policies using locality information and present results from a prototype implementation on a Convex SPP 1000/XA. This machine can be characterized as a cache-coherent NUMA multiprocessor.

The rest of the paper is organized as follows. Section 2 describes the architecture of the **E**rlangen **Li**ghtweight **T**hread **E**nvironment (**ELiTE**), a scheduling architecture for cache-coherent NUMA multiprocessors developed and implemented at the University of Erlangen. Several affinity policies are evaluated in Section 3. Finally, we conclude in Section 4.

2 Erlangen Lightweight Thread Environment (ELiTE)

In 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. Cache affinity scheduling for bus-based multiprocessors has been investigated [17][22] in detail because cache architectures become more and more dominant. The decisions within this type of scheduling base on CPU utilization and information about the processor where a specific thread was most recently executed. State timing information from each process is additionally be used e.g. in SGI's IRIX operating system [3]. Our approach to memory-conscious scheduling goes beyond the use of information about timing and execution location by using cache miss information for each level of the memory hierarchy.

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 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, these scheduling policies focus on a high processor utilization while reducing bus contention [1].
The focus of thread scheduling has to move when we look at scalable shared-memory architectures with non-uniform memory access. Modern superscalar RISC-based processors are able to perform multiple operations per clock cycle while simultaneously performing a load/store operation to the processor cache. A multiprocessor system can only take advantage of this immense computing power if the processors can be supplied with data in time. The bandwidth of interconnection networks is no longer a bottleneck for today's scalable parallel processors (e.g. the Scalable Coherent Interface (SCI) of the Convex SPP has a bandwidth of 2.8 GBytes/s). But switches as well as affordable dynamic memory cause a latency of about a hundred nanoseconds, while processor cycles need only a few nanoseconds. The consequence of this discrepancy is that scheduling policies for NUMA architectures have to satisfy three essential design goals:

(1) **Distributed Scheduling:** Data structures of the scheduler (run queues, synchronization objects and pools for reusable memory regions) are distributed. There are no global structures with the potential risk of contention.

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

(3) **Latency Hiding:** Prefetch operations cause overlapping of computation and communication.

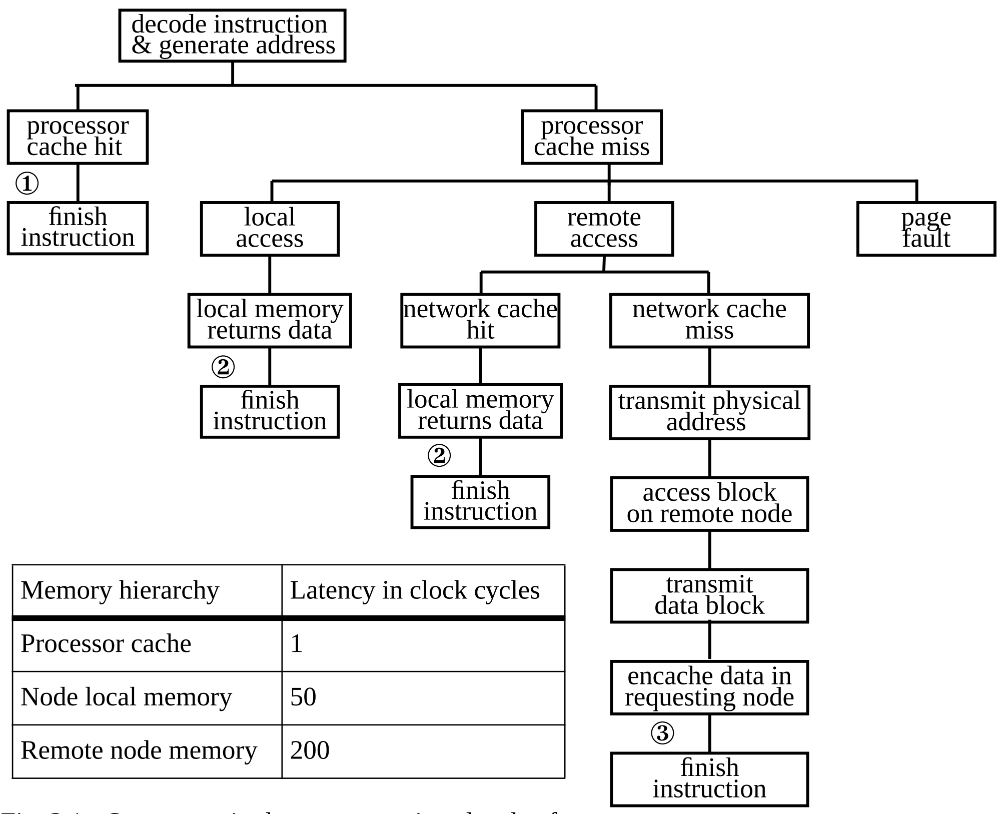
As contemporary threads packages, developed for use on shared-memory multiprocessors with a modest number of processors, have design goals which cannot be applied to scalable NUMA multiprocessors with a high number of processors, novel scheduling architectures have to be designed. After presenting the architecture of the Convex SPP, a cache-coherent NUMA multiprocessor, we describe the architecture and implementation details of the ELiTE runtime system.

