

Runtime Adjustment of Parallel Nested Loops

Alejandro Duran¹, Raúl Silvera², Julita Corbalán¹, and Jesús Labarta¹

¹ CEPBA-IBM Research Institute,
Departament d'Arquitectura de Computadors,
Universitat Politècnica de Catalunya,
Jordi Girona, 1-3, Barcelona, Spain
{aduran, juli, jesus}@ac.upc.es

² IBM Toronto Lab, 8200 Warden Ave
Markham, ON, L6G 1C7, Canada
rauls@ca.ibm.com

Abstract. OpenMP allows programmers to specify nested parallelism in parallel applications. In the case of scientific applications, parallel loops are the most important source of parallelism. In this paper we present an automatic mechanism to dynamically detect the best way to exploit the parallelism when having nested parallel loops. This mechanism is based on the number of threads, the problem size, and the number of iterations on the loop. To do that, we claim that programmers must specify the potential application parallelism and give the runtime the responsibility to decide the best way to exploit it. We have implemented this mechanism inside the IBM XL runtime library. Evaluation shows that our mechanism dynamically adapts the parallelism generated to the application and runtime parameters, reaching the same speedup as the best static parallelization (with a priori information).

1 Introduction

OpenMP[1] is the standard programming model in shared-memory multiprocessor systems. This programming model provides programmers with a set of directives to explicitly define parallel regions in applications. These directives are translated by the compiler that generates code for the runtime parallel library.

The runtime parallel library is in charge of managing the application parallelism. It can try to optimize decisions that can only be resolved at runtime such as the number of processors used to execute a parallel region or the scheduling of a parallel loop.

In the case of numerical applications, loops are the most important source of parallelism. The typical structure of these parallel loops is several nested loops that cover several matrix dimensions (i,j,k,...). Often, these inner loops are also parallel.

A typical practice of parallel programmers is to insert the parallel directive in only one loop, usually the outer loop, serializing the rest of the nested loops. This approach is sufficient if the number of iterations of the outer loop is high and the iteration's granularity is coarse enough. However, in some cases the number of iterations of the outer parallel loop is not enough to distribute between the available number of processors, even though the total computational work is very high.

With this behavior, programmers decide a priori the loop parallelization, ignoring runtime characteristics such as the number of threads available at the moment of opening parallelism.

In this paper we claim that programmers must use OpenMP directives just to specify parallelism, annotating all application parallelism, and give the runtime library the responsibility of select the best way to exploit the available parallelism. This would be done as a function of the application characteristics and resource availability. Instead of exploiting either the outer or the inner loop, we propose to apply a mixed approach, opening parallelism in several iterations of the outer loop and exploiting the parallelism of the inner loop in the rest of the iterations.

The runtime will gather information of the parallel loops and will make a decision of whether to open parallelism at the outer loop, do it at the inner one or do some iterations of the outer loop in parallel and some iterations of the inner loop in parallel. This last scheme causes the overheads to be minimized by executing as much as possible at the outer level while getting the necessary granularity for obtaining a good load balance.

We have modified the IBM XL runtime library to include the proposed mechanism to dynamically decide the most convenient way to exploit parallelism in nested parallel loops. It uses simple heuristics based on the number of iterations of the loops and the number of threads available.

We have executed several applications and measured the speedup achieved when executed with fixed parallelizations (outer, inner, and nested), compared with the speedup achieved with our proposal. Results show that applications executed with the new runtime reach the same or even better speedup than with the best static (a priori) parallelization.

The rest of the paper is organized as follows: in section 2 work related to our proposal is discussed. Then, section 3 describes the available ways to exploit parallel nested loops and the proposed semantics. Section 4 describes our runtime mechanism to choose between the different options. In section 5 an evaluation of the proposal on several applications is presented and section 6 concludes the paper and discusses future lines of research.

2 Related Work

One important way to reduce the overhead associated with parallelization is choosing an appropriate granularity as showed by Chen et al. [2]. Harrison et al. [3] proposed to dynamically select between a serial and parallel version of a loop based on the measured granularity. They propose having a threshold to decide whether to use the sequential or the parallel version. Moreira et al. [4] proposed a dynamic granularity control mechanism that tries to find the best granularity in a hierarchical task graph. Jin et al. [5] show the different overheads between parallelizing the outer or the inner loop of a Cloud Modeling Code.

