Efficient Runtime Thread Management for the Nano-Threads Programming Model

Dimitrios S. Nikolopoulos, Eleftherios D. Polychronopoulos
and Theodore S. Papatheodorou

High Performance Computing Architectures Laboratory
Department of Computer Engineering and Informatics
University of Patras
Rio 26500, Patras, Greece

Abstract. The nano-threads programming model was proposed to effectively integrate multiprogramming on shared-memory multiprocessors, with the exploitation of fine-grain parallelism from standard applications. A prerequisite for the applicability of the nano-threads programming model is the ability of the runtime environment to manage parallelism at any level of granularity with minimal overheads. In this paper, we introduce runtime techniques for efficient memory management and user-level scheduling in an experimental runtime system designed to support the nano-threads programming model. We evaluate the exploitation of processor affinity for the management of nano-thread contexts, and the use of hierarchical queues to implement user-level scheduling strategies for applications with inherent multilevel parallelism. The proposed mechanisms attempt to obtain maximum benefits from data locality on cache-coherent NUMA multiprocessors. Through the use of synthetic benchmarks, we find that our mechanism for memory management in the runtime system reduces overheads by 52% on average, compared to other known mechanisms. The use of hierarchical queues gives significant performance improvements between 17% and 40%, compared to scheduling strategies that use local queues.

1 Introduction

The nano-threads programming model, introduced in [Poly93], integrates parallelizing compilers technology, user-level multithreading, and operating system support, to provide an efficient execution environment for parallel jobs on multiprogrammed shared-memory multiprocessors. The model uses lightweight user-level threads for the effective exploitation of fine-grain parallelism at different levels of granularity. Multithreading is considered not only as a way to express parallelism, but also as a mechanism to support multiprogramming. The model introduces the concept of *scalable binaries*, i.e., autoscheduled applications, which adapt to changes of the resources allocated to them from the operating system. Applications adapt dynamically at runtime, by adjusting their degree of parallelism. Construction and efficient execution of scalable binaries is achieved through close coordination between the compiler, the multithreading runtime system, and the operating system.

This work was supported by the NANOS project (ESPRIT No. 21907)

The execution environment of the nano-threads programming model provides runtime support to create and schedule parallel threads of execution called nano-threads. Runtime support can be provided either by a runtime library used at the compiler backend, or by a mechanism which directly injects multithreading code into applications. Both approaches should use mechanisms that introduce minimal overheads for nano-threads management, since exploitation of fine-grain parallelism is desirable. Furthermore, the runtime system should provide a convenient framework for implementing efficient user-level schedulers. Among other properties, like preserving data locality and load balancing, the user-level schedulers should be able to handle multilevel and unstructured parallelism.

Martorell et.al. [Mart95, Mart96], have recently presented NthLib, a prototype runtime library that supports the nano-threads programming model. NthLib implements user-level multithreading and provides primitives to translate and schedule task graphs. In this paper, we propose two efficient thread management alternatives, which we implemented in the context of NthLib. The first alternative aims at reducing the overheads of memory management in the runtime system. We introduce a technique that exploits the affinity of threads for processors during the initialization of nano-thread contexts. The second alternative provides a framework for the development of efficient scheduling strategies for parallel applications running in the nano-threads execution environment. We introduce a mechanism for user-level scheduling, based on hierarchical ready queues. This mechanism can be used to efficiently schedule coarse and fine-grain nano-threads, derived from the exploitation of different levels of parallelism from different sources within the same program. We evaluate our approaches on a Silicon Graphics Origin2000 with synthetic benchmarks. The results show that our memory management technique reduces the runtime system overheads by 41% on average when local memory pools are used, and by 52% on average when a hierarchy of memory pools is used. The proposed user-level scheduling mechanism gives performance gains between 17% and 41%, compared to mechanisms that use local ready queues. The overall results of this work show that memory locality can be exploited to improve not only the performance of a parallel computation, but also the performance of the runtime mechanisms that execute this computation.

The rest of this paper is organized as follows: Section 2 gives a brief overview of NthLib. Section 3 presents our thread management alternatives. Section 4 describes our evaluation methodology and Section 5 presents experimental results. Section 6 summarizes our conclusions and presents future work.