2.1 Architecture of the CONVEX SPP

The Convex Exemplar Architecture [7] implemented in the Convex SPP multiprocessor is a representative of cache-coherent NUMA architectures. A symmetric multiprocessor called hypernode is the building block of 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 on a single hypernode can access up to two GBytes of main memory over a non-blocking crossbar switch. The memory in remote hypernodes can be accessed via the interconnect. To reduce network traffic, part of the memory is configured as a network cache with 64-byte cachelines. Load/store operations step through various stages depending on the locality of the referenced memory region (see figure 2.1).

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 and 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 precision of 1µs.



	Memory hierarchy	Latency in clock cycles
①	Processor cache	1
②	Node local memory	50
③	Remote node memory	200

Fig. 2.1 Stages required to access various levels of the memory hierarchy

The operating system is a MACH 3.0 microkernel with a HP/UX-compatible Unix server on top. It provides the system call interface from Hewlett-Packard's Unix and an additional system interface to create and control kernel threads.

2.2 Scheduling Architecture of ELiTE

The overhead associated with lightweight processes goes beyond the cost of thread management, because of memory transfers between the various levels of the memory hierarchy. We present a scheduling architecture outlined in [5], refined and implemented in [18].

The following architectural features characterize the ELiTE runtime system:

– Division of thread control block (TCB) and stack allocation (see figure 2.2):
  Each processor manages its own pool of free TCBs ① and stacks ③. If a new thread is created, the creating processor allocates and initializes a free TCB. After initialization the TCB is enqueued in a startqueue ②. A processor with an empty runqueue ④ fetches a TCB from a startqueue and can run the thread after allocating and initializing a stack. By separating TCB and stack allocation, memory objects of a thread, which have to be modified, will be allocated from memory pools managed and touched by the modifying processor. The consequence is a high cache reusage and a low cache miss rate.

– Pool and startqueue hierarchy correspond to memory hierarchy (see figure 2.2):
  The number of startqueue entries is limited on the first level (processor level) and the second level (node level). If an overflow occurs, the TCB becomes enqueued in the next level. The consequence is a high degree of locality with an implicit load distribution.
  When the stack- and TCB-pools of the first level (processor level) are empty, new memory objects will be enqueued from the second level. Likewise, memory objects will be moved from the first level to the second level when an overflow occurs. If a pool on the second level is empty, new memory will be allocated from global memory. Therefore, global pools are not useful. The consequence of this strategy is high reusage of local memory while keeping memory allocations to a minimum.