Nested parallelism is an active area of research in the OpenMP community. Gonzalez et al. [6] discussed some extensions to the OpenMP standard to create teams of threads for nested execution. Another approach to nested parallelism was presented by Shah et al. [7] using work queues for nested parallel dynamic work. Different studies have been

done showing the benefit of using nested parallelism mainly on large SMP machines [8, 9, 10, 11, 12].

In [13] authors propose to convert parallel calls into parallel-ready sequential calls and to execute the excess parallelism as sequential code. In our work, we also exploit the idea of considering the programmer specification as a hint, but not only the potential parallelism, but also the way the application should exploit that parallelism: How many levels of parallelism, which level to exploit, and the thread distribution inside them.

In [14], authors propose a forall statement to unify the existing constructions to specify parallelism. At this moment, the OpenMP standard unifies the parallelism specifications in a set of directives. In the paper authors focused their work in the semantics of one level of parallelism, whereas this work is focused on multiple levels of parallelism and how to efficiently exploit them as a function of the application characteristics and resource availability.

3 Nested Loops Parallelization

When a programmer finds a nested loop such as the one shown in Figure 1, he has to decide the way to parallelize it. The current OpenMP standard supports three different approaches by simply introducing a *c\$omp parallel do* directive in the appropriate place(s). The most used option is only parallelizing the *outer* loop. This option is a good choice if the number of iterations of the outer loop is enough compared to the number of available processors to reach a good data distribution. However, if iterations are computationally intensive and there are few iteration, this option can result in heavy load imbalance. The second option is only parallelizing the *inner* loop. Parallelizing the inner loop will potentially provide smaller pieces of work so they can be distributed evenly between the available threads but it has more overhead due to work distribution and synchronization between threads. These overheads can be extremely prohibitive if the loop granularity is too fine. A third option is to parallelize both loops having *nested* parallelism. This choice suffers from the same problem as parallelizing the inner loop plus the overhead of distributing data from the outer loop but it has shown to be a good choice for systems with large number of processors [8].

The last possible configuration is a *mixed* approach and cannot be explicitly specified by the OpenMP programming model. The idea is to parallelize the maximum number of iterations from the outer loop that do not lead to load imbalance and then execute the

```
do i = 1,N
  do j = 1,M
    Compute_1()
  end do
end do
```

Fig. 1. Simple nested loops

```
!$omp parallel do
do i = 1,N
!$omp parallel do
  do j = 1,M
    Compute_1()
  end do
end do
```

Fig. 2. Proposed parallelization for nested loops

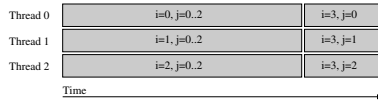


Fig. 3. Mixed parallelization approach for two nested loops (with four and three iterations) and three threads

remaining iterations sequentially, while parallelizing the inner loop to provide enough work to balance the overall execution (see Figure 3). As it is not supported by the standard it has to be coded manually in the source code and this is against the idea of simplicity and portability behind the OpenMP standard.

In this paper, we claim that parallel programmers must use OpenMP directives to declare the application parallelism (as shown in Figure 2) and not decide or impose which level of parallelism should be exploited. As we have seen, this decision can be difficult because it depends on a lot of parameters that may be only known at runtime. What we propose is to dynamically classify nested parallel loops and decide which configuration is more suitable depending on : loop granularity, number of iterations and number of available threads. Depending on these three conditions, the runtime will parallelize the outer loop, the inner loop, or both in a nested or a mixed approach. The classification and configuration will be done automatically by the runtime, transparent to the programmer. In giving this responsibility to the runtime (which is already possible through OMP_DYNAMIC environment variable), applications will be more portable as the runtime adjusts the applications to the specific environment where they are executed. Applications will also benefit from future optimizations in the runtime parallelizing mechanisms without having to modify them.

4 Runtime Support

To be able to decide the kind of parallelism (outer, inner, mixed or nested) to be used for a given loop the runtime has to: (1) discover the structure of the parallel loops of the application and their characteristics; (2) using that information, the runtime has to decide how to proceed and which iterations are assigned to each level in a mixed approach.

We have implemented a mechanism for both tasks inside IBM’s XL runtime library for OpenMP. The mechanism is fairly general and holds for multiple levels of nesting. Our prototype implementation works for iterations of the same weight. To extend it to any kind of loop, some other runtime balancing technique, like the Adjust schedule[15], should be integrated.

4.1 Discovering the Application’s Structure

Each time the application wants to start a parallel loop it calls the appropriate runtime service passing some parameters including the address of the outlined function that contains the parallelized code. This address is used to identify the parallel loops (as each has a different function). For each loop the runtime allocates a descriptor where relevant information is stored.

