

# Enhancing the Performance of Autoscheduling with Locality-Based Partitioning in Distributed Shared Memory Multiprocessors<sup>\*</sup>

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  
{dsn,edp,tsp}@hpclab.ceid.upatras.gr

**Abstract.** Autoscheduling is a parallel program compilation and execution model that combines uniquely three features: Automatic extraction of loop and functional parallelism at any level of granularity, dynamic scheduling of parallel tasks, and dynamic program adaptability on multiprogrammed shared memory multiprocessors. This paper presents a technique that enhances the performance of autoscheduling in Distributed Shared Memory (DSM) multiprocessors, targeting mainly at medium and large scale systems, where poor data locality and excessive communication impose performance bottlenecks. Our technique partitions the application Hierarchical Task Graph and maps the derived partitions to clusters of processors in the DSM architecture. Autoscheduling is then applied separately for each partition to enhance data locality and reduce communication costs. Our experimental results show that partitioning achieves remarkable performance improvements compared to a standard autoscheduling environment and a commercial parallelizing compiler.

## 1 Introduction

Distributed Shared Memory (DSM) multiprocessors have evolved as a powerful and viable platform for parallel computing. Unfortunately, automatic parallelization for these systems becomes often an onerous duty, requiring numerous manual optimizations and substantial effort from the programmer to sustain high performance [3]. The development of programming models and compilation techniques that exploit the advantages and overcome the architectural bottlenecks of DSM systems remains a challenge.

Among several parallel compilation and execution models proposed in the literature, *autoscheduling* [13] is one of the most promising approaches. Autoscheduling provides a compilation framework which merges the control flow and data flow models to efficiently exploit multiple levels of loop and functional parallelism. The compilation framework is coupled with an execution environment

---

<sup>\*</sup> This work was supported by the European Commission under ESPRIT IV project No. 21907 (NANOS)

that schedules parallel programs dynamically on a variable number of processors, by controlling the granularity of parallel tasks at runtime. Coordination between the compiler and the execution environment achieves effective integration of multiprocessing with multiprogramming.

In this paper, we identify sources of performance degradation for an autoscheduling environment running on a DSM multiprocessor. Performance bottlenecks arise as a consequence of poor caching performance and excessive communication through the interconnection network. We present a technique that attempts to resolve these problems by augmenting autoscheduling with partitioning and clustering. We partition the application task graph in subgraphs with independent dataflows and map the derived subgraphs to processor clusters. The topology of the clusters conforms to the system architecture. Partitioning and clustering are performed dynamically at runtime. After mapping partitions to clusters, autoscheduling is applied separately to each partition to extract and schedule the parallelism.

Using application and synthetic benchmarks, we demonstrate that partitioning reduces significantly the execution time of autoscheduled programs on a 32-processor SGI Origin2000. For the same set of benchmarks, partitioned autoscheduling outperforms the native SGI loop parallelizer, even in situations where standard autoscheduling fails to do so.

The rest of this paper is organized as follows: Section 2 provides some background on autoscheduling and discuss its performance limitations. Section 3 presents our technique along with some implementation issues. Section 4 describes our evaluation framework and Section 5 provides detailed experimental results. We summarize our conclusions in Section 6.

## 2 Autoscheduling and Performance Limitations

An autoscheduling environment consists of a parallelizing compiler, a multithreading runtime library and an enhanced operating system scheduler. The compiler applies control and data dependence analysis to produce an intermediate program representation called the Hierarchical Task Graph (HTG) [5]. Programs are represented with a hierarchical structure consisting of simple and compound nodes. Simple nodes correspond to basic blocks of sequential code. Compound nodes represent tasks amenable to parallelization, such as parallel loops and parallel sections. The levels of the hierarchy are determined by the nesting of loops and correspond to different levels of exploitable parallelism. Nodes are connected with arcs representing execution precedences. A boolean expression with a set of firing conditions is associated with each node. When this expression evaluates to TRUE, all data and control dependences of the node are satisfied and the corresponding task is enabled for execution.

The compiler generates parallel code by means of a user-level runtime system which creates and schedules lightweight execution flows called *nano-threads* [13]. The granularity of the generated parallelism is controlled dynamically at runtime. A communication mechanism between the runtime system and the operat-

ing system scheduler keeps parallel programs informed of processor availability and enables dynamic program adaptability to varying execution conditions [14].