2 NthLib: The Nano-Threads Library

NthLib [Mart95, Mart96] provides a runtime execution environment for applications written in standard C or FORTRAN. The runtime environment assumes that applications are automatically parallelized by a compiler, which produces an intermediate representation called the Hierarchical Task Graph (HTG)[Girk92]. The HTG representation has the ability to capture different levels of both structured (loop-level) and unstructured (task-level) parallelism. The runtime system uses the HTG representation to apply an *autoscheduling* execution mechanism [More95]. This mechanism gener-

ates *drive code*, which creates and schedules nano-threads that execute the parallel tasks represented by the HTG. The runtime system operates in close coordination with the operating system, in order to dynamically control the granularity of the generated nano-threads. Granularity control is based on information like the number of processors available to the application, estimations of the runtime system overheads, and the critical task size.

NthLib implements nano-threads with private contexts, which are kept in nano-thread stacks. The library uses the mechanisms provided by QuickThreads [Kepp93], a portable tool for building multithreading packages. An important aspect of NthLib is that it uses the C runtime stack to execute nano-threads. This means that nano-threads use standard mechanisms to access local and global data. These mechanisms can be highly optimized by existing C compilers. Moreover, they avoid the overhead of explicit memory allocation of activation frames from the heap and the maintenance of a cactus stack [More95]. Therefore, they simplify memory management at user-level. NthLib allocates memory from the heap only for nano-thread stacks. Stacks are recycled in a memory pool, in order to avoid the overhead of invoking the operating system memory allocator each time a new nano-thread is created. Context switching and blocking of nano-threads is implemented with the QuickThreads primitives.

NthLib uses local (per-processor[1]) ready queues and a global ready queue, where nano-threads can be submitted for execution. The queues provide a framework for implementing user-level scheduling policies in the context of NthLib. Local queues can be employed mainly to schedule parallel loops for maintaining locality. The global queue can be used for coarse-grain tasks and load balancing. The processors execute a scheduling loop throughout the lifetime of the application. In each iteration of the scheduling loop, a processor tries to dispatch a nano-thread for execution by visiting the ready queues in a predefined order. Dispatching of nano-threads is performed internally in NthLib. However, creation and enqueuing of nano-threads in ready queues is left entirely to the application. These actions can be performed either by the compiler, or directly by the programmer, who may provide scheduling hints to the runtime system. This approach increases flexibility, since user-level scheduling is often dependent on the application characteristics. Several scheduling algorithms for parallel loops implemented in Nthlib, can be found in [Mart97, Poly97].

3 Thread Management Alternatives

In this section we introduce two thread management techniques which we have implemented in NthLib. The first technique performs efficient memory management in the runtime system, by exploiting the affinity of nano-threads for processors. The proposed mechanism aims at reducing the overhead of initializing and managing nano-thread contexts at user-level. The second technique uses hierarchical ready queues for user-level scheduling in the runtime system. This technique provides a scheduling framework for applications with multilevel parallelism. The idea is to execute nested paral-

[1] Throughout this paper the term *processors* refers to kernel threads with a shared address space, that the operating system provides to parallel applications.

lelism that originates from different sources in the same program, in different clusters of the multiprocessor, in order to exploit memory locality to the extent possible. Our techniques can be effectively combined to handle efficiently fine-grain, multilevel parallelism in NUMA multiprocessors.

3.1 Memory Management

Before presenting our mechanism, we provide some background on memory management in NthLib and other multithreading runtime systems.

In order to implement the semantics of the autoscheduling model of execution, NthLib makes a clear distinction between nano-thread creation and nano-thread submission for execution (enqueing). A nano-thread is created as soon as the incoming control dependences of the corresponding HTG task are satisfied. Therefore, nano-threads are created when it is explicitly known that they are going to be executed. However, if the corresponding HTG task has pending incoming data dependences, the nano-thread cannot be inserted in a ready queue before these data dependences are resolved.

Creation of a nano-thread consists of allocating and initializing a nano-thread *descriptor*. The descriptor is a data structure with fixed size, that contains the nano-thread stack, the successors of the nano-thread in the HTG and the number of unresolved incoming data dependences. The function that the nano-thread is going to execute and the arguments of this function are passed to the nano-thread stack. Memory for the stacks is allocated from the heap and stacks are recycled in order to avoid the overheads of memory allocation by the operating system. Recycling is performed with a central pool of free stacks. The runtime system adds a stack to the pool when the corresponding nano-thread finishes execution. When a new nano-thread is created, the runtime system tries to obtain a stack from the pool, before attempting to allocate memory from the heap. Recycling of stacks is a common approach for reducing overheads and memory consumption in multithreading runtime systems.