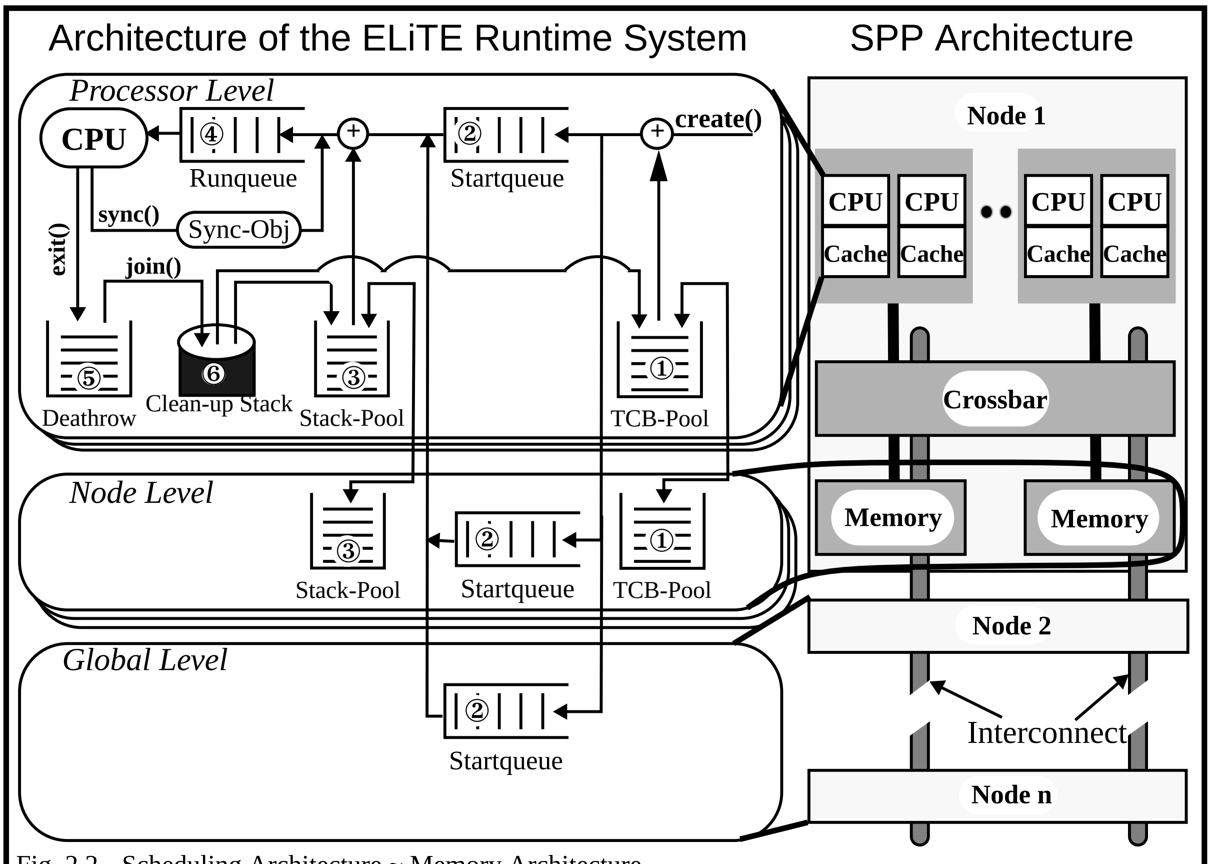


Fig. 2.2 Scheduling Architecture ≈ Memory Architecture

– Local runqueues with load balancing:
  Each processor manages its own priority runqueue (see figure 2.2 ④). The priority of a thread depends on its affinity. The scheduler prefers threads with high affinity. A processor with an empty runqueue, finding no threads in the startqueues, scans the runqueues of the processors in the same node and finally the runqueues of all other processors for runable processes. The advantages of local runqueues are high cache reusage, low data cache invalidation and minimal contention for queue locks.

– Local deathrow with local clean-up stack:
  When a thread exits, its context is stored in the deathrow (see figure 2.2 ⑤) of the processor. The processor executing the join() reads the exit status of the joined thread and pushes an entry on the clean-up stack (see figure 2.2 ⑥) of the processor, which executed the exit(). The processors periodically scan their local clean-up stacks and remove the contexts of joined threads from the deathrow and push memory objects (stacks and TCBs) into the local memory pools. Because processors executing a join never modify the deathrow and memory pools, cache invalidations can be avoided and the memory locality will be preserved. Cache misses are reduced to a minimum, because the push onto the clean-up stack concerns only a single cacheline.

We measured the number of threads a single thread can fork and join if *n* CPUs can start and run the created threads. We compared an approach with central pools and another with distributed pools (see figure 2.3). The results show that the

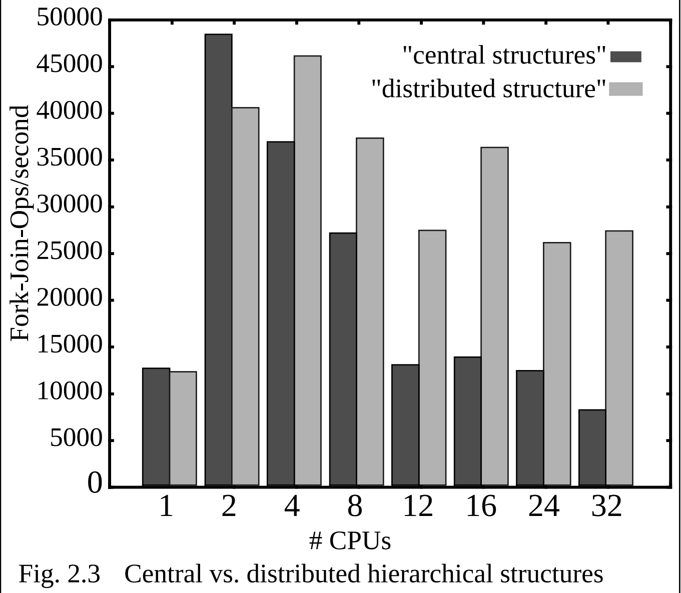


Fig. 2.3 Central vs. distributed hierarchical structures

fork-join-rate of the centralized approach drops when using 4-8 processors due to lock contention and dramatically drops when using more than 8 processors (more than one hypernode) due to allocation of stacks not cached on the local node. The distributed approach does not scale because of the limited fork rate of the single forking thread. But there is no severe performance degradation, because all stacks are allocated from local pools with encached memory. Furthermore, locking of central structures (pools and queues) and remote memory access can be reduced to a minimum by the mechanisms of local deathrow and clean-up stack.

– Distributed synchronization objects with local wake-up stack:
  Unlike common UNIX sleep queues with hashed entries [13], the ELiTE runtime system binds blocked threads to synchronization objects (see figure 2.4 ②). If a process becomes unblocked, a reference to its TCB will be pushed on the wake-up stack (see figure 2.4 ③). Likewise the deathrow management, the processors scan their local wake-up stacks periodically and enqueue unblocked threads in the local runqueue (see figure 2.4 ①).