Each compound node in the HTG is annotated as a candidate source of parallelism. The runtime system decomposes all the compound nodes which contain enough parallelism to utilize the processors available at a given execution instance. If several compound nodes coexist at the same level of the hierarchy, parallelism can be extracted from all of them simultaneously. In that case, nano-threads originating from different compound nodes are executed in an overlapped manner across the processors. As an example, consider the HTG of the multiplication of two matrices  $Y, Z$  with complex entries, calculated as:

$$X = YZ = (A + jB)(C + jD) = (AC - BD) + j(BC + AD) \quad (1)$$

The computation consists of four independent matrix by matrix multiplications, followed by an addition and a subtraction. Each of these tasks is a compound node with inherent loop parallelism. If a sufficient number of processors is available at runtime, autoscheduling may execute the matrix multiplications simultaneously and each matrix multiplication in parallel to exploit two levels of parallelism.

Overlapped execution of nano-threads originating from different compound nodes can have undesirable effects on a DSM multiprocessor. This is particularly true when each compound node individually uses all the available processors. Three performance implications may arise in this case: Increase of conflict and capacity cache misses, heavy traffic in the interconnection network, and higher runtime overhead.

Bursts of conflict and capacity misses occur when the working sets of concurrently executing compound nodes do not fit in the caches, thus forcing the caches to alternate frequently between different working sets. This happens as processors are switched between nanothreads originating from different compound nodes. Overlapped execution of compound nodes stresses the interconnection network, due to excessive data communication. The runtime system overhead is higher, since multiple processors are forced to access heavily shared ready queues and memory pools, in order to create and schedule nano-threads. This side-effect exacerbates contention and communication traffic due to remote memory accesses [11].

### 3 Performance Enhancements

Our method to alleviate the problems presented in Sect. 2 while preserving the semantics of autoscheduling, is to decompose the program HTG into partitions with independent data flows and the processor space of the program into clusters, with a one-to-one mapping between HTG partitions and processor clusters. After partitioning and clustering, autoscheduling is applied separately for each partition. Parallelism from each partition is extracted to utilize the processors of the cluster to which the partition is mapped. Processor clusters serve as memory

locality domains for the partitions. This means that the working set of a partition is stored in physical memory that lies within the corresponding cluster, in order to reduce memory latency during the execution of the partition. The communication required for a partition is performed within the cluster, thus putting less pressure in the interconnection network and avoiding long distance memory accesses. Furthermore, data dependent partitions use the same processors to preserve locality. If a partition  $\mathcal{P}$  is data dependent on a partition  $\mathcal{Q}$ , our mechanism executes  $\mathcal{P}$  either in a superset, or a subset of the cluster to which  $\mathcal{P}$  is mapped.

The relative size of the clusters is determined with the following procedure: Compound nodes that can be simultaneously executed are detected in the HTG and grouped in sets. The computational requirements of each compound node are estimated with execution profiling and then weighed with the aggregate computational requirements of the compound nodes that belong to the same set. Processors are distributed proportionally to each compound node according to its weighed computational requirements. The actual size of the clusters is set at runtime, and depends on their precomputed relative size and processor availability. Let  $P$  be the number of processors available to the application at an execution instance,  $n$  the number of concurrently executing compound nodes at that instance and  $r_i, i = 1 \dots n$ , the computational requirement of each compound node, expressed in CPU time. Then the processor set is partitioned into  $n$  clusters  $c_i, i = 1 \dots n$ , and each cluster receives  $p_{c_i}$  processors where:

$$p_{c_i} = \max\left\{1, \frac{r_i}{\sum_{i=1}^n r_i} P\right\} \quad (2)$$

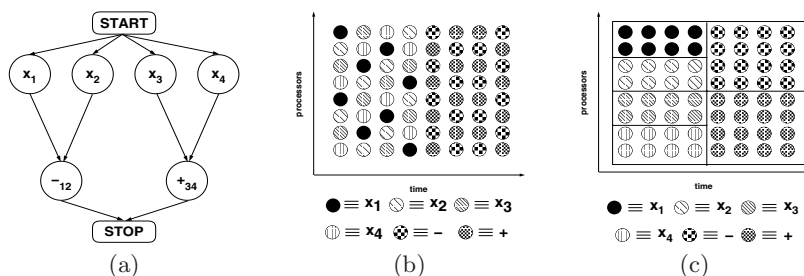
If the number of available processors does not exceed the degree of functional parallelism exposed by the application —expressed as the number of concurrently executing compound nodes—, then our technique degenerates to standard auto-scheduling. Otherwise, autoscheduling is applied within the bounds of each cluster by extracting the data and/or functional parallelism of the associated HTG partition. The clusters can be dynamically reconfigured at runtime. Load imbalances between clusters are not considered. We rather rely on the proportional allocation of processors to clusters to handle these cases. However, load imbalances within the clusters across parallel loops or parallel sections are handled by the user-level scheduler which employs affinity scheduling and work stealing [15].