General-purpose multithreading runtime systems do not necessarily use the above mechanism to create threads. One alternative approach is to create threads without private contexts. The applicability of this approach in the nano-threads library is limited, since the library must provide a way to maintain the scope of variables, as well as the context of blocked nano-threads. An additional drawback is that the number of nano-threads and hence the number of contexts that will be created during program execution remains unknown until runtime, due to the dynamic control of nano-threads granularity by the runtime library. Lazy stack allocation is another approach which decouples the initialization of the thread descriptor from the allocation of the thread stack. The stack allocation in this case is postponed until the thread is ready for execution. Due to efficiency reasons, we prefer to perform a single memory allocation of a fixed data structure for the nano-thread descriptor and the nano-thread stack, instead of two distinct allocations. In this way, we also avoid adding the overhead of initializing the stack in context-switch time. Stackless threads are used in the Filaments package [Free96]. The ELiTE runtime system [Bell96] uses lazy stack allocation. A thorough investigation of these techniques can be found in [Niko97].

Using a central memory pool for stack recycling does not always minimize the over-

heads of memory allocation in the nano-threads library. If, at a certain point during program execution, several processors create nano-threads simultaneously, accesses to the memory pool may cause high contention and result to a bottleneck. A simple solution is to use local pools of stacks as proposed in [Ande89]. Although this solution reduces contention, it should satisfy a set of criteria in order to be practically applicable. Local pools must be kept balanced in order to avoid situations where some processors always find free stacks in their local pools, while others invoke repeatedly the operating system memory allocator. Furthermore, local pools should enable the exploitation of data locality, particularly in NUMA multiprocessors where locality is a performance-critical factor. In this case, locality can be exploited when the nano-thread descriptor and the nano-thread stack are initialized. If the memory addresses accessed during initialization reside in the cache of the intializing processor, or as close to the processor as possible in the memory hierarchy, the initialization time will be significantly reduced.

In order to meet the above criteria we introduce a stack allocation scheme that uses local pools of stacks and a central pool. Pools are implemented as LIFO buffers with fixed size. Our experimentation with buffer sizes showed that buffers of 4 to 8 entries are adequate for stack recycling. The buffers operate in the following way: Whenever a processor finishes the execution of a nano-thread, it puts the nano-thread stack in the front of its local LIFO. When the same processor creates a new nano-thread, it tries to obtain a stack from the front of its local LIFO. In this way, we increase the probability that the processor will access recently touched addresses, that may still reside in the cache. The mechanism establishes a form of affinity of newly created nano-threads for processors. Nano-threads prefer to obtain their contexts from other nano-threads that ran recently on the same processor. This form of affinity can be exploited to reduce the overheads of managing the nano-thread contexts. In order to keep the local pools balanced we use the central pool, where processors can find stacks if their local pools are empty. Stacks are inserted in the central pool by a processor whose local pool is full, in order to be used by other processors.

The previously described technique is very simple and the results in Sect. 5 indicate that it has significant gains compared to the scheme proposed in [Mart96]. An interesting alternative, is to use the same mechanism with a hierarchy of pools. Intuitively, this hierarchy may correspond to the actual memory hierarchy of a NUMA system. We evaluate this approach in conjuction with the use of hierarchical ready queues, which is discussed in the next subsection.

3.2 Hierarchical Ready Queues

Dandamundi and Cheng [Dand95] have recently evaluated through simulations the efficiency of using hierarchical ready queues for scheduling on shared memory multiprocessors. In the proposed scheme, ready queues are organized as a tree which has a central ready queue at the root and local ready queues at the leaves. Each processor may access the queues that reside along the path that starts from the root and ends at the processor local queue. However, a processor may dispatch threads for execution only from the local queue. When accessing a ready queue at a higher level, the processor simply transfers a fraction of the number of threads in the ready queue one level down

in the tree. Ready threads are always enqueued at the root of the hierarchy. Hierarchical queues combine the load balancing properties of the central ready queue, with the data locality properties of local ready queues. A drawback of this scheme is that the average number of accesses needed to dispatch a thread for execution, is always higher than that needed when using a central ready queue, local ready queues, or a combination of both.

In this work we consider hierarchical queues as an effective alternative for scheduling nano-threads on NUMA multiprocessors. The idea is to map the hierarchy of ready queues to the memory hierarchy of the multiprocessor and try to exploit locality. We present a concrete example using the Silicon Graphics Origin2000 [Laud97] as a reference NUMA architecture.

