

# Automatic Thread Distribution For Nested Parallelism In OpenMP

## 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.*

## 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 [18] 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 nested parallelism is considered. However, the specification does not provide implementation hints that are necessary to efficiently exploit nested parallelism.

The research framework consisting of the NanosCompiler [9] and the NthLib [16, 15] runtime library provides full support for nested parallelism in OpenMP. This framework has been used in by some researchers to parallelize their applications making use of nested parallelism [19, 12]. 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 [10]. 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 [11, 8].

Recently, this research framework has been extended towards a global solution for the automatic selection of the best parallelization strategy for OpenMP applications. Along the parallelization process, it is quite common that programmers cope with different parallelization strategies, either single level or multilevel. The decision is usually dependent of several factors, such as the available number of threads or specific application characteristics. It may even depend of the input data sets. In the case of nested parallelism, the decision needs to specify how the available resources (threads) are distributed among the levels of parallelism. As we will show in the next section, this is not an easy task. For this reason, this paper proposes a mechanism to automatically compute the appropriate thread distribution based on information gathered at runtime, information that is supposed to 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 [11, 8, 7, 10, 12].

Figure 1 shows a simplified version of the structure of BT-MZ, one of the codes included in the NAS multizone benchmarks [17]. We will use it to show the kind of situations where nested parallelism can be exploited and leads to a performance increment. 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 would correspond 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 computation allows for the definition of another level of parallelism, as 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 level of parallelism, a STATIC scheduling is going to be set. The overall computation is enclosed in a *time step* loop 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 is necessary in any technique that dynamically improves the behavior of an application based on past behavior.

```

...
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

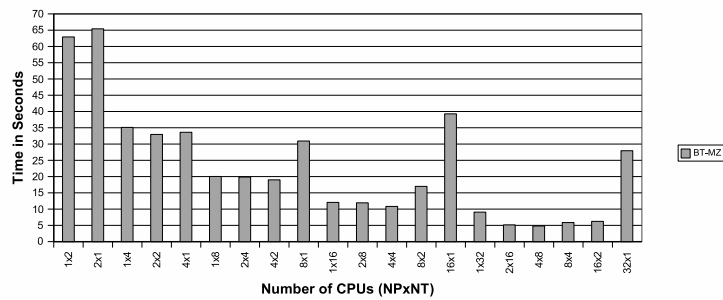
```

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

The programmer can define two parallelization strategies that just exploit a single level of parallelism. The first one just exploiting the inter-zone parallelism, that we will call the *outer* version. The second one just exploiting the intra-zone parallelism, 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 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. 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 have to be analyzed when the *inner* version is considered. The most important issue is granularity. Spreading 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. This is going to be noticeable when executing with a large number of threads (again, 32 or more).



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

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 get arranged in a particular way that avoids the grain size and work unbalance problems. Regarding the the grain size problem, the only solution is to forbid having the inner level of parallelism executed by all the available threads. This leads to thread clustering strategies. The main idea is to create, for each block or zone, an different set of threads that will execute the work coming out from the inner level of parallelism. The `NUM_THREADS` clause in the source code can be used for that purpose. This is solving the grain size problem, but nothing is done to arrange the possible work unbalance created in the outer level of parallelism. In order to face that, the thread sets could be defined according to the amount of work assigned to each parallel branch in the outer level. In the example, this needs of 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 this, Figure 2 shows how the execution time of the BT-MZ application changes with different allocation of threads in the outer level (NP) and inner level (NT) [12]. Since in this application zones have different sizes, NT is different for each zone, so NT represents the average value. Notice that for each number of processors (2, 4, 8, 16 or 32) the best parallelization strategy may be different. This difficulty in determining the most appropriate thread distribution between the levels of parallelism has been taken as the main motivation for the work in this paper. The proposal is to rely on runtime mechanism to automatically derive it at runtime.

### 3 Automatic Thread Groups

In this section we describe the main issues to consider, while looking for a runtime mechanism that allows for the computation of optimal thread distributions. Based on what has been described in the previous section 2, setting the number of groups in the outermost parallel level, plus a thread reservation for the inner levels of parallelism, is enough for having the thread distribution totally defined. Thus, the number of groups and the number of threads assigned to each one seem to be the main issues to consider. Actually, the appropriateness of a thread distribution is determined by this two factors plus the work distribution that is going to be performed during the execution of the nested parallelism. Depending on the existing work unbalance, a thread distribution might or might not succeed in improving the performance. Notice that the thread reservation for the inner levels of parallelism might not be in concordance with the work distribution. In that case the work unbalance persists. Thus, three elements condition the benefits that can be obtained from a thread distribution: the number of groups, the number of threads assigned to each group, and the work distribution.

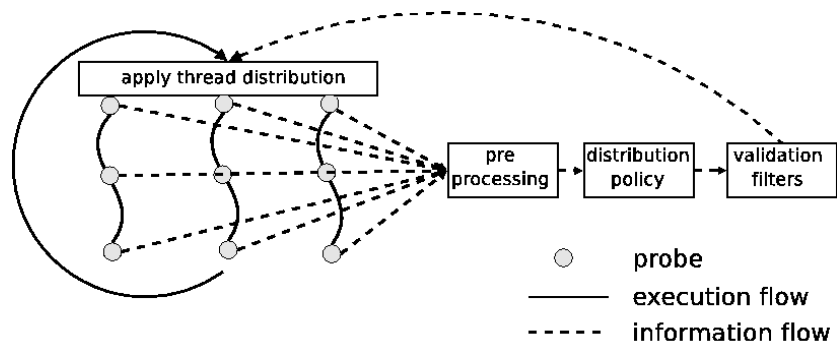
Opposite to what has been described in section 2, now the main objective is to achieve an appropriate thread distribution automatically, without any programmer hint. The implementation of the runtime system has to deal with the previously mentioned three elements. The proposal that this paper presents is based on tuning the thread reservation for the inner levels. The number of groups and the work distribution are going to be treated out of the runtime implementation. Particularly, the number of groups is going to be set by the programmer, while the work distribution is going to be defined by the application through the available schemes in the OpenMP standard. The runtime has to be able of deriving the best thread distribution for a given work distribution and number of groups. The only requisite for our implementation is that the program behaves in an iterative manner.

#### 3.1 Methodology

For each nest of parallel regions where it is wanted the runtime to automatically compute the thread distribution, a number of groups has been set and a work distribution has been defined. Initially, it is assumed that each group is equally weighted in terms of computation, so the available threads are uniformly distributed. The runtime needs to gathering some kind of information that helps it to derive how the computation is distributed among the groups. Detecting possible work unbalance will then be translated to changes on the thread distribution, that