Automatic Thread Distribution For Nested Parallelism In OpenMP

Alejandro Duran
Computer Architecture Department, Technical University of Catalonia
cr. Jordi Girona 1-3
Despatx D6-215
08034 - Barcelona, Spain
aduran@ac.upc.edu

Marc Gonzàlez
Computer Architecture Department, Technical University of Catalonia
cr. Jordi Girona 1-3
Despatx C6-E207
08034 - Barcelona, Spain
marc@ac.upc.edu

Julita Corbalán
Computer Architecture Department, Technical University of Catalonia
cr. Jordi Girona 1-3
Despatx C6-203
08034 - Barcelona, Spain
juli@ac.upc.edu

ABSTRACT

OpenMP is becoming the standard programming model for shared–memory parallel architectures. One of its most interesting features in the language is the support for nested parallelism. Previous research and parallelization experiences have shown the benefits of using nested parallelism as an alternative to combining several programming models such as MPI and OpenMP. However, all these works rely on the manual definition of an appropriate distribution of all the available thread across the different levels of parallelism. Some proposals have been made to extend the OpenMP language to allow the programmers to specify the thread distribution.

This paper proposes a mechanism to dynamically compute the most appropriate thread distribution strategy. The mechanism is based on gathering information at runtime to derive the structure of the nested parallelism. This information is used to determine how the overall computation is distributed between the parallel branches in the outermost level of parallelism, which is constant in this work. According to this, threads in the innermost level of parallelism are distributed.

The proposed mechanism is evaluated in two different environments: a research environment, the Nanos OpenMP research platform, and a commercial environment, the IBM XL runtime library. The performance numbers obtained validate the mechanism in both environments and they show the importance of selecting the proper amount of parallelism in the outer level.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel programming; D.3.4 [**Processors**]: runtime environments; D.4.1 [**Process Management**]: Multiprocessing; D.4.8 [**Performance**]: Measurements; C.4 [**Performance of Systems**]: Measurement techniques

General Terms

Algorithms,Experimentation, Performance

Keywords

OpenMP, nested parallelism, thread clustering

1. INTRODUCTION

Shared–memory parallel architectures are becoming common platforms for the development of CPU demanding applications. Today, these architectures are found in the form of small symmetric multiprocessors on a chip, on a board, or on a rack with a modest number of processors (usually less than 32). In order to benefit from the potential parallelism they offer, programmers require programming models to develop their parallel applications with a reasonable performance/simplicity trade-off.

OpenMP [20] is becoming the standard for specifying the parallelism in applications for these small shared-memory parallel architectures. Its success comes from the fact that OpenMP makes the parallelization process easy and incremental. One of the most interesting aspects in the OpenMP programming model is that it considers nested parallelism. However, the specification does not provide the necessary implementation hints to efficiently exploit nested parallelism.

The research framework consisting of the NanosCompiler [11] and the NthLib [18, 17] runtime library provides full support for nested parallelism in OpenMP. some researchers have used this framework to parallelize their applications using nested parallelism [12, 3]. Some extensions to the OpenMP specification have been proposed to efficiently exploit nested parallelism and allow programmers to specify the appropriate allocation of resources to the different levels of parallelism [13]. Basically, those extensions introduce thread–clustering mechanisms in the OpenMP programming model. Other research works have also shown the performance benefits of thread–clustering mechanisms when exploiting nested parallelism [4, 5].

We extended this research framework towards a global solution for the automatic selection of the best parallelization strategy for OpenMP applications. Along the parallelization process, programmers often attempt try different parallelization strategies, either single level or multilevel. Deciding which to use depends of several factors, such as the available number of threads or specific application characteristics. It may even depend on the input data sets. In the case of nested parallelism, the programmer needs to specify how the available resources (threads) are distributed among the levels of parallelism. As we will show in the next section, this decision is not an easy task. For this reason, in this paper, we propose a mechanism to automatically compute the appropriate thread distribution based on information gathered at runtime. This information will give an accurate description of the structure of the nested parallelism.

The structure of the paper is as follows: section 2 presents the main motivations for this paper. Section 3 describes the main contribution of this paper: the automatic thread groups definition. Section 4 presents the performance evaluation. Section 5 describes the main contributions of previous research works in nested parallelism. Finally, section 6 presents the conclusions of this paper and outlines some future work.

2. MOTIVATION

In order to motivate the proposal of this paper, we summarize in this section the main observations and conclusions of previous research works [4, 5, 2, 13, 3].

Figure 1 shows a simplified version of the structure of BT-MZ, one of the codes included in the NAS multizone benchmarks [9]. We will use it to show the kind of situations where nested parallelism can be exploited to increase the performance of the application. The code performs a computation over a *blocked* data structure. For each block of data (or zone), some work is performed in a subroutine called *adi*. A first level of parallelism appears since all zones can be computed in parallel. This corresponds to a *task* level parallelism and is coded by the parallelizing directive PARALLEL DO. A STATIC work distribution will be performed among the threads that execute this first level of parallelism. The definition of another level of parallelism is possible: the computation performed in each zone of data is organized in the form of a parallel loop. This parallel level would correspond to *data* parallelism.The code in subroutine *adi* contains a parallelizing directive for this loop. For this second level of parallelism, a STATIC scheduling is set too. A *time step* loop encloses the overall computation and at the end of each iteration, there is some data movement to update zone boundaries. This iterative structure is common in most numerical applications and it is necessary in any technique that dynamically improves the behavior of an application based on its past behavior.