The block diagram of a 32-processor Origin2000 system is outlined in Fig. 1. The basic building block of Origin2000 is a dual-processor node, which contains up to 4 gigabytes of main memory, the directory and the I/O subsystem. These components are connected to a hub and hubs in turn, are connected to six-ported routers that form a high-speed interconnection network with a hypercube topology. A straightforward mapping of hierarchical ready queues to the Origin2000 architecture, is to assign ready queues to individual processors, nodes, routers and finally the whole system. This leads to a complete hierarchy shown in Fig. 2(a) for a 16-processor system. Alternatively, in order to reduce the average number of queue accesses needed for dispatching a thread, we can consider only local queues, router queues and a central system queue, and build the three-level hierarchy shown in Fig. 2(b). Hierarchies for other NUMA architectures can be easily built in a similar manner.

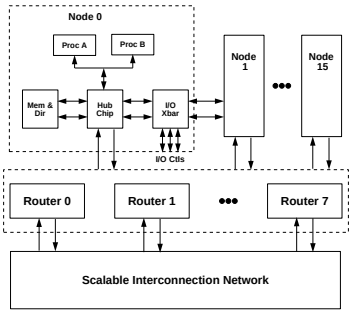


Fig. 1. Origin2000 block diagram

We modify the scheme proposed in [Dand95], to adapt it to the nano-threads programming model. Our goal is to use the hierarchical queues to schedule nano-threads, given the following facts:

- The runtime system may decide to exploit nested parallelism.
- The runtime system may decide to exploit parallelism that may originate from different sources within the same program.

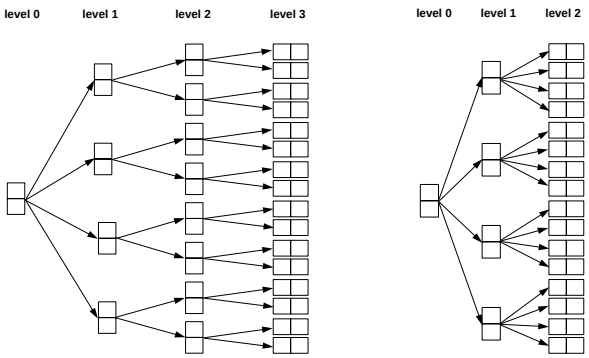


Fig. 2. Two hierarchical queue organizations for a 16-processor Origin2000 system

– Parallel tasks that originate from different sources access different data sets and may have different synchronization patterns.

We allow processors to enqueue threads arbitrarily in any queue that resides in the path from the root to their local queue. Furthermore, we let processors dispatch and execute threads from any queue along the same path, starting from their local queue and moving up the hierarchy to the root of the tree, in each iteration of their scheduling loop.

The intuition behind this approach is that nested parallel nano-threads should be executed in the same cluster of the machine, in order to preserve data locality. At the same time, parallelism that originates from different sources can be executed on different clusters, in order to avoid the undesirable interferences between nano-threads that access different data sets. A cluster can be arbitrarily defined as a *memory locality domain* of the underlying machine architecture. Each level of the queues hierarchy represents a partitioning of the machine in memory locality domains. If level l of the hierarchy contains n ready queues $Q_{li}, i = 1 \ldots n$, then this level represents a partitioning of the machine into n domains. Each domain i contains the processors whose local queues are at the leaves of the subtree emanating from Q_{li}. Our mechanism lets the application map the generated parallelism to memory locality domains. The number of domains used depends on the application characteristics and determines the level of the hierarchy selected to schedule the computation. Furthermore, if nano-threads scheduled at a given level of the hierarchy create nested nano-threads, the innermost nano-threads are scheduled in the same domains with their outermost counterparts.

As an example, consider an instance of a program where two processors generate threads to execute two distinct nested parallel loops, that reference diffrent data. A scheduling scheme that uses local queues would suggest to schedule both loops concurrently across all executing processors. Although this scheme maximizes the degree of parallelism for the execution of the loops, it may not actually be the best alternative. Nano-threads from the first loop pollute the cache footprints of nano-threads from the

second loop and vice versa. This interference may lead to poor cache performance, even if each individual loop exhibits good data locality. Following the hierarchical scheme, we can remedy this problem by assigning the loops to two distinct clusters of the machine. This approach maximizes data locality and avoids cache interferences, at the expense of reducing the degree of parallelism for each loop. In addition, the hierarchical scheme lets iterations from the innermost loops execute in the same clusters with their outermost counterparts. This strategy allows the computation to establish processor affinity and exploit cache reuse.