The topology of the clusters is kept consistent with the underlying system architecture. For DSM architectures, clusters are built incrementally using the Symmetric Multiprocessing (SMP) nodes as the basic building block. The interconnection network is exploited to form clusters with SMP nodes which are physically close to each other. The clusters can shrink or grow at runtime, either by splitting, or by joining directly connected clusters.

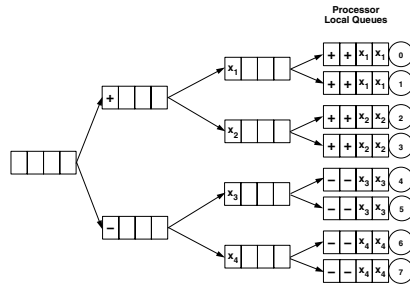
The entire mechanism is implemented at user-level. Partitions are detected from the HTG, and communicated to the runtime system through directives. Static information about the relative processor requirements of each partition is also communicated to the runtime system. The actual cluster sizes are computed

at runtime using Eq. 2. The infrastructure for mapping the partitions to processors is provided by a hierarchy of ready queues [2]. The hierarchy corresponds to a tree with a global system queue at the root, per-processor local queues at the leaves and a number of intermediate levels of ready queues. Mapping of partitions to clusters is performed with Distributed Hierarchical Control (DHC) [4]. When a compound node is assigned to a cluster of size  $p$ , the corresponding nano-thread is mapped to a ready queue at the root of a subtree that has at least  $p$  processors at the leaves. The processor that creates the nano-thread searches the hierarchy up until it finds a queue from which a tree of the appropriate size is emanated. Nested nano-threads created by the outermost nano-thread are mapped to the queues that reside at the leaves of the subtree. Processor clusters are implicitly configured using DHC, without invoking the operating system to construct the memory locality domains.

As an example, Fig. 1.(a) illustrates the task graph of complex matrix multiplication. Following our scheme, the processor set that executes the computation is initially partitioned in four clusters of equal size, and a matrix multiplication is executed separately in each cluster. Within a cluster, the matrix multiplication is autoscheduled on a number of processors equal to the size of the cluster. When the subtraction and addition tasks are activated, pairs of neighbouring clusters are joined to form two clusters of doubled size and execute the tasks. Figures 1.(b) and 1.(c) demonstrate executions of the task graph under standard autoscheduling and our scheme respectively. Circles indicate nano-threads, while their fill patterns indicate compound nodes from which the nano-threads originate. Under standard autoscheduling, the execution sequence of nano-threads from the concurrently executing compound nodes is determined from the serialization of processor accesses to the ready queues and may change between subsequent executions of the program. With partitioning, compound nodes are still executed concurrently, but the parallelism extracted from each partition is scheduled in isolation within a cluster. Figure 2 illustrates a hierarchy of ready queues for an 8-processor system and the scheduling strategy for the complex matrix multiplication example.



**Fig. 1.** Complex matrix multiplication task graph and execution with standard and partitioned autoscheduling.  $\cdot$ ,  $\times$ ,  $+$ ,  $-$  indicate matrix by matrix operations.



**Fig. 2.** Hierarchy of ready queues and enqueueing of complex matrix multiplication tasks.

### 4 Evaluation Framework

We evaluate our technique with a number of application and synthetic benchmarks. The application benchmarks presented here were used in a previous study of autoscheduling [9]. This gives us the opportunity for a quantitative analysis of our results in direct comparison with existing work in the field. The purpose of the synthetic benchmarks is to demonstrate the performance implications of autoscheduling and the ability of our method to overcome them.

The application benchmarks include the complex matrix multiplication kernel, the kernel of a Fourier-Chebyshev spectral computational fluid dynamics code [7], and an industrial molecular dynamics code [16]. All benchmarks have two levels of exploitable parallelism, namely a constant degree of functional parallelism at the outer level and data parallelism at the inner level. The complex matrix multiplication and the cfd kernels use matrices of size  $256 \times 256$ . The molecular dynamics kernel consists of 14 parallel functions, each one executing a parallel loop with 1000 iterations. More details on the structure of these benchmarks, as well as their performance under autoscheduling on a bus-based multiprocessor can be found in [9].