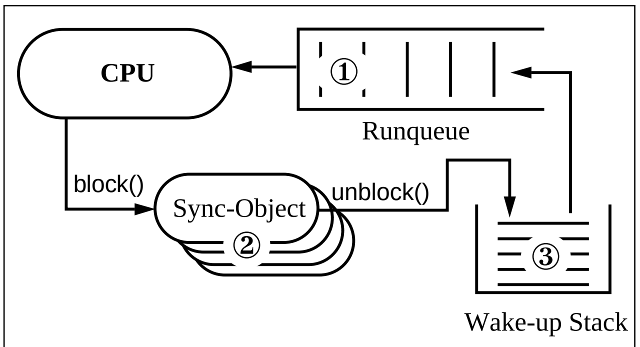


Fig. 2.4 Distributed Lock Structures

2.3 Implementation Details

2.3.1 Fast context switch

A fast context switch free of race conditions is the basis of most synchronization mechanisms inside a runtime system.
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 context switching, we have investigated two models:

– **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. However, this simple and secure switching model is very time consuming, as two context switches are necessary per thread switch.

– **Preswitch**: After saving the state of the old thread, the stack of the new thread is used to enqueue the TCB of the old thread 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 is used in ELiTE. Based on the QuickThreads package of the University of Washington [9], which provides the preswitch model for various processor architectures, we have ported QuickThreads to the HPPA-RISC processor architecture. On a CONVEX SPP using this type of processor, the following times for a context switch can be reported:

Operation	Clock Cycles
Context switch between threads with all data in cache	153
Context switch between threads in the same node	1122
Context switch between threads in different 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 almost exactly the proportions expected to result from a memory latency of 1/50/200 cycles and 32/(64) Bytes (network-) cache lines. Most of the time is spent saving and restoring the callee-saves registers. The consequence is that switching can only be optimized by reducing the number of registers 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 not 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.

2.3.2 Fast synchronization

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-phase waiting algorithm combines the advantage of polling and signalling. A thread blocks after a fixed polling interval. The polling threshold depends on the overhead of blocking. The results in [14] are relevant for us, because the timing behavior of the MIT Alewife multiprocessors bears great resemblance to RISC-based scalable architectures like Convex SPP or KSR1/2.

In the ELiTE runtime system we use two-phase locking with a fixed number of spin cycles.
We measure the clock cycles a lock is held, and calculate the average duration (in clock cycles) of the last 8 acquirements of each specific lock.

Choice of spin cycles	Number of spin cycles
Average lock-holding cycles > default	0 (Block immediately)
Average lock-holding cycles < default	Default value

If the average of lock-holding cycles exceeds a proposed value (default is 50% of the cycles for a context switch), we block at once. Otherwise we spin the default number of cycles given in the startup. To reduce memory accesses while spinning we use exponential backoff. For details refer to [18].
We have measured the peak performance for synchronization by starting 4096 micro-threads (8 kBytes stack and no workload) doing nothing but synchronizing. The total amount of memory is about 4096*8192 KBytes = 32 MBytes.
With central queues we see a severe performance loss due to lock contention and data cache corruption as a consequence of non local memory accesses (see figure 2.5)

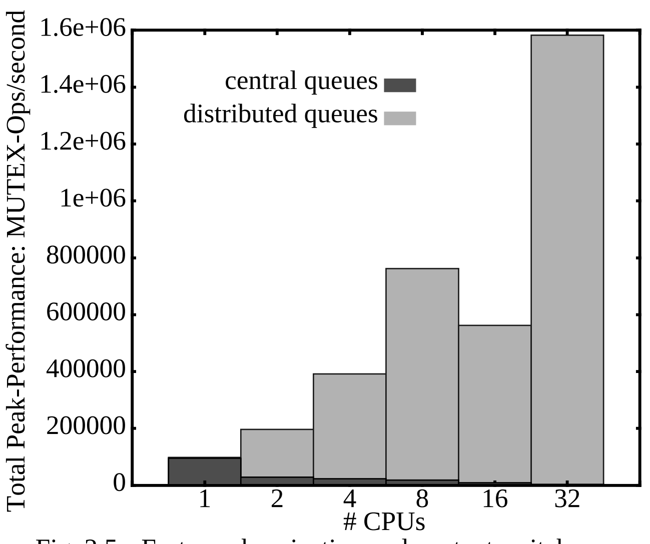


Fig. 2.5 Fast synchronization and context switch with 4096 threads

Using a distributed approach, the time for a synchronization depends on the time to save/restore the stacks into the processor/network-caches and to access the synchronization objects. We reach the peak performance of about 100000 synchronizations/second per processor (1-8 processors in figure 2.5), if all data can be held in the processor caches. If the stacks can be stored in the processor caches, but the synchronization objects have to be touched in part from multiple nodes, the synchronization performance deteriorates to about 50000 synchronizations/second per processor (32 processors in figure 2.5). If the stacks do not fit into the processor-caches (16 processors), they are stored in the network caches (= local memory). This effect reduces the performance to about 40000 synchronizations/second per processor.