The programmer can choose between two parallelization strategies that exploit a single level of parallelism. The first one exploits the inter–zone parallelism. We will call it the *outer* version. The second one exploits the intra–zone parallelism. This is the *inner* version. The performance for the *outer* version is clearly limited by the number of zones. Using more threads than the number of zones does not contribute

```
      ...
      do step = 1, niters
        ...
    C Inter-zone parallelism
    !$OMP PARALLEL NUM_THREADS(num_groups)
    !$OMP DO
        do zone = 1, num_zones
          CALL adi ( zone, ...)
        end do
    !$OMP END DO
    !$OMP END PARALLEL
        ...
    C Update zone boundaries
        ...
      end do
      ...
      end

      subroutine adi ( zone_id, ... )

    C Intra-zone parallelism
    !$OMP PARALLEL NUM_THREADS(zone_threads(zone_id))
    !$OMP DO
      do j = 1, k_size(zone_id)
        ...
      end do
    !$OMP END DO
    !$OMP END PARALLEL
      end
```

Figure 1: Main structure of the BT-MZ code with nested parallelism.

to improve performance. In addition, zones may have different sizes and lead to an unbalanced execution in which some of the threads waste execution cycles waiting for the others to finish their work. In case of important size differences, the unbalance degree might cause a noticeable performance degradation. So, two main factors are limiting the performance of the outer version. First, the relation between the number of available threads and the number of zones. Second, the unbalance degree expressed through the size differences between the zones. In both cases, the *outer* version is not going to adapt and increment the performance when an important number of threads (32 or more) are available.

Other issues limit the *inner* version performance. The most important issue is granularity. Creating the work in this level of parallelism among a large number of threads might cause that the work assigned to each thread is too small to take profit from the parallel execution. The finest granularity that can be exploited is conditioned by the runtime overheads related to parallelism creation and termination, work distribution and thread synchronizations. These overheads are going to be noticeable when executing with a large number of threads (again, 32 or more).

After exploring the single level possibilities, it is necessary to point out why a nested strategy would overcome the detected limitations. By exploiting both levels of parallelism (inner and outer levels), nothing is gained, unless threads are arranged in a way that avoids the grain size and work unbalance problems. Regarding the grain size problem, the

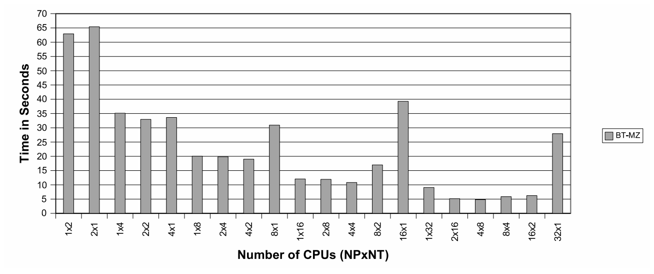


Figure 2: BT-MZ class A execution times on a IBM Regatta

only solution is to forbid that all available threads execute the inner level of parallelism. So, Thread clustering strategies are the solution. The main idea is to create, for each block or zone, an different set of threads that will execute the work defined at the inner level of parallelism. The NUM_THREADS clause in the source code is used for that purpose. This strategy solves the grain size problem, but it does nothing to solve the possible work unbalance created in the outer level of parallelism. In order to face that, the thread sets should be defined according to the amount of work assigned to each parallel branch in the outer level. In the example, this is done by having different values for the argument of the NUM_THREADS clause, depending on the *zone* that a set of threads is going to work on. In order to illustrate the impact of this thread clustering strategy, Figure 2 shows how the execution time of the BT-MZ application changes with different allocations of threads in the outer level (NP) and inner level (NT) [3]. NT represents the average value as its exact value is different for each zone since their have different sizes in this application. For each number of processors (2, 4, 8, 16 or 32) the best parallelization strategy is different. This difficulty in determining the most appropriate thread distribution between the levels of parallelism is the main motivation for the work in this paper. Our proposal is to rely on runtime mechanism to automatically derive the optimal thread distribution at runtime.

3. AUTOMATIC THREAD GROUPS

In this section we describe the main issues to consider to develop a runtime mechanism that computes optimal distributions of threads. The main issues to consider are the number of groups in the outermost parallel level and a thread reservation for the inner levels. These two factors totally define a thread distribution. Actually, these factors plus the distribution of work among threads determine the appropriateness of a thread distribution. Depending on the unbalance of the work distribution a thread distribution will or will not succeed in improving the performance of the application. If the thread reservation for the inner levels of parallelism is not in concordance with the work distribution work unbalance will persist. Thus, three elements condition the benefits that can be obtained from a thread distribution: the number of outer groups, the number of threads assigned to each group, and the work distribution.

Our objective is to find an appropriate thread distribution without programmer intervention. The implementation of the runtime system needs to deal with the three elements previously mentioned. Our proposal focuses only on tuning the thread distribution for the inner levels. Our runtime implementation is not going to adjust the number of outer groups or the work distribution. These two factors will be specified in the application by the programmer using the appropriate OpenMP directives. Given a work distribution and a number of groups the runtime will be able to derive the best thread assignment for the inner level. The only requisite for our implementation is that the program behaves in a iterative manner.