Our first synthetic benchmark consists of 8 parallel dot products. Each dot product is computed with two vectors of 64 Kilobytes of double precision elements. A detailed description of the benchmark can be found in [11]. Two levels of functional and data parallelism are exploitable, similarly to the application benchmarks. The dataflow of this benchmark favours processor clustering to maintain locality. The benchmark experiences increased conflict cache misses due to the alignment of data in memory, and a low ratio of computation to overhead because of the fine thread granularity. The second synthetic benchmark consists of four parallel point LU factorizations, followed by two parallel additions of the factorized matrices. The matrix size used is  $256 \times 256$ . Autoscheduling is expected to lead to conflicts between elements of different matrices in the caches, and increased remote cache invalidations due to the all to all communication pattern of the computation. Partitioning is expected to save cache misses and localize communication.

**Table 1.** Maximum speedups attained for the benchmarks with the three parallelization schemes.

Benchmark	Loop Parallel		Autoscheduled		Partitioned Autosch.	
	$S_{max}$	$p$	$S_{max}$	$p$	$S_{max}$	$p$
synthetic dot product	1.42	4	1.82	4	3.52	8
synthetic LU	7.06	16	7.34	16	15.72	32
cmm kernel	7.14	16	8.51	16	10.08	16
cfid kernel	5.59	12	6.22	12	7.31	12
md kernel	1.13	14	1.03	14	1.34	28

We performed our experiments on a SGI Origin2000 [6], equipped with 32 MIPS R1000 processors. We implemented partitioning and clustering using NthLib [8]. NthLib is a multithreading runtime library designed to support autoscheduling. The library is currently under development in the context of the ESPRIT project NANOS [10]. NthLib provides lightweight threads, created and scheduled entirely at user-level. The runtime system translates HTG representations into multithreading code, while attempting to preserve desirable properties such as data locality and load balancing. At the same time, NthLib communicates with the operating system to enable application adaptivity to the runtime conditions. Automatic parallelization was performed with HPF-like directives and the Parafrase-2 compiler [1,12]. Three versions of each benchmark were used in the experiments: (a) a *loop parallel* version produced by the native SGI parallelizing compiler and exploiting a single level of parallelism from loops; (b) a *autoscheduled* version, which exploits multiple levels of loop and functional parallelism; and (c) a *partitioned autoscheduled* version which uses autoscheduling enhanced with partitioning and clustering. All benchmarks were compiled with the `-O3` optimization flag and executed in a dedicated environment.

## 5 Performance Results

In this section, we present detailed results from our experiments. Table 1 shows the maximum speedups attained for the benchmarks presented in Sect. 4 with the three parallelization schemes. The table also shows the number of processors on which the maximum speedup is attained in each case.

The obtained speedups are justified as follows: For the synthetic dot product benchmark, poor speedup is a consequence of increased runtime overhead and cache conflicts. This benchmark scales better only if the problem size and thread granularity are significantly increased. The molecular dynamics kernel exhibits poor speedups due to poor data locality and high irregularity in the native code. Our results for this code agree with those reported in [9]. The other three codes exhibit good speedups, with respect to the structure of the computations, the platform architecture and the problem size.

The primary conclusion drawn from Table 1 is that in all cases, the maximum attainable speedup is obtained from the partitioned autoscheduled versions. The

partitioned autoscheduled versions of the synthetic benchmarks and the molecular dynamics kernel scale up to a higher number of processors than both the autoscheduled and the loop parallel versions. The autoscheduled versions have always better speedups than the loop parallel versions with the exception of the molecular dynamics kernel. These results are consistent with the results in [9] and strengthen the argument that autoscheduling exploits unstructured parallelism more effectively.

All benchmarks achieve better speedups and some of them do scale up to 32 processors if the problem sizes and thread granularities are adequately increased. However, we insisted in using small problem sizes to stress the ability of the runtime system to handle fine-grain parallelism, a prerequisite for the applicability of autoscheduling. We believe that parallel sections and loops with such a fine granularity are frequently met in scientific codes, and the question whether parallelizing these code fragments or not is worth the effort remains often dangling. Autoscheduling is a practical scheme for exploiting this kind of parallelism from scientific applications and our experiments intend to demonstrate this feature. A more thorough study of the codes as well as architecture-specific optimizations that could improve the speedups on our experimental platform are out of the scope of this paper.