The scheduling approach presented in this paper considers the locality behavior of individual threads at runtime. This approach is possible if the hardware provides information about cache misses. An additional optimization is possible, if we look at the interaction of threads on the same memory regions, by comparing their working set. In [16] TLB information is used to find cooperating processes on kernel level. For fast interaction a user-level readable TLB would be necessary, accessible as fast as the processor cache and with a fine resolution in the range of 1 kByte. This could be a feature of new processor generations. The knowledge of synchronization events is an alternative way to identify cooperating threads. This approach will be discussed in section 3.3.

2.3.3 Queue Structures

The decisions of a memory-conscious scheduler depend on the affinity of the threads to a specific memory region, e.g. cache or node local memory. Consequently, threads have to be enqueued according to their affinity. Several data structures for priority queues exist [10], where Fibonacci heaps and relaxed heaps [8] need only O(log *#threads*) operations for the time-critical ‘find_and_remove_maximum’-operation, which is necessary to identify and extract the processes with maximum locality from the priority queue. But heap-structures are not suitable for runqueues, because heaps can not be partitioned fast enough in the case of load imbalance.
Priority queues implemented as binary trees make fast de- and enqueueing possible and can be divided very easily into partitions with entries of high or low locality.

2.3.4 Kernel Interface

User-level runtime systems use kernel threads as virtual processors, assuming an equivalence of physical and virtual processor. This assumption does not hold, because events like page faults, I/O and system calls block the virtual processors. The equivalence of physical and virtual processors can be achieved by notification of the user-level threads package, which can thus react adequately.
Scheduler Activations, proposed in [2], use kernel threads to upcall the runtime system. This strategy suffers from the fact that a free processor is needed to run the kernel thread upcalling the user level. But there is no free processor in the case of a request for suspension of a virtual processor. The consequence is an expensive context switch on kernel level causing TLB misses and data cache corruption.
In [12] and [22] communication mechanisms between the kernel and a user-level thread library are proposed to reduce 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. Software interrupts signal to the thread package whenever a scheduling decision may be required. For example, polling of shared memory in a safe suspension point is used to instruct the runtime system to suspend a thread, while signalling is 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 of 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 [11] and is 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 of the deblocking via a shared page or shared-memory segment. If this deblocked user-level thread is selected for execution, the corresponding kernel thread initiates 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.

2.3.5 Application 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 [4][15], 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 cache before the context switch occurs.

3 Affinity Scheduling

The performance of a computer is considerably influenced by the fast supply of data to the available processing units. Only if the data essential for operation is cached in fast memory can the processor work without latency and contention. Affinity scheduling tries to prefer processes with a high amount of cached data in order to increase throughput.

Besides information about processor number and time behavior, we use information about data locality in our scheduling architecture. Locality information about each process/thread like the cache miss rate, the processor stall time, and the processor of last execution can be used to calculate an affinity value. A prerequisite is a computer architecture providing information about cache misses and CPU latencies due to memory access. Contemporary architectures like Convex SPP and KSR1/2 provide this information, future processor architectures like HPPA 8000, MIPS R10000 or Ultra SPARC will gather this information on chip. This information is usually only used for off-line profiling. But why should we ignore information for optimizing the behavior of multithreaded applications at runtime?

In the next subsections we describe several affinity strategies and the prospect of the proposed technique.

3.1 Scheduling Strategies

We have designed and implemented a user-level runtime system, offering the possibility to easily import new affinity strategies. The strategies examined are listed in the table below:

- Using virtual time stamps, each thread is assigned a sequence number after its run. This strategy does not need cache miss information, and can be used on every type of hardware. Threads with the highest time stamp given by the same processor will be preferred. If a fast global time source with high resolution is included in the hardware, more precise timing strategies can be used [3].
- The *Minimum Misses* strategy compares the number of cache misses during the last run. The thread with the lowest number of cache misses is preferred. This strategy favors threads that block frequently, since they have shorter runs and few cache misses.

Scheduling strategy	Basis of decision	Policy
No Affinity	Processor location	LIFO
Virtual Time	Sequence numbers	Most recently run
Minimum Misses	Cache misses	Thread with minimal # of cache misses
Reload Transient	Cache misses	Minimal reload transient based on a Markov chain

- The reload transient model [19] is more complex, but offers some potential. We refer to the working set of a thread that is present in the cache as its footprint in the cache. The reload transient is defined as the cost to establish the footprint of a thread after restarting it. We use a Markov model to calculate the footprint of each thread. In the state transition diagram in figure 3.1. each node *V* represents a state with *v* valid cache lines of a thread residing in the cache. Our direct mapped cache consists of *N* cache lines.

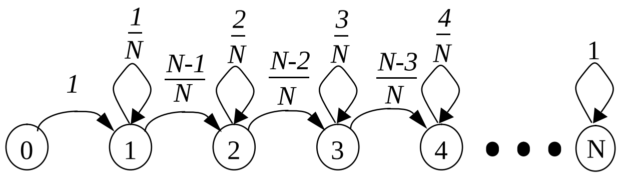


Fig. 3.1 Markov chain of valid cache lines

The probability to increment the number of valid cache lines as a consequence of a cache miss during the run of a thread is $(N-v)/N$. The probability that a cache miss hits a valid cache line is v/N. We can generate the transition probability matrix and calculate the *M*-step transition probability matrix for each node *V*, describing the probability for a thread to have a certain number of valid cache lines in the cache after *M* cache misses happened.