There are two more observations concerning hierarchical queues. The first observation, is that the scheme that we propose does not lack flexibility compared to other schemes that use a central queue and/or local queues. Generation and enqueuing of nano-threads are left entirely to the application or the compiler. Both are able to schedule the computation according to application-specific characteristics. The second observation, is that the proposed mechanism needs some additional support from the operating system, when used in a multiprogramming environment. The basic requirement is that the processor allocation policy of the operating system should establish physical partitions of the machine that can be mapped to hierarchies. Algorithms for partitioning can be found in [Feit97].

4 Evaluation Methodology

We evaluate our approaches with synthetic benchmarks, which we coded in C using the NthLib interface. We conducted our experiments on a 32-processor Silicon Graphics Origin2000. Our performance evaluation is based on the comparison of the following thread management alternatives.

- *Base*: This scheme uses a global ready queue and local ready queues for scheduling, and a central memory pool for stack allocation and recycling. This is the scheme currently implemented in NthLib and we use it as the base for our comparisons.
- *Stack_aff*: This scheme uses a global ready queue and local ready queues for scheduling, and the mechanism presented in Sect. 3.1 for allocation and recycling of nano-thread contexts. The size of the local LIFOs is experimentally set to 8.
- *Hier*: This scheme uses the hierarchy of ready queues shown in Fig. 2(b). We incorporate the affinity mechanism for stack initialization presented in Sect. 3.1, by using a hierarchy of LIFOs with a one-to-one mapping between LIFOs and ready queues. The affinity mechanism used in this case is identical to the mechanism used in *Stack_aff*, with the exception that processors scan one more level of LIFOs for free stacks.

We use two synthetic benchmarks to evaluate our approaches. Our primary goal is to evaluate the efficiency of the proposed schemes, with respect to the following criteria:

- Exploitation of parallelism at the finest level of granularity at any instance during program execution.
- Exploitation of multilevel and nested parallelism, that may originate from different sources within a program.

We use two benchmarks that capture different models of parallelism, which can be met in a wide range of applications.

The first benchmark follows the fork/join model. The benchmark creates 1 million empty nano-threads. Nano-threads are created in parallel by all processors participating in the execution. Each processor creates an equal number of nano-threads in bursts. After creating a burst, a processor blocks and waits for the execution of the burst to terminate. In the *Base* and *Stack_aff* alternatives, the number of nano-threads in each burst is set to be equal to the number of processors, in order to model a computation with frequent synchronization between processors. In the *Hier* alternative, the machine is divided in clusters and each burst is executed in a single cluster. The burst size used in this case is equal to the number of processors in the cluster. The implementation of *Hier* preserves locality, at the expense of increasing the synchronization costs for thread management. In all schemes nano-threads within a burst are scheduled in an interleaved manner across the available processors.

This benchmark is appropriate for estimating the pure runtime overheads of the nano-thread management alternatives, since nano-threads perform no useful computation. We use this benchmark to evaluate the performance of the memory management mechanism presented in Sect. 3.1.

The second synthetic benchmark is represented by the task graph shown in Fig. 3. Each task $T_i, i = 1 \ldots 8$ corresponds to a parallel loop. Tasks $T_1 \ldots T_4$ correspond to scalar-by-vector products, computed for four different vectors of double precision numbers. Tasks $T_5 \ldots T_8$ correspond to dot products, computed with the outputs of tasks $T_1 \ldots T_4$. The arcs indicate precedence relationships between the tasks that result from inherent data dependences. The sizes of the vectors are varied from 2048 to 16384 elements in subsequent executions of the benchmark. In each execution, the whole task graph is traversed repeatedly.

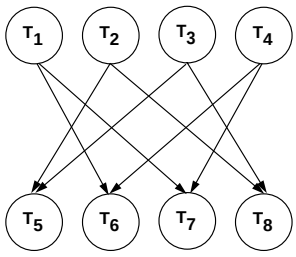


Fig. 3. Task graph for the second benchmark

This benchmark has some desirable properties. The benchmark allows the exploitation of two levels of parallelism. At the first level, tasks $T_1 \ldots T_4$ can be executed in parallel, followed by tasks $T_5 \ldots T_8$. At the second level, the loops inside each task can be parallelized with a loop translation scheme. Tasks T_1, T_4, T_6, and T_7 access the

same vectors, so scheduling these tasks and the generated loops locally in the machine is beneficial. The same holds for tasks $T_2, T_3, T_5,$ and T_8. Repeated execution of the task graph allows tasks to establish affinity for certain processors.