Figure 3 illustrates the normalized execution times of the benchmarks when executed on a variable number of processors. Normalization is performed against the sequential execution time excluding the parallelization overhead. The conclusions concerning the relative performance of the three parallelization methods, agree with those derived from Table 1. The partitioned autoscheduled versions exhibit good behaviour even when executed on 32 processors, despite the fact that most benchmarks do not scale up to this point. Compared to loop parallelization and standard autoscheduling, our technique improves speedup by a factor of two and 36% respectively for the complex matrix multiplication. The corresponding numbers for the CFD kernel are 55% and 45%. For the synthetic dot product benchmark, partitioned autoscheduling is the only mechanism that achieves any speedup on 32 processors.

Two benchmarks that exhibit interesting behaviour are the synthetic LU benchmark and the molecular dynamics kernel. In the former, autoscheduling gives only marginal improvements compared to loop parallelization, while in the latter, autoscheduling results in performance degradation. In both cases, partitioning overcomes these limitations. The speedup of the synthetic LU benchmark is almost doubled with partitioning. The molecular dynamics kernel enjoys a noticeable speedup improvement of 19%.

In order to gain a better insight into our results, we performed experiments that measured memory performance under the three parallelization schemes. We used the hardware counters of MIPS R10000 and the `perfex` utility to obtain various performance statistics. Figure 4 presents a summary of the results from these experiments. The charts show the total number of cache misses and remote invalidations for each benchmark when executed on 32 processors. The results indicate clearly that partitioned autoscheduling improves memory performance

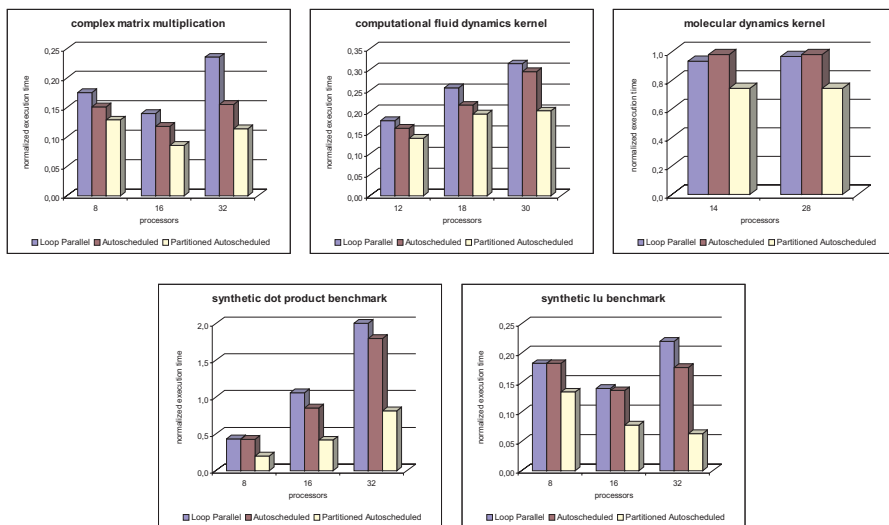


Fig. 3. Normalized execution times of the benchmarks

by reducing cache misses and interconnection network traffic. Cache misses and remote invalidations are on average halved with partitioning and clustering.

## 6 Conclusions

This paper demonstrated that although autoscheduling exploits potentially more parallelism, it is still amenable to enhancements in order to achieve high performance in DSM multiprocessors. We presented a simple and easy to implement technique which takes advantage of data and communication locality by applying partitioning and clustering of autoscheduled programs. The results proved that autoscheduling remains the exploiting choice for multilevel parallelism when the mechanisms that extract and schedule the parallelism are aware of the target architecture. Several issues of partitioned autoscheduling need further investigation. Topics for further research might be the automation of the clustering procedure using program analysis in the compiler, and the evaluation of partitioned autoscheduling under multiprogramming conditions. Our current work is oriented towards these directions.

## Acknowledgements

We would like to thank Constantine Polychronopoulos for his support and valuable comments, the European Center for Parallelism in Barcelona (CEPBA) for providing us access to their Origin2000, all our partners in the NANOS project and George Tsolis for his help in improving the appearance of the paper.