3.1 Methodology

For each nest of parallel regions where the runtime will automatically compute a thread distribution the programmer previously defined a number of groups and a work distribution schema. The initial assumption is that each group weights the same in terms of computation, so the available threads are uniformly distributed. The runtime needs to gather some kind of information that helps it to infer how the computation is distributed among the groups. The detection of work unbalance will then be translated to changes on the thread distribution: increasing the number of threads for those groups heavily weighted and decreasing the number of threads in those with a lesser load. Therefore, deciding what kind of information will be gathered, is an important issue of the implementation. Assuming that a gathering mechanism is available, we need to define a procedure to infer the work distribution and link it to a policy to redistribute the threads accordingly. At this point, it is necessary to have some evaluating process, pointing out the improvement that will be obtained from applying the new thread distribution. A function predicting the benefit has to be implemented, but under some threshold control, in order to avoid unnecessary changes in the distribution. Notice that the lack of such mechanism would open the system to undesirable effects (e.g. constantly moving threads across the groups or even thread ping-pong).

Figure 3 shows our general framework for computing the optimal thread distribution. It is a typical feedback guided scenario in which three phases can be distinguished. A first one where the information is gathered and transformed in order to obtain a description of the parallelism structure in terms of load balance. A second one where a new thread distribution is established through the obtained information. Finally a validation phase where the distribution is accepted or not.

3.2 Implementation

This section describes the implementation decisions and the details of the three phases mentioned in the previous section: *Runtime information sampling*, *Thread distribution policy* and *Validation of a thread distribution.*

3.2.1 Runtime information sampling

In the implementation of this phase, we had to made to decisions: what information will the runtime system measure and it will consider this data is to obtain a model of the parallelism structure of the application. In our implementation, execution time is the preferred metric, but of course, the reliability of the measurements depends on the places selected for the probes.

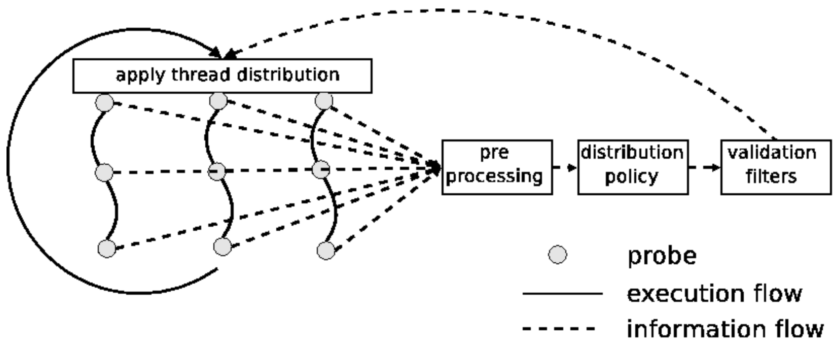


Figure 3: Framework design overview

What to sample, where to sample

Our runtime system places time probes at the beginning and at the end of the execution of each thread. This allows for a good approximation of the work each group of threads does, unless the runtime system introduces unreasonable overheads that distort the measurements. Notice that the way the probes are placed make the accumulated time include all the overheads related to thread creation/termination, barrier synchronizations and so on. We expect these overheads will be low enough so they will not interfere in the sampling.

A better, but more complex, approach is placing the probes at the beginning of the work sharing construct and before the implicit barrier at the end of each work sharing construct. Accumulating the time in each work sharing we will obtain a good sample for the amount of work of each group of threads.

Information preprocessing

Our runtime implementation takes the measurements in the outermost level of parallelism as a description of the parallelism structure of the application. The variance between those measurements will report the unbalance degree that was obtained since the last thread distribution was applied. The measurements are normalized to the minimum sample. That is, the runtime finds the thread with the minimum execution time and it divides the rest of values by this one. This normalization erases the small variations of the sampling process giving a more meaningful collection of computational weights for each group of threads.

In this step, the runtime could also compute additional metrics from the sampled data, like the degree of unbalance, that could be useful to the distribution policy.

3.2.2 Thread distribution policy

After the parallelism structure is modelled, the thread distribution policy is invoked to compute the optimal distribution of threads between the outer level groups. Different policies could be implemented here.

We have implemented an algorithm based on the work from Gonzàlez et al.[13]. Figure 4 shows the code algorithm written in Fortran90 syntax. The variable **weight** contains the proportions previously mentioned. The variable **ngroups** refers to the number of threads devoted to the execution of the parallelism in the outermost level. The Variable **howmany** specifies the number of threads to be used for the execution of the parallelism in the inner levels. The variable **num_threads** refers to the total number of threads available. First, the algorithm assigns one thread per *group*. This ensures that at least one thread is assigned for the execution of the inner levels of parallelism that *group* encounters. After that, the rest of threads are distributed according to the proportions in vector **weight**.

```
pos=minloc(samples(1:ngroups))
weight(1:ngroups)=samples(1:ngroups)/samples(pos)
howmany(1:ngroups) = 1
do while (sum(howmany(1:ngroups)) .lt. num_threads)
   pos = maxloc(weight(1:ngroups)/
                               howmany(1:ngroups))
   howmany(pos) = howmany(pos) + 1
end do
```

Figure 4: Thread distribution algorithm.

3.2.3 Validation of a thread distribution

After the policy computes a thread distribution a number of filters may be used to validate that we will be obtain a benefit after applying the new distribution.

Critical path validation filter

Moving threads is not free. There is some penalty mainly caused because of data movement across caches. This filter discards those cases where moving threads between groups will not result in an performance increment that overcomes the penalty of the movement.

Using the time samples and the new thread distribution, this filter computes an estimation of the critical path that would result if the new distribution was applied. The time samples are divided by the number of assigned threads in the new distribution. If the maximum value of this divisions, i.e. the critical path, is greater than the one obtained with the current thread distribution the distribution. It also is discarded if the time difference is below a certain threshold. Figure 5 shows the described algorithm in Fortran90 syntax. We suggest 5% as the threshold value.