For the parallelization of loops we use the adaptive-size chunking algorithm proposed in [Mart97]. This algorithm executes loops with bursts of nano-threads. In the implementation for the *Base* and *Stack_aff* alternatives, loop parallelization is done using all the available processors. In the *Hier* version, parallelization is done by partitioning the machine and correspondingly the queues hierarchy in two symmetric clusters. Tasks $T_1, T_4, T_6,$ and T_7 are scheduled in the first cluster, while $T_2, T_3, T_5,$ and T_8 are scheduled in the second. Partitioning is performed to maximize locality of references, since tasks scheduled in the same cluster reference the same vectors. Tasks T_i are scheduled directly in router queues. The parallel loop generated from each task T_i is scheduled with adaptive-size chunking in the cluster to which the processor that executes T_i belongs.

The structure of this benchmark is representative for many scientific codes with multilevel and irregular parallelism. The SWIM and HYDRO2D codes from the SPECfp benchmark suite, as well as Computational Fluid Dynamics (CFD) codes are indicative examples.

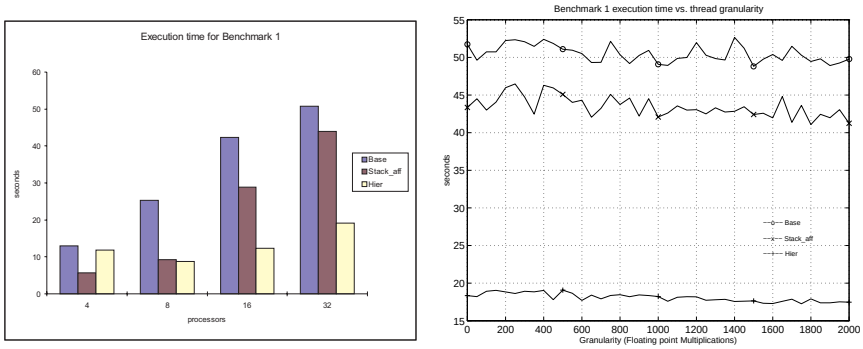
5 Experimental Results

In this section, we present detailed experimental results, derived from experiments with the thread management alternatives presented in Sect. 3.

Figure 4(a) plots the execution time for the first benchmark, using 4, 8, 16 and 32 processors. Each group of bars corresponds to the three alternatives *Base, Stack_aff* and *Hier*. Execution time in this benchmark is dominated by runtime system overheads and the cost of synchronization between processors. Therefore, it is not expected to give significant speedups, if any. Comparing *Base* with *Stack_aff* we see that the use of our memory management mechanism gives performance gains between 13% and 63% with an average gain of 41%. *Base* performs worse because of the frequent synchronization between processors for the access of the central memory pool. *Hier* performs better than both *Base* and *Stack_aff*, with 8 or more processors. Note that when 4 processors are used, *Hier* performs no partitioning, therefore it is not able to obtain significant benefits from locality. Compared to *Stack_aff*, *Hier* reduces execution time by 5%-67%. The average gain of *Hier* compared to *Base* is 52%. This indicates that incorporating our memory management mechanism in a hierarchical scheme is highly effective. The benefits of *Hier* over *Stack_aff* are justified by the fact that *Hier* clusters the management of parallelism (including initialization, scheduling, synchronization and recycling of nano-thread contexts) in memory locality domains. This allows better exploitation of locality.

In order to evaluate the ability of the proposed mechanisms to handle efficiently parallelism of very fine granularity, we ran the first benchmark on 32 processors and let the nano-threads execute a small amount of work ranging from 0 to 2000 floating point multiplications. The execution time is plotted in Fig. 4(b). We observe that despite the increase of granularity, which implies an increase of the ratio of computation

to overheads, the performance of the three approaches is practically unaffected. *Hier* remains the best alternative for fine nano-thread granularities and the performance of the alternatives does not converge as granularity remains at low levels.