Equivalently, we can calculate the transition probability matrix for a blocked thread. The number of resident cache lines is decremented with a certain probability for each cache miss caused by the intermediate run of an other thread .

Using these probabilities we calculate the expected footprint size *F* after each run. The size of the footprint depends on the number of cache lines *v* still valid at startup and the number of cache misses *cm* which occurred during this run $E[F|V = v, M = cm]$.

We can also compute the expected number of lines *V* still valid depending on the footprint *f* of a specific thread and the number of cache misses *om* caused by other threads $E[V|F = f, M = om]$.

Basis of our cache affinity calculation is the expected footprint size *F* of a thread in its last run and the expected number of valid cache lines *V* before its potential run. The reload transient is defined as the expected number of cache misses $E[M|F = f, V = v]$ when rebuilding the working set of a rescheduled thread. Our scheduling policy selects the thread with minimal reload transient. A characteristic of our apporach is that the order among runable threads remains the same even if the scheduler enqueues new runable threads. Consequently the reload transient has only to be determined, when a thread is deblocked and enqueued. By pre-computing expected reload transients for a set of footprints and a set of different cache miss numbers the priority calculation of the scheduler can be reduced to a simple table lookup [18].

3.2 Interpretation of Measurements

We have implemented and tested the proposed strategies as part of the ELiTE architecture on a Convex SPP 1000/XA in a range from 1 to 32 processors. The test environment includes synthetic tests with artificial workloads as well as real-world numerical kernels like gauss elimination, LU-decomposition and adaptive solvers on irregular grids[15][4]. All measurements show that a fine-grained parallelization does not inherently imply low performance. On the contrary, a fine-grained numerical efficient algorithm outperforms most conventional methods, because fine-gained parallelism implies a high data locality in most cases. This locality can be used by a sophisticated scheduler to achieve good overall performance even if we register some overhead for complex strategies. But the impact of scheduling in a fine-grained environment affects performance much more than with a coarse-grained approach. The runtime of a fine-grained LU-decomposition can vary by a factor of ten, depending on the scheduling architecture.

With a synthetic workload of 1024 threads, each snooping through a working set of 64 kBytes between synchronizations, we can get a good impression of affinity strategies (see figure 3.2.1). We measure how many synchronizations can be executed in one second with different numbers of processors. This example resembles parallel numerical applications with regular execution order like iterative solvers on block-structured grids.

Applications with irregular execution order make high demands on the scheduling policy. We measure the number of smoothing operations per second on an unstructured grid executed by a full-adaptive iterative solver [15]. To demonstrate the influence of the scheduling strategy on the application performance, we compare the smoothing rate of the proposed affinity strategies with the *No Affinity* strategy. A relative smoothing performance of 2 means, that the adaptive solver executes twice the number of smoothing operation under the affinity scheduler compared to an execution under the *No Affinity* scheduler (see figure 3.2.2)

- *No Affinity* will be outperformed in general by all strategies using locality information
- The simple *Minimal Misses* strategy performs quite well in cases with homogeneous execution behaviour. As this strategy favors threads that block frequently, anomalous behavior is possible if some threads aquire locks during the polling interval whereas other threads block.
- The *Virtual time* approach only uses timing behavior and the processor number of the last run. It never shows anomalous behavior and performs very efficiently with moderate processor numbers (1-16 processors) because it offers the minimal overhead compared to the other strategies. It should be the policy of choice for UMA architectures not offering cache miss information (see [3])
- The *Reload Transient* strategy shows the best performance in multinode architectures with non-uniform memory access. The overhead of gathering cache miss information and computing the expected working sets is only justified when memory latency really strikes. This is the case on all contemporary and future scalable parallel processors like Convex SPP, KSR 1/2, SUN Ultra MPP, Sequent NUMA-Q and multiprocessors coupled with SCI-Hardware (SCI = Scalable Coherent Interface).

The hardware of the Convex SPP 1000 used for our implementation cannot distinguish between processor- and network-cache misses, but these two types of cache miss differ by a factor of 4 in their penalty. Coming SPP generations offer information about both types of cache fault, which permits a much better calculation of the working set and will clearly outperform all other proposed affinity techniques.