A stack of the calls (for opening parallelism) to the runtime is also maintained. When a new loop is encountered, if the stack is not empty, the loop is related to the stack's top loop. This way a tree is obtained that represents the call path of parallel loops in the application.

Though compiler analysis could also be used to detect the application's structure, it has not been used because we wanted to push the runtime detection to the limit and because static analysis can not always detect nested loops.

4.2 Deciding the Kind of Parallelization to Apply

When the application requests to open parallelism for a given loop, the runtime uses the information stored in the loop's descriptor to decide whether to serialize the loop (i.e., only inner loops will be parallelized), open parallelism for some of the iterations (i.e., mixed parallelization) or for all the iterations (i.e., outer or nested parallelization).

This decision is straight forward:

- If it has no inner loops open full parallelism at this level since it is the only one available.
- Otherwise, if the ratio of total iterations divided by the number of threads is
 - an integer then open full parallelism at this level. Inner levels will be serialized.
 - a real number greater than one then execute in parallel $\text{trunc}(\text{total iterations} / \text{number of threads})$ at this level. The remaining are executed serially afterwards parallelizing the inner level.
 - a real number lower than one then we choose to open full parallelism at this level and allow nested parallelism in inner loops.

Note that the decision mechanism lacks control granularity based on runtime measures or compiler information. This kind of control should be integrated in order to avoid opening excessive fine levels.

This decision is saved in the loop descriptor and reused afterwards. If a change in the information is produced (a nested loop is found, the number of iterations or threads change, ...) then it is recalculated.

When the chosen method is not nested parallelism then some (or all) iterations of the inner or outer loop must be serialized. This is achieved by taking a special path inside the runtime with almost no overhead that avoids all work distribution and synchronization code. This way there is no need to have two different version of the code (one parallel and serial) that could increase the executable object leading to greater memory consumption.

When nested parallelism is used the number of teams of threads in the outer loop is also chosen as this allows better utilization than an all versus all approach to nested parallelism. This decision is done by computing the greatest common divisor (*gcd*) between the number of iterations on the outer level and the number of threads available. The *gcd* is the number of teams that will be spawned in the outer level. In the inner level all available processors will be distributed evenly across the teams. Also, the loop records which threads were part of the team and tries to reuse them in the following executions resulting in improvements due to data affinity.

The first time a loop is executed, the runtime does not know if the loop will have a nested loop inside or not. For this reason, it is executed slightly different. At this point

there is two possible decisions: execute the loop assuming that there is a nested loop inside or execute the loop assuming there is no nested loop inside.

In the first case, the runtime will execute the loop as whether no nested loop was inside. If later on a nested loop is discovered, a new decision could be applied. If the number of steps of the algorithm is high this is a simple solution but when this is not the case such initial decision can heavily reduce the benefits of the algorithm.

We propose to assume that there will be an inner nested loop and calculate the normal decision strategy for this loop. If this decision leads to a mixed parallelization approach this will give the runtime the opportunity to detect inner parallelism during the execution of the initial chunk of outer iterations. If no inner parallelism is detected, the tail of of the iteration space is also parallelized afterwards.

5 Evaluation

5.1 Environment

The evaluation was done in a dedicated 16-way 375Mhz Power3 with 4 Gb of memory running AIX 5.2. The compiler used was IBM's xlf compiler using `-qsmp=omp` and `-O3` flags. In nested versions `-qsmp=nested` was also used. The OpenMP runtime was IBM's XL support library to which our proposal was added.

For all the applications we show the speedup (taking serial time as base time) obtained with the different parallelization approaches (parallelizing the inner level, parallelizing the outer level, parallelizing both in a mixed way as explained in section 3 or parallelizing both with nested parallelism) and the one achieved by our runtime (described in section 4). The mixed parallelization was done manually.

5.2 Synthetic Application

First, a synthetic application was evaluated for proof of concept. It contains two parallel loops, each having a computational delay. Two different cases are shown, (1) one where the number of iterations of the outer loop is large (500 iterations) and (2) another where is small and is not multiple of the number of threads (17 iterations).

As we can see, the best way to parallelize the loops is different depending on the scenario. For the case with 500 iterations (see Figure 4(a)) the best choice is to parallelize the outer loop while for the case with 17 iterations (see Figure 4(b)) the best speedup is achieved parallelizing both loops in a mixed approach. In both cases, the runtime is able to decide an appropriate way of parallelizing the loops.