(a) (b)

Fig. 4. Execution time for the first benchmark with zero and varying thread granularity

Figures 5(a) through (d) plot the execution time of the second benchmark for different vector sizes ranging from 2048 to 16384 elements. Each chart plots execution time versus the number of processors for *Base*, *Stack_aff*, and *Hier*, for executions on up to 16 processors. The results verify our intuition that hierarchical queues exploit better nested parallelism. *Hier* outperforms clearly the other approaches. The gains of *Hier* compared to *Base* on 16 processors, range between 17% and 41%. *Stack_aff* has only marginal gains over *Base*. These gains can be attributed to the use of our memory management mechanism.

The main advantage of *Hier*, is that it performs efficient partitioning and mapping of the computation to the machine architecture, in order to maximize locality and avoid interferences between nano-threads with different data access traces. Both *Base* and *Stack_aff* have poorer cache performance than *Hier*, despite the fact that they use all the available processors to execute the parallel loops. Note that the benchmark suffers from performance degradation when executed on more than 8 processors. Futhermore, the benchmark exhibits reasonable speedups with all three alternatives, only when vector sizes of 16384 are used. The exception is *Hier*, that achieves speedup with a vector size of 8192 too. This problem is progressively alleviated when the size of the vectors and the nano-threads granularity are adequately increased. However, we experimented with small vector sizes, in order to measure the ability of our mechanisms to handle effectively fine-grain parallelism.

Preliminary results from experiments with application benchmarks (not shown here) verify the validiy of our approaches. *Hier* gives average performance gains of 48% and 17% compared to *Base*, with the kernel of a Fourier-Chebyshev spectral computational

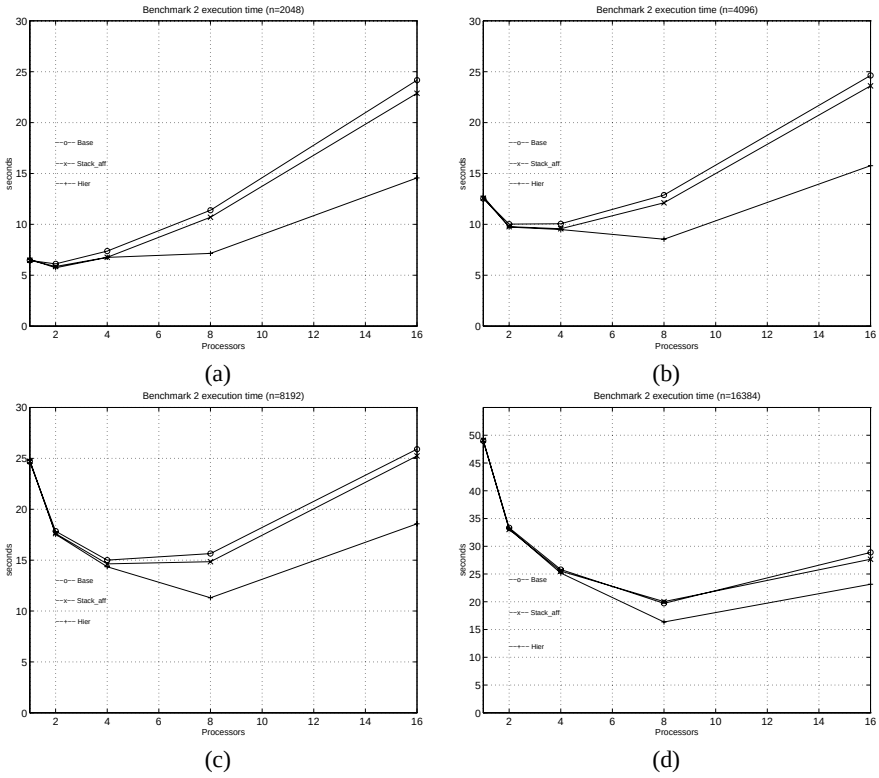


Fig. 5. Execution times for the second benchmark with different problem sizes.

fluid dynamics code and a complex matrix multiplication kernel respectively.

Summarizing, we conclude that the use of hierarchical schemes for scheduling and memory management in the runtime system gives substantial performance gains on NUMA multiprocessors. This is a direct consequence of the way that hierarchical schemes decompose the management and execution of parallelism in memory locality domains.

6 Conclusions and Future Work

In this paper, we introduced thread management alternatives, which we applied in a runtime system designed to support the nano-threads programming model. We proposed a mechanism that exploits affinity of threads for processors, in order to reduce the overhead of managing nano-thread contexts in the runtime system. We also proposed a scheduling framework based on hierarchical ready queues, which allows effective partitioning of parallel computations by taking into account the platform architecture. Our results show that the use of hierarchical schemes in NUMA machines are highly effec-

tive, because of their ability to take full advantage of data locality. We recommend the use of these schemes in the nano-threads programming model for two reasons: First, they are able to manage parallelism of very fine granularity with minimal overheads. Second, they are appropriate for scheduling unstructured and nested parallelism, which may be extracted from different sources within an application.