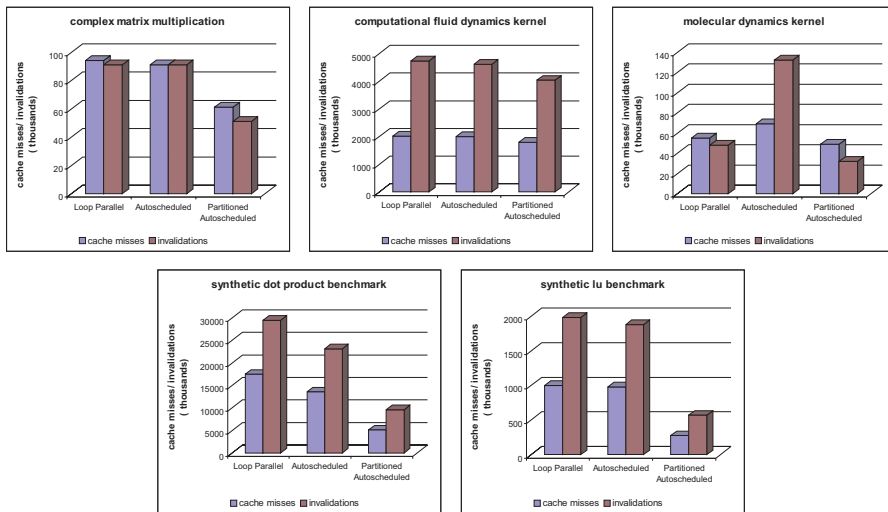


Fig. 4. Memory performance of the benchmarks

## References

1. E. Ayguadé, X. Martorell, J. Labarta, M. González, and N. Navarro, *Exploiting Parallelism through Directives on the Nano-Threads Programming Model*, Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing, Minneapolis, Minnesota, August 1997. 497
2. S. Dandamundi, *Reducing Run Queue Contention in Shared Memory Multiprocessors*, IEEE Computer, Vol. 30(3), pp. 82–89, March 1997. 495
3. R. Eigenmann, J. Hoeflinger and D. Padua, *On the Automatic Parallelization of Perfect Benchmarks*, IEEE Transactions on Parallel and Distributed Systems, Vol. 9(1), pp. 5–23, January 1998. 491
4. D. Feitelson and L. Rudolph, *Distributed Hierarchical Control for Parallel Processing*, IEEE Computer, Vol. 23(5), pp. 65–79, May 1990. 495
5. 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. 492
6. J. Laudon and D. Lenoski, *The SGI Origin: A ccNUMA Highly Scalable Server*, Proceedings of the 24th Annual International Symposium on Computer Architecture, pp. 241–251, Denver, Colorado, June 1997. 497
7. S. Lyons, T. Hanratty and J. MacLaughlin, *Large-scale Computer Simulation of Fully Developed Channel Flow with Heat Transfer*, International Journal of Numerical Methods for Fluids, vol. 13, pp. 999–1028, 1991. 496
8. X. Martorell, J. Labarta, N. Navarro and E. Ayguadé, *A Library Implementation of the Nano-Threads Programming Model*, Proceedings of Euro-Par’96, pp. 644–649, Lyon, France, August 1996. 497
9. 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. 496, 497, 498

10. NANOS Consortium, *Nano-Threads Programming Model Specification*, ESPRIT Project No. 21907 (NANOS), Deliverable M1.D1, July 1997, also available at <http://www.ac.upc.es/NANOS> 497
11. D. Nikolopoulos, E. Polychronopoulos and T. Papatheodorou, *Efficient Runtime Thread Management for the Nano-Threads Programming Model*, Proceedings of the IPPS/SPDP'98 Workshop on Runtime Systems for Parallel Programming, LNCS vol. 1388, pp. 183–194, Orlando, Florida, March/April 1998. 493, 496
12. C. Polychronopoulos, M. Girkar, M. Haghghat, C. Lee, B. Leung and D. Schouten, *Parafraze-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs*, International Journal of High Speed Computing, Vol. 1 (1), 1989. 497
13. C. Polychronopoulos, N. Bitar and S. Kleiman, *Nano-Threads: A User-Level Threads Architecture*, CSRD Technical Report 1297, University of Illinois at Urbana-Champaign, 1993. 491, 492
14. E. Polychronopoulos, X. Martorell, D. Nikolopoulos, J. Labarta, T. Papatheodorou and N. Navarro, *Kernel-Level Scheduling for the Nano-Threads Programming Model*, Proceedings of the 12th ACM International Conference on Supercomputing, Melbourne, Australia, July 1998. 493
15. E. Polychronopoulos and T. Papatheodorou, *Dynamic Bisectioning Scheduling for Scalable Shared-Memory Multiprocessors*, Technical Report LHPCA-010697, University of Patras, June 1997. 494
16. B. Quentrec and C. Brot, *New Method for Searching for Neighbors in Molecular Dynamics Computations*, Journal of Computational Physics, Vol. 13, pp. 430–432, 1975. 496