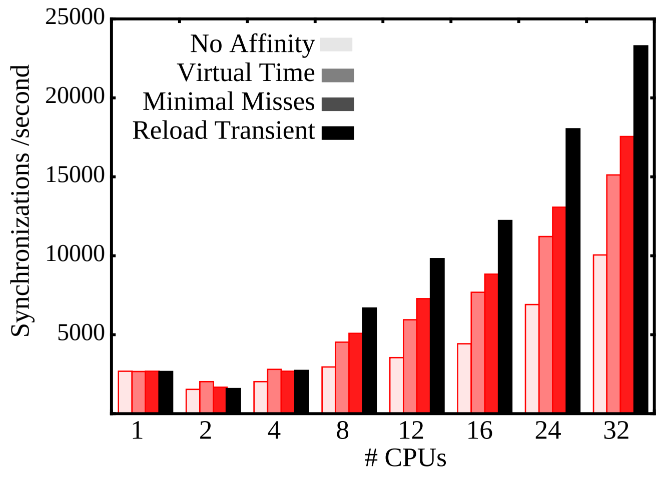


Fig. 3.2.1 Performance of applications with regular execution order

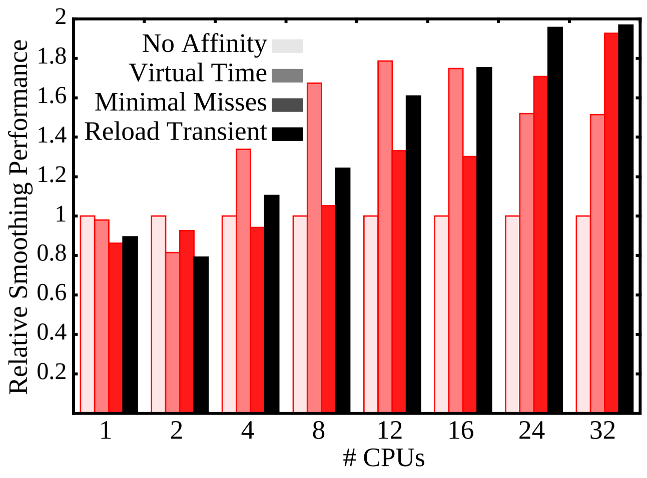


Fig. 3.2.2 Performance of adaptive applications with irregular execution order

3.3 Towards Optimal Affinity Scheduling

Scheduling policies should use all available information to calculate the best possible process mapping. For optimal process scheduling, knowledge about synchronizing threads is necessary. Cooperating threads can be identified by synchronization events and collocated at the same processor, whereas a compromise between collocation and load balance has to be made. To decide an ideal point defined by the location and timing of execution, we define gravitational and repulsive forces between threads by the frequency and extent of information sharing.

The forces influence affinity in a metric space, so that threads migrate near to their (sub)optimal operation point. This affinity model resembles the computational field model for migrating objects proposed in [20].

A thread *A* running on processor *X* with a great cache affinity exerts a gravitational force on thread *B* running on processor *Y* with just a small cache affinity. Thread *A* exerts the force by increasing the affinity of thread *B* to processor *X* each time *A* synchronizes with *B*. The value used for increasing the affinity depends on the size of the memory region shared by the two threads (see figure 3.3). In the ELiTE runtime system we use the number of cache misses occurring during the modification of the shared-memory region as the basis of the affinity adjustment.

Cooperating threads with a high compute load and rare synchronization events exert repulsive forces on each other (see figure 3.3). Threads showing this behavior can be recognized by a high number of cache misses and a long time-frame between blocking events. If these threads run on the same processor, they decrease the affinity of their synchronization partner. Threads with a low affinity can be caught by an idle processor.

Threads with low affinity sharing a memory region which is large relative to the private working set exert aggregative forces on each other (see figure 3.3). Before unblocking, the affinity values of a synchronization partner will be modified such that the partner will run up on the same processor as the unblocking thread. The aggregative force is influenced by the number of cache misses on shared and thread-private memory regions.

The proposed strategy has the potential to come closer to the optimum operation point of all threads concerning location and timing of execution. By introducing gravitational and aggregative forces, the number of cache misses of cooperating threads can be reduced, while compute-intensive threads with low synchronization rate exerting repulsive forces can be distributed very easily.

The fact that the hardware of the Convex SPP 1000 cannot distinguish between processor- and network-cache misses, together with the high costs for reading the cache miss counters (costs of one read are equivalent to a user-level context switch), means that there are no publishable results of our implementation. Coming processor generations with fast readable on-chip miss counters will allow reasonable measurements using this promising scheduling strategy.