We currently investigate the effectiveness of further optimizations in the nano-threads library, including the use of parallel-access queues and software prefetching. We also investigate concrete user-level partitioning algorithms with hierarchical queues, which we evaluate with scientific application codes.

Acknowledgements

We would like to thank Constantine Polychronopoulos for his support of this work, Xavier Martorell for his help in conducting the experiments, the European Center for Parallelism in Barcelona (CEPBA) for providing us access to their Origin2000 system and the referees for their helpful comments.

References

[Ande89] T. Anderson, E. Lazowska and H. Levy, *The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors*, IEEE Transactions on Computers, vol. 38(12), pp. 1632–1644, December 1989.

[Bell96] F. Bellosa and M. Steckermeier, *The Performance Implications of Locality Information Usage in Shared-Memory Multiprocessors*, Journal of Parallel and Distributed Computing, vol. 37(1), pp. 113–121, August 1996.

[Dand95] S. Dandamundi and P. Cheng, *A Hierarchical Task Queue Organization for Shared-Memory Multiprocessor Systems*, IEEE Transactions on Parallel and Distributed Systems, vol. 6(1), pp. 1–16, January 1995.

[Feit97] D. Feitelson, *Job Scheduling in Multiprogrammed Parallel Systems*, IBM Research Report 19790, Second Revision, August 1997.

[Free96] V. Freeh, D. Lowenthal, and G. Andrews, *Efficient Support for Fine-Grain Parallelism on Shared-Memory Machines*, Technical Report TR96-1, Univeristy of Arizona, January 1996.

[Girk92] M. Girkar and C. Polychronopoulos, *Automatic Extraction of Functional Parallelism from Ordinary Programs*, IEEE Transactions on Parallel and Distributed Systems, vol. 3(2), pp. 166–178, March 1992.

[Kepp93] D. Keppel, *Tools and Techniques for Building Fast Portable Threads Packages*, Technical Report UWCSE 93-05-06, University of Washington at Seattle, May 1993.

[Laud97] J. Laudon and D. Lenoski, *The SGI Origin: A ccNUMA Highly Scalable Server*, Proceedings of the 24th International Symposium on Computer Architecture, pp. 241–251, Denver, Colorado, June 1997.

[Mart95] X. Martorell, J. Labarta, N. Navarro and E. Ayguadé, *Nano-Threads Library Design, Implementation and Evaluation*, Technical Report UPC-DAC-1995-33, Universitat Politècnica de Catalunya, November 1995.

[Mart96] X. Martorell, J. Labarta, N. Navarro and E. Ayguadé, *A Library Implementation of the Nano-Threads Programming Model*, Proceedings of the 2nd International Euro-Par Conference, pp. 644–649, Lyon, France, August 1996.

[Mart97] X. Martorell, J. Labarta, N. Navarro and E. Ayguadé, *Analysis of Several Scheduling Algorithms under the Nano-threads Programming Model*, Proceedings of the 11th International Parallel Processing Symposium, pp. 281–287, Geneva, Switzerland, April 1997.

[More95] J. Moreira, *On the Implementation and Effectiveness of Autoscheduling for Shared-Memory Multiprocessors*, PhD Thesis, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering, 1995.

[Niko97] D. Nikolopoulos, E. Polychronopoulos, I. Tsolakis and T. Papatheodorou, *A Comparative Study of Multithreading Runtime Systems for Parallel Programming*, Technical Report HPCAL-TR-010797, University of Patras, Department of Computer Engineering and Informatics, July 1997.

[Poly93] C. Polychronopoulos, N. Bitar and S. Kleiman, *Nano-Threads: A User-Level Threads Architecture*, Technical Report 1297, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1993.

[Poly97] E. Polychronopoulos and T. Papatheodorou, *Dynamic Bisectioning Scheduling for Scalable Shared-Memory Multiprocessors based on the Nano-Threads Programming Model*, Technical Report HPCAL-TR-010697, University of Patras, Department of Computer Engineering and Informatics, June 1997.

This article was processed using the LaTeX macro package with LLNCS style