5.3 LU Kernel

We have also evaluated a LU kernel. It uses blocking (size of each block is 150 elements) and the schedule used in all loops is STATIC. The outermost parallel loop decreases the number of iterations executed at each step by one and thereby changes the amount of parallelism in each step.

The speedups for the LU computation for different matrix size is shown in Figures 5(a), 5(b) and 5(c). As can be seen, the performance of the different approaches

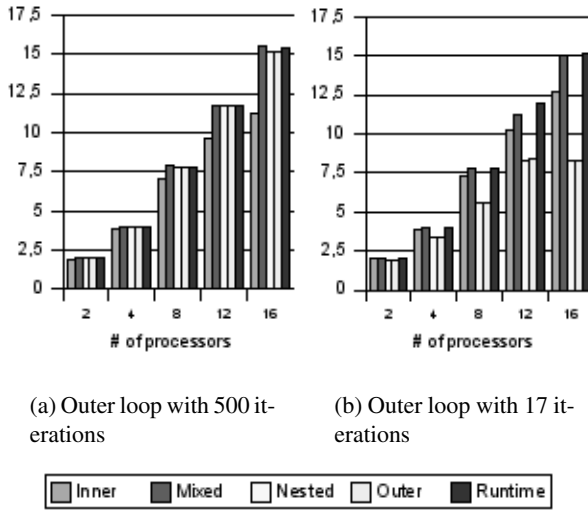


Fig. 4. Speedups for the synthetic cases

varies significantly depending on the matrix size and the number of threads available. The runtime decisions are close to the best most of the time. It can be seen that if the number of processors is large the nested parallelization approach outperforms the others, including the runtime. In this case the runtime actually chooses to use nested parallelism but the number of teams it decides to use is not optimal leading to some performance degradation.

5.4 MBLOCK Benchmark

The mblock benchmark is a multi-block algorithm that performs a simulation of the propagation of a constant source of heat in an object. The output of the benchmark is the temperature at each point of the object. The heat propagation is computed using the Laplace equation. The object is modeled as a multi-block structure composed of a number of rectangular blocks. Blocks are connected through a set of links at specific positions. The internal representation of the blocks is as follows.

After an initialization phase, an iterative solver computes the temperature of each point in the structure. Each block computation can be done in parallel, also parallelism exist inside each block. Propagation of the temperature between blocks can also be done in parallel.

Three different inputs have been used. First, a set composed of eight large blocks (128x128x128 elements). Second, a set composed by sixteen medium sized blocks (40x40x64 elements) and then another set of eight medium sized blocks.

The nested version used to evaluate this benchmark is a modified one that allows to fix the number of teams in the outer level and distribute the threads between all the teams equally.

In Figure 6(a) it can be seen how the best methods changes with the number of threads. From two to eight threads all the method have similar speedups. With 12 threads,