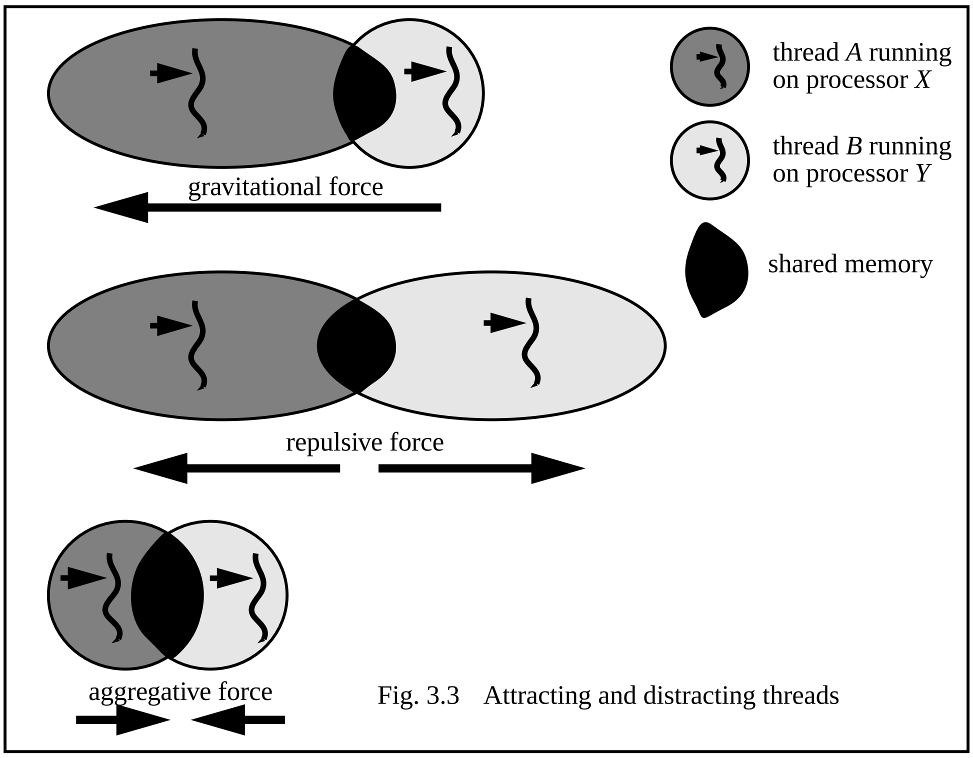

thread *A* running on processor *X*
thread *B* running on processor *Y*
shared memory
gravitational force
repulsive force
aggregative force

Fig. 3.3 Attracting and distracting threads

4 Summary and Conclusions

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

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. Distributed data structures like those proposed in the ELiTE architecture are an absolute necessity. Scheduling strategies using locality information improve cache locality and therefore throughput. Strategies based on Markov chains offer the best process reordering in scalable architectures with non-uniform memory access despite their algorithmic overhead.

The trade-off between scheduling overhead and performance gain due to better locality will favor complex strategies using cache miss information particularly in architectures with high memory latency and large caches. Consequently, the proposed scheduling techniques can be used from the desktop to the supercomputer. Further research will be dedicated to novel strategies using all available locality information.

5 Acknowledgments

I would like to thank Thomas Eirich (IBM Zurich Research Laboratory), Franz Hauck (Vrije Universiteit Amsterdam), Matthias Gente, Fridolin Hofmann, Christoph Koppe, Armin Rueth and Michael Schröder (University of Erlangen) for many useful and insightful comments. Special thanks to Martin Steckermeier for designing and implementing the Mthreads runtime system. I would also like to thank the Convex operating system group for providing their Convex SPP software and their background in OS development. The ELiTE 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, *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. *ACM Transactions on Computer Systems*, 10(1), Feb. 1992, pp. 53-79

[3] J. Barton, N. Bitar - SGI Corp., A Scalable Multi-Discipline, Multiple-Processor Scheduling Framework for IRIX", In *Proc. of IPPS'95 Workshop of Job Scheduling Strategies for Parallel Processing*, LNCS 949, Apr. 1995

[4] F. Bellosa: Implementierung adaptiver Verfahren auf komplexen Geometrien mit leichtgewichtigen Prozessen, University Erlangen-Nürnberg, IMMD IV, TR-I4-94-07, Oct. 1994

[5] F.Bellosa: Memory-Conscious Scheduling and Processor Allocation on NUMA Architectures, University Erlangen-Nürnberg, IMMD IV, TR-I4-95-06, May 1995

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

[7] Convex: Exemplar SPP1000/1200 Architecture, Convex Press, May 1995

[8] 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

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

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

[11] C. Koppe: Sleeping Threads: A Kernel Mechanism for Support of Efficient User-Level Threads, In *Proc. of International Conference of Parallel and Distributed Computing and Systems PDCS'95*, Washington D.C., Oct. 95

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

[13] S. Leffler: The Design and Implementation of the 4.3BSD UNIX Operating System, Addison-Wesley, 1990

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

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

[16] P. Sabalvarro, W. Weihl: Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors, In *Proc. of IPPS'95 Workshop of Job Scheduling Strategies for Parallel Processing*, LNCS 949, April 1995

[17] S. Squillante; E. D. Lazowska: Using Processor Cache Affinity in Shared-Memory Multiprocessor, Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2),Feb. 1993, pp.131-143

[18] M. Steckermeier: Using Locality Information in User-Level Scheduling, University Erlangen-Nürnberg, IMMD IV, TR-I4-95-14, Dec. 1995

[19] D. Thiebaut, H. Stone: Footprints in the Cache, *ACM Transactions on Computer Systems*, 5(4), Nov. 1987, pp. 305-329

[20] M. Tokoro: Computational Field Model: Toward a New Computation Model/Methodology for Open Distributed Environment, Sony Computer Science Laboratory Inc., SCSL-TR-90-006, June 1990

[21] J. Torellas, A. Tucker, A. Gupta: Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors, *Journal of Parallel and Distributed Computing*, Vol. 24, 1995, pp. 139-151

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