Ping-pong effect

This filter is in charge on detecting a ping-pong situation, where a number of distributions are applied cyclically without any real gain. This filter uses the history of computed

```
threshold= number between 0 and 1
pos=maxloc(samples(1:ngroups)/howmany(1:ngroups))
new_critical_path=samples(pos)/howmany(pos)
if ( new_critical_path .lt prev_critical_path .and.
     (new_critical_path - prev_critical_path) .gt.
     (threshold * prev_critical_path) ) then
  return true
else
  return false
endif
```

Figure 5: Critical path validation algorithm.

thread distributions to detect this situation. When the filter detects the ping-pong it chooses the distribution that best worked and it filters out any other.

More threads than chunks of work anomaly

The distribution algorithm is not aware of the number of chunks of work that will be defined in the inner levels of parallelism. The parallel regions in the inner levels might offer different degrees of parallelism translated to how chunks of work are distributed among the threads. It is possible that one parallel region offers enough chunks so all the threads can work, while others do not. Our implementation is sensitive to this effect. But if this situation occurred it will be reflected in the measurements and the thread distribution algorithm will solve it.

4. EVALUATION

The proposal in this paper was implemented in two different environments: a research environment, the NANOS OpenMP runtime library and a commercial environment, the IBM XL runtime library. We use for the evaluation two benchmarks from the NAS Multizone suite. The following sections describe the execution environments and the most important aspects of the benchmarks. These aspects condition the nested parallelism execution and the thread distribution computation.

4.1 Execution environments

4.1.1 NANOS environment

We executed the benchmarks that use the NANOS runtime library in a Silicon Graphics Origin2000 system [16] with 64 R10k processors, running at 250 MHz with 4 Mb of secondary cache each. The benchmarks were compiled was done using the NANOS compiler which performs all the OpenMP transformations. This compiler is a source-to-source compiler. To build the binaries, we used the native compilers with the following flags:*-64 -Ofast=ip27 -LNO:prefetch ahead=1:auto dist=on.*

4.1.2 IBM XL environment

Benchmarks using the IBM XL environment were run in a p690 32-way Power4 [8] machine at 1.1 Ghz with 128 Gb of RAM. We used IBM's XLF compiler with the following flags:*-O3 -qipa=noobject -qsmp=omp.* The operating system was AIX 5.2.

4.2 Applications

We evaluated two applications from the NAS Multizone benchmark suite [9]: BT and SP with input data class A. These benchmarks solve discretized versions of the unsteady, compressible Navier Stokes equations in three spatial dimensions. The two applications compute over a data structure composed by blocks. The computation processes one block after another. Then, some data is propagated between the blocks. Parallelism appears at two levels. At the outermost level, all the blocks can be processed in parallel. At the innermost level, the computation in each block can be coded through parallel `do` loops. This structure allows for the definition of a two-level parallel strategy. The main difference between the two benchmarks is the composition of the blocks, which is going to be the main issue in the evaluation. In the case of the BT-MZ, the input data is composed by blocks of different sizes, while in SP-MZ all blocks are of the same size.

4.2.1 BT-MZ class A

BT-MZ, using input class A, works with an input data composed by 16 three-dimensional blocks. For each block, BT-MZ computes different phases. All of them implement a nest of three `do` loops: one per dimension. Table 1 shows the size of each block, according to the dimension sizes. Usually, the outermost loop corresponds to the K-dimension and is parallelized. Two phases must be parallelized on the J-dimension because of data dependences.

Applying a single level strategy forces the programmer to choose between two possibilities. Exploiting the parallelism between blocks, which is limited by two factors: only 16 threads can obtain work as there are only 16 blocks, and what is worst, the parallelism is highly unbalanced. Last column on table 1 shows the proportions between the blocks. Between the the smallest block (block 1) and the largest one (block 16) there is a factor of 19.9. The other possibility is to exploit the parallelism inside the block. The loops over the K-dimension and J-dimension (this last one only in two phases) are parallelized. According to the information in table 1, the K-dimension is 16 for all blocks and the J-dimension varies within 13, 21, 36 and 58. When the loop on the K-dimension is executed in parallel, only 16 threads will obtain work. Again, this limits the performance. Therefore, best option is to use a two-level strategy, combining the *inter* and *intra* block parallelism. This strategy generates 16 per 16 chunks of work. Therefore, even with a large number of threads all them get work. But, the problem of unbalance persists.

BT-MZ comes with a two load balancing algorithms. These algorithms represent slightly more than a 5% of the total code. The first algorithm, distributes blocks in the outer level of parallelism trying all groups have a similar amount of computational work. The second one assigns a number a threads in the inner level of parallelism to each of the outer groups based on the computational load of the zones assigned to each outer group. Both methods are calculated before the start of the computation based on knowledge of the data shape and computational weight of the application.

4.2.2 SP-MZ class A

The SP-MZ benchmark is very similar to the BT-MZ benchmark. The main difference between both benchmarks is related to the sizes of the blocks in the input data structure. In the SP-MZ benchmark the input data is composed by 16 three-dimensional blocks, all of them of the same size:

Block	I-dimension	J-dimension	K-dimension	Size	Proportions
1	13	13	16	2704	1
2	21	13	16	4368	1.61
3	36	13	16	7488	2.76
4	58	13	16	12064	4.46
5	13	21	16	4368	1.61
6	21	21	16	7056	2.61
7	36	21	16	12096	4.47
8	58	21	16	19488	7.20
9	13	36	16	7488	2.76
10	21	36	16	12096	4.47
11	36	36	16	20736	7.66
12	58	36	16	33408	12.35
13	13	58	16	12064	4.46
14	21	58	16	19488	7.20
15	36	58	16	33408	12.35
16	58	58	16	53824	19.9

Table 1: Block sizes for BT-MZ class A.