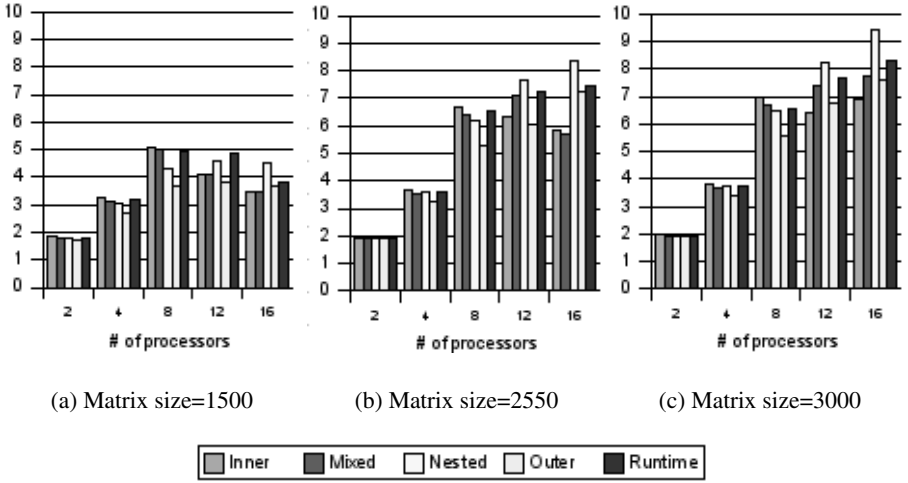


Fig. 5. Speedups for the LU kernel

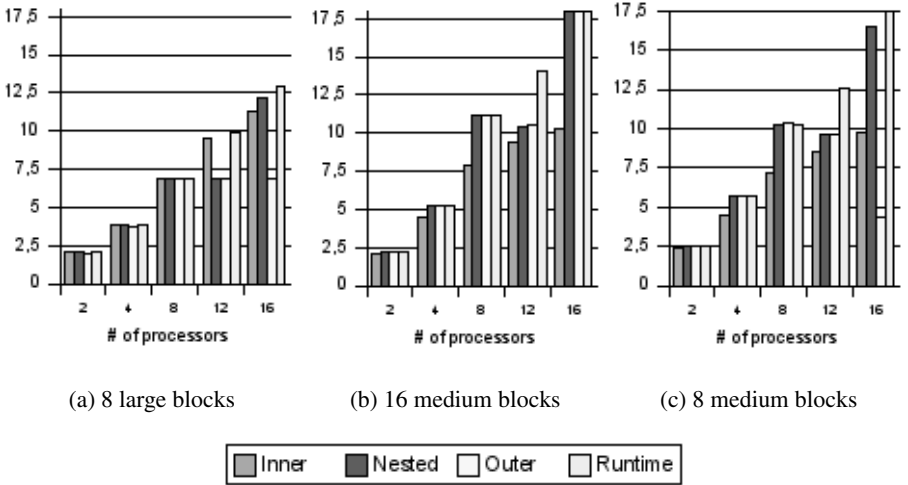


Fig. 6. Speedups for the MBLOCK benchmark

parallelizing the inner loop results in a 38% of gain with respect the other methods but with sixteen threads using nested parallelism gives a 9% of gain with respect the inner method.

When the input is changed to sixteen medium blocks (see Figure 6(b)), the inner method performs poorly while the outer and the inner methods perform very well. When the input is only eight medium blocks (see Figure 6(c)), the outer method scales only up to eight threads because there is not enough work to feed more threads. So, in this case the best choice is to use nested parallelism.

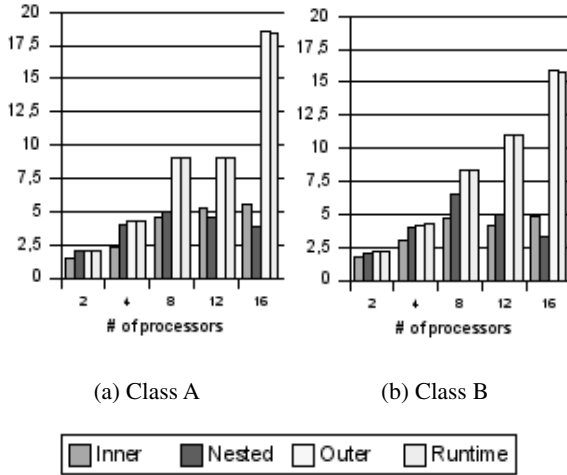


Fig. 7. Speedups for the SP-MZ benchmark

In all cases, the runtime chooses a method that works close to the best one or even better. The difference in performance between the runtime and the nested method with 12 threads, as seen in figures 6(b) and 6(c), is because the runtime uses a better distribution of the threads through the teams than the one used in the nested version.

5.5 SP-MZ Benchmark

The SP-MZ benchmark is part of the NPB-MZ benchmark suite [16]. It is a modified version of the SP NAS Parallel benchmarks [17]. The original discretization mesh was divided into a two-dimensional tiling of three-dimensional zones. Inside each zone a normal SP problem is resolved. Zone computation can be done in parallel as well as computation inside each block. Predefined classes A (16 zones of $32 \times 32 \times 16$ elements) and B (64 zones of $38 \times 26 \times 17$ elements) were evaluated.

We can see in Figures 7(a) and 7(b) that given any number of threads this benchmark works best choosing outer level parallelism. The runtime gets the same performance in all the cases.

It is interesting to note that in the case with twelve processors in Figure 7(a) the runtime chooses to use a mixed parallelization scheme but it still gets the same performance as parallelizing the outer level. We believe this is because the outer level has higher benefits from locality (that is why we see superlinear behavior). On the other hand the mixed approach, not having as good locality, achieves the same time by avoiding the imbalance present in the outer parallelization.

6 Conclusions and Future Work

OpenMP allows the specification of nested parallel loops. In that case, deciding which of those loops must be parallelized is a hard task that, in addition, depends on runtime parameters such as the number of available threads or the problem size.

In this work we claim that the programmer must use OpenMP directives to specify the existing parallelism and let the runtime library decide how to exploit this parallelism. Also, we present an approach to exploit nested parallel loops to be applied in those cases where exploiting only one of the levels of parallelism is not enough to reach a good scalability.