32 x 32 x 16 (K, J, I dimensions, respectively). The computation evolves over different phases, where each phase is implemented by three nested loops, one per dimension. The outermost loop in each phase is always parallelized.

As in the case of BT-MZ, a single level strategy can not be efficiently applied. Just exploiting the *inter* block parallelism is not unbalanced at all, but suffers from the same limitation regarding the number of threads to be used. Thus, again, a two level strategy is the best option: combining *inter* and *intra* block parallelism.

4.3 Methodology

4.3.1 BT-MZ

As it has been explained in previous section 4.2.1, the BT-MZ benchmark incorporates two load balancing algorithms which perform data and thread distribution. As it is possible to activate/deactivate each type of distribution, different benchmark versions have been evaluated. The comparisons were done under equal data distribution, since what we want to evaluate is how powerful is the proposed mechanism for automatic thread distribution. Thus, the performance results have been organized according to the data distribution. The experiments have been separated in two different sets. First, the data distribution algorithm was deactivated, and the zones were distributed among the thread groups following a STATIC scheme. Second, the data distribution was defined by the existing algorithm in the benchmark. In each case, four versions were tested with different thread distributions. The *automatic* version dynamically computes the thread distribution. This version was the only version using our proposed framework for automatic thread distribution. In all others, our framework was deactivated. The *uniform* version defined a uniform thread distribution among the groups. The *manual* version performed a thread distribution computed by the load balancing algorithm in the benchmark. The *automatic* version tries different distributions until it finds a stable distribution. The *preassigned* version used the stable distribution obtained from the execution of the *automatic* version.

4.3.2 SP-MZ

The SP-MZ benchmark presents an input organized in equally sized zones. Thus, no unbalance is produced in the outermost level of parallelism. Having a totally balanced input, allowed to check that the proposed mechanisms adapt to this situation.

Two versions of the benchmark were evaluated. The *automatic* version works with the automatic thread distribution mechanism. This version starts using a uniform thread distribution, and its execution allowed to check that the proposed mechanism detects the absence of unbalance, and that it does not change to a non-uniform thread distribution. The *manual* version used a constant uniform thread distribution. The adaptive mechanisms are deactivated. Comparing these two versions we quantified the overheads in the thread distribution algorithm.

4.4 Evaluation in the NANOS environment

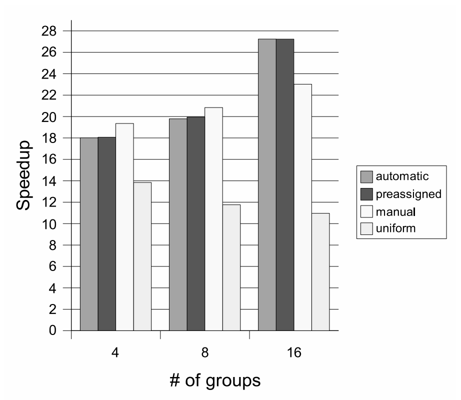
4.4.1 BT-MZ class A

Figure 6 shows the results for the BT-MZ benchmark. All numbers are speed-up with serial time as base time. Graphics in figure 6(a) shows the speed-up for all the versions using a STATIC zone distribution (the data distribution mechanism is deactivated). The *automatic* version obtained 18.01, 19.79 and 27.24 for 4, 8 and 16 groups. The *uniform* version speed-ups were: 13.84, 11.76 and 10.96. When we compare this numbers we clearly see that the *automatic* version computed a thread distribution that balanced the application. We conclude the same when we compare the *uniform* and *manual* versions results. The numbers for the *manual* version were: 19.35, 20.84 and 23.02. Again, the *manual* version outperformed the *uniform* version.

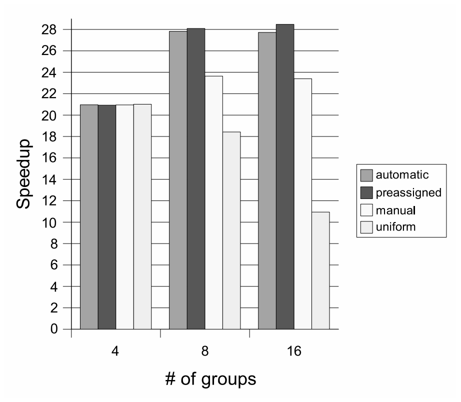
The *manual* and *automatic* perform similarly but in the case of 16 groups. With 16 groups the *automatic* version obtains 4 more points of speed-up. These results support our idea that the thread distribution can be computed by the runtime freeing the programmer of tasks of computing himself. Table 2 shows the thread distribution for each version. In the case of the *automatic* version it shows the stable distribution. The differences in performance between these two versions

Version	Groups 4	Groups 8	Groups 16
automatic	3 3 8 18	1 2 1 3 4 7 3 13	1 1 1 1 1 1 1 2 1 1 2 4 1 2 4 8
manual	3 6 9 14	1 3 1 4 3 6 4 10	1 1 1 1 1 1 2 2 1 2 2 4 2 2 4 5

Table 2: BT-MZ class A thread distributions for STATIC zone distribution (NANOS environment)



(a) with STATIC zone distribution



(b) with zone clustering

Figure 6: BT-MZ class A speedups using 32 cpus (NANOS environment)

are explained through the thread distribution obtained by each version.

Comparing the *automatic* and *preassigned* versions we can see how the initial phase of the *automatic* version affects its performance. We consider the initial phase until a stable thread distribution is found. The *preassigned* obtains 18.07, 19.95 and 27.23 points of speed-up with 4, 8 and 16 groups respectively. From these numbers, which are statistically identical, we can conclude that the automatic mechanisms have a low overhead and the application can afford to use them.

Figure 6(b) shows the performance numbers for all the versions using the zone distribution algorithm encoded in the application. We reach similar conclusions using this distribution scheme. The *automatic* and *manual* versions perform better than the *uniform* version. This means that data distribution could not perfectly balance the application as a non-uniform thread distribution was still necessary (for 8 and 16 groups). The *automatic* version performed better than the *manual* version because it found better thread distributions. Table 3 shows the differences between the thread distributions computed by the *automatic* and the *manual* version. The *automatic* version and the *preassigned* version had the same results. We can see, again, that the mechanism overheads are low.

4.4.2 SP-MZ class A

Figure 7 shows the performance numbers, with 32 processors, for all the evaluated versions. The *manual* version obtained 9.51, 16.34 and 24.28 points of speed-up with 4, 8 and 16 groups. Notice the differences in performance, depending on the number of groups. We suspect that this is due to locality effects. A lesser number of groups than the number of zones causes that more than one zone is assigned to the same group of threads. The worst scenario is with 4 groups where 4 zones are assigned to each group. In the case of 16 groups, the zones were distributed one zone per group of threads, so threads only work with one zone. The same phenomena is observed in the evaluation of the *automatic* version. This versions performs as the *manual* version. With 4, 8 and 16 groups, the obtained speed-ups are 9.41, 16.42 and 24.29 respectively. The thread distribution mechanism was verified. With any number of groups, the *automatic* version always defines a uniform thread distribution. In the case of 4 groups, 8 threads were assigned per group. With 8 and 16 groups, 4 and 2 threads were assigned to each group, respectively. We conclude that the proposed mechanisms adapt well to balanced work distribution, without introducing unreasonable overheads. Differences in performance related to the number of groups need further research. As it was pointed out in previous section 3,determining the appropriate number of groups is out of the scope of this research work. Only the thread distribution possibilities were explored.

4.5 Evaluation in the IBM XL environment

4.5.1 BT-MZ class A

Figure 8(a) shows the results for the evaluation of BT-MZ with a STATIC zone distribution. The *automatic* and the *manual* versions outperformed the *uniform* version, which is heavily unbalanced. The *manual* version is between a 59% faster, with 4 groups, and a 119% faster, with 16 groups, than the *uniform* version. The *automatic* version gains range from a 48% with 4 groups to a 134% with 16 groups. This means both algorithms computed balanced thread distributions. For 8 and 16 groups, the *automatic* version slightly outperformed the *manual* version. When executed with 4 groups the speed-up of the *manual* version is higher. But, the *preassigned* version, that used the distribution computed

Version	Groups 4	Groups 8	Groups 16
automatic	8 8 8 8	7 4 4 4 4 3 3 3	7 4 4 2 2 2 2 1 1 1 1 1 1 1 1 1
manual	8 8 8 8	6 4 4 3 4 3 4 4	5 4 4 2 2 2 2 2 1 2 1 1 1 1 1 1

Table 3: BT-MZ class A thread distributions with zone clustering (NANOS environment)

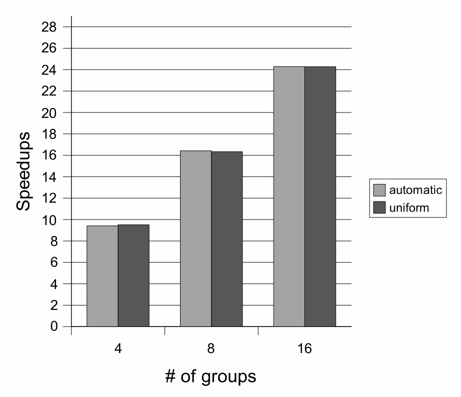
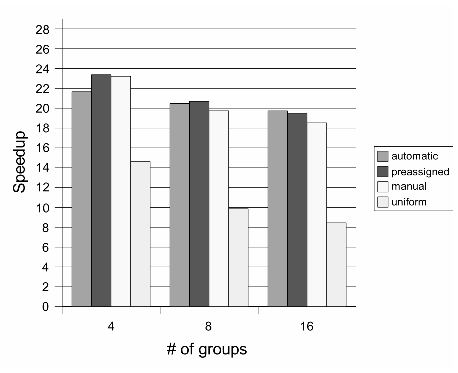
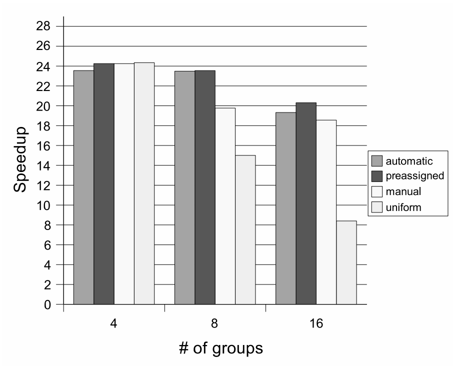


Figure 7: SP-MZ class A speedups using 32 CPUs (NANOS environment)



(a) with STATIC zone distribution



(b) with zone clustering

Figure 8: BT-MZ class A speedups using 32 CPUs (XL environment)

by the *automatic* version, performs as well as the *manual* version. Table 4 shows the distributions used by both versions. We can conclude there are some minor overheads in the library that reduce the gain of a good thread distribution.

Figure 6(b) shows the performance results for BT-MZ using the zone clustering algorithm encoded in the application. Defining 4 groups, the zone distribution was balanced. That is why all the versions achieved the same speed-up. When 8 and 16 groups are defined, the *manual* version outperformed the *uniform* distribution of threads: it was a 32% faster with 8 groups and a 121% faster with 16 groups. The *automatic* version outperformed the *uniform* and the *manual* versions. Compared to the *uniform* version, it was a 57% faster with 8 groups and a 130% faster with 16 groups. Compared to the *manual* version, the increase in performance was a 19% with 8 groups and a 4% of increase with 16 groups. Thread distributions between the versions can be compared in table 5.

Versions using a STATIC zone distribution performed worst than those that use the zone clustering algorithm. This is because after clustering is applied the data distribution is less unbalanced. Which is easier to balance. And being more balanced they perform better.

Also note that for all the versions and with both data distributions, as more groups were used the speed-up decreased. When more groups are used there are fewer threads available to be moved between groups as each group needs at least one thread. With a high number of groups, a *perfect* thread distribution would assign fractions of threads to the groups and this can not be done.

4.5.2 SP-MZ

Figure 9 shows the speed-up obtained by each evaluated version of SP-MZ. Both versions, *manual* and *automatic*, obtained similar results. This means that mechanism overhead is low in this environment too. The *automatic* version determines, in all the cases, a uniform distribution of threads.

All speed-up numbers are similar independent of the number of groups used. This contrasts with the previous test environment where the group number had a great influence in performance.

5. RELATED WORK

Nested parallelism is an active area of research in the OpenMP community. Several experiments [6, 14, 1, 19, 21, 15] present the possibility of mixing more than one programming model for exploiting nested parallelism. Typically, applications, which execute under such parallel strategy, define a first level of parallelism using a distributed memory paradigm plus a second level of parallelism implemented with shared

Version	Groups 4	Groups 8	Groups 16
automatic	3 5 10 14	1 2 1 4 2 6 3 13	1 1 1 1 1 1 2 2 1 2 2 4 1 2 4 6
manual	3 6 9 14	1 3 1 4 3 6 4 10	1 1 1 1 1 1 2 2 1 2 2 4 2 2 4 5

Table 4: BT-MZ class A thread distributions for STATIC zone distribution (XL environment)

Version	Groups 4	Groups 8	Groups 16
automatic	8 8 8 8	7 4 5 3 4 3 3 3	7 4 4 2 2 2 2 1 1 1 1 1 1 1 1 1
manual	8 8 8 8	6 4 4 3 4 3 4 4	5 4 4 2 2 2 2 2 1 2 1 1 1 1 1 1

Table 5: BT-MZ class A thread distributions with zone clustering (XL environment)

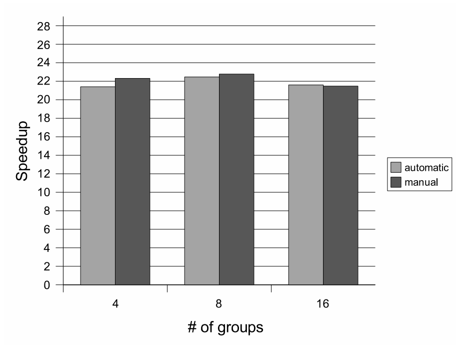


Figure 9: SP-MZ class A speedups using 32 CPUs (XL environment)

memory model, usually programmed with OpenMP. Results showed that nested parallelism performs better than conventional single level strategies.

Other authors worked with nested parallelism in a pure OpenMP model. Experiments with nested parallelism [2, 5] pointed out the main problems to get a good performance. Several authors [13, 4, 3] have argued that a *thread distribution* will solve these problems. The thread grouping mechanism was studied and organized as a proposal to extend the OpenMP language [2]: constructs were defined to allow programmer specify the most appropriate thread distribution between the levels of parallelism and. Also, this proposal presented an optimal algorithm to compute the thread distribution. But, all these works rely on the programmer providing the required information: the number of branches in the outermost parallel level and the computational weight associated to each branch. Programmers can obtain the computational weight through representative parameters of the computation, such as the amount of memory used in each branch or other specific characteristic of the application (number of "elements" treated by each computational branch, number of iterations, ...).

Thread distribution has been showed to be necessary and critical for performing well when exploiting nested parallelism. After several contributions, an open question remained: would it be possible to avoid the programmer intervention in order to obtain the most suitable thread distribution? In this paper we propose a mechanism to do it, based on the previous works of Duran et al. [7, 10]. Those works introduced the facility of sampling the execution time for each branch in the outermost level of parallelism. We use these measurements as an approximation of the computational weight in the overall nest of parallelism. From these weights is possible to obtain a suitable thread distribution for nesting parallelism exploitation.

6. CONCLUSIONS AND FUTURE WORK

This paper explored the viability, in the context of OpenMP nested parallelism, of a runtime mechanism to distribute threads in the inner level of parallelism between different groups in the outer level. Our proposal works by gathering information, at runtime, about the time each group spent doing useful work. Based on these measures, an algorithm calculates a new thread distribution that is applied afterwards. We have implemented the algorithm in two different environments which shows that the proposal is not dependant on a specific runtime implementation.