The idea is to exploit the outer level of parallelism as much as possible and execute the remaining iterations in serial, exploiting, in these iterations, the inner level of parallelism.

We have implemented our proposal inside IBM's XL library and executed several benchmarks. Our results show that the runtime, with a simple set of heuristic, is able to determine the appropriate way of parallization between outer, inner, nested or mixed parallelization approaches in most cases. The heuristics used by the decision algorithm may be improved for example by using rules based on feedback measures that will allow better decision of the type of parallelism to use. Different ways to find the optimal number of teams need to be explored to fully exploit this kind of technique.

The present work will also be extended to other worksharing constructs as sections or shares. Also the current mechanism will we extended to be able to cope with inhomogenous distributions. Measures taken by the runtime will be useful in these kind of scenarios.

Further, we will explore the potential of using compile time information inside the runtime. This information could be not only the application structure but the size of data structures or estimated execution time of parallel regions. Runtime should not rely on that information but it could take advantage of it when available for example to reduce the initial overhead.

Acknowledgements

Authors want to thank Marc González for his help on nested parallelism concepts. This work has been supported by the IBM CAS program, the POP European Future Emerging Technologies project under contract IST-2001-33071 and by the Spanish Ministry of Science and Education under contract TIC2001-0995-C02-01.

References

1. OpenMP Organization. Openmp fortran application interface, v. 2.0. *www.openmp.org*, June 2000.
2. D. Chen, H. Su, and P. Yew. The impact of synchronization and granularity in parallel systems. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 239–248, 1990.
3. W. Harrison III and J. H. Chow. Dynamic control of parallelism and granularity in executing nested parallel loops. In *Proceedings of IEEE Third Symposium on Parallel and Distributed Processing*, pages 678–685, 1991.
4. J. E. Moreira, D. Schouten, and C. Polychronopoulos. The performance impact of granularity control and functional parallelism. In C. H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 481–597. Springer, Berlin, Heidelberg, 1995.

5. H. Jin, G. Jost, D. Johnson, and W. Tao. Experience on the parallelization of a cloud modelling code using computer-aided tools. Technical report, NASA Ames Research Center, March 2003. NAS-03-006.
6. M. González, J. Oliver, X. Martorell, E. Ayguadé, J. Labarta, and N. Navarro. OpenMP extensions for thread groups and their run-time support. *Lecture Notes in Computer Science*, 2017:324–??, 2001.
7. S. Shah, G. Haab, P. Petersen, and J. Throop. Flexible control structures for parallelism in openmp. In *1st European Workshop on OpenMP*, September 1999.
8. E. Ayguadé, M. González, X. Martorell, and G. Jost. Employing nested openmp for the parallelization of multi-zone computational fluid dynamics applications. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2004.
9. E. Ayguade, X. Martorell, J. Labarta, M. Gonzalez, and N. Navarro. Exploiting multiple levels of parallelism in openmp: A case study. In *Proceedings of the 1999 International Conference on Parallel Processing*, September 1999.
10. R. Blikberg and T. Sørenvik. Nested parallelism: Allocation of processors to tasks and openmp implementation. In *Proceedings of the 1999 International Conference on Parallel Processing*, September 1999.
11. Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa. Performance evaluation of openmp applications with nested parallelism. In Sandhya Dwarkadas, editor, *Languages, Compilers, and Run-Time Systems for Scalable Computers, 5th International Workshop, LCR 2000*, volume 1915 of *Lecture Notes in Computer Science*. Springer, 2000.
12. R. Blikberg and T. Sørenvik. Nested parallelism in openmp. In *ParCo 2003*, 2003.
13. Seth C. Goldstein, Klaus E. Schauer, and David E. Culler. Enabling Primitives for Compiling Parallel Languages. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, May 1995.
14. P. Dechering, L. Breebaart, F. Kuijman, and K. van Reeuwijk. Semantics and implementation of a generalized forall statement for parallel languages. In *11th International Parallel Processing Symposium (IPPS '97)*, April 1997.
15. E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martínez, R. Silvera, and X. Martorell. Is the schedule clause really necessary in openmp? In M.J. Voss, editor, *Proceedings of the International Workshop on OpenMP Applications and Tools 2003*, volume 2716 of *Lecture Notes in Computer Science*, pages 69–83, June 2003.
16. R.F. Van der Wijngaart and H. Jin. Nas parallel benchmarks, multi-zone versions. Technical report, NASA Ames Research Center, July 2003. NAS-03-010.
17. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.