Our results confirmed that the proposed runtime mechanism performs as well, or even better, as algorithms handcrafted by the programmer of the application. Even, when application work well with a uniform distribution the mechanism shows a low overhead. This suggests that a runtime could use our mechanism as a default option for nested parallelism.

Results also showed that the different environments have different behavior with the same application and input. As the runtime technique reacts to the architectural differences (because they also imply differences in the sampled times) a runtime library that uses the techniques described in the paper becomes more portable, from a performance point of view, across different platforms.

As the evaluation showed, the number of groups is a critical parameter in the performance of an application. Future work will deal with the problem of automatically deciding how many groups should be used at the outer level. This will allow an optimal exploitation of nested parallelism without programmer intervention.

7. ACKNOWLEDGMENTS

This research has been supported by the Ministry of Science and Technology of Spain under contract TIN2004-07739-C02-01, the European Union under contract IST-2001-33071 and the IBM CAS program.

8. ADDITIONAL AUTHORS

Additional authors: Xavier Martorell (Technical University of Catalonia, email: `xavim@ac.upc.edu`), Eduard Ayguadé (Technical University of Catalonia, email: `eduard@ac.upc.`

edu), Jesús Labarta (Technical University of Catalonia, email: jesus@ac.upc.edu) and Raúl Silvera (IBM Toronto Lab, email: rauls@ca.ibm.com).

9. REFERENCES

[1] D. an May and S. Schmidt. From a vector computer to an mp-cluster - hybrid parallelization of the cfd code panta. In *Proceedings on the 2nd European Workshop on OpenMP (EWOMP2000)*, September 2000.

[2] 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.

[3] 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.

[4] R. Blikberg. Parallelizing amrclaw by nesting techniques. In *Proceedings on the 4th European Workshop on OpenMP (EWOMP2002)*, September 2002.

[5] R. Blikberg and T. Sørevik. Nested parallelism: Allocation of processors to tasks and openmp implementation. In *Proceedings of the 1999 International Conference on Parallel Processing*, September 1999.

[6] T. Boku, S. Yoshikawa, M. Sato, C. G. Hoover, and W. G. Hoover. Implementation and performance evaluation of spam particle code with openmp-mpi hybrid programming. In *Proceedings of the 3rd European Workshop on OpenMP (EWOMP2001)*, September 2001.

[7] J. Corbalán, A. Duran, and J. Labarta. Dynamic load balancing of mpi+openmp applications. In *Proceedings of the International Conference on Parallel Processing (ICPP2004)*, August 2004.

[8] I. Corporation. Power4 system microarchitecture. October 2001.

[9] R. V. der Wijngaart and H.Jin. Nas parallel benchmarks, multi-zone versions. Technical report, NASA Ames Research Center, July 2003. NAS-03-010.

[10] A. Duran, R. Silvera, J. Corbalán, and J. Labarta. Runtime adjustment of parallel nested loops. In *Proceedings of the Workshop on OpenMP Applications and Tools (WOMPAT2004)*, May 2004.

[11] M. Gonzalez, E. Ayguade, J. Labarta, X. Martorell, N. Navarro, and J. Oliver. Nanoscompiler: A research platform for openmp extensions. In *Proceedings on the 1st European Workshop on OpenMP (EWOMP1999)*, September 1999.

[12] M. Gonzalez, E. Ayguade, X. Martorell, J. Labarta, and P. V. Luong. Dual-level parallelism exploitation with openmp in coastal ocean circulation modeling. In *Proceedings of the International Workshop on OpenMP: Experiences and Implementations (WOMPEI 2002)*, 2002.

[13] 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*, 2017:324–332, 2001.

[14] P. Kloos, F. Mathey, and P. Blaise. Openmp and mpi programming with a cg algorithm. In *Proceedings of the 2nd European Workshop on OpenMP (EWOMP2000)*, September 2000.

[15] P. Lanucara and S. Rovida. Conjugate-gradient algorithms: an mpi-openmp implementation on distributed shared memory systems. In *Proceedings on the 1st European Workshop on OpenMP (EWOMP1999)*, September 1999.

[16] J. Laudon and D. Lenoski. The sgi origin: A ccnuma highly scalable server. In *Proceedings of the 24th. Annual International Symposium on Computer Architecture, (ISCA97)*, 1997.

[17] X. Martorell, E. Ayguadé, N. Navarro, J. Corbalan, M. Gonzàlez, and J. Labarta. Thread fork/join techniques for multi-level parallelism exploitation in numa multiprocessors. In *Proceedings of the 13th. ACM International Conference on Supercomputing (ICS99)*, June 1999.

[18] X. Martorell, E. Ayguadé, N. Navarro, and J. Labarta. A library implementation of the nano-threads programming model. In *Proceedings of the 2nd. Europar Conference (EUROPAR1996)*, August 1996.

[19] F. Mathey, P. Kloos, and P. Blaise. Openmp optimisation of a parallel mpi cfd code. In *Proceedings on the 2nd European Workshop on OpenMP (EWOMP2000)*, September 2000.

[20] O. Organization. Openmp fortran application interface, v. 2.0. *www.openmp.org*, June 2000.

[21] L. Smith. Epcc development and performance of a hybrid openmp/mpi quantum monte carlo code. In *Proceedings of the 1st European Workshop on OpenMP (EWOMP1999)*, September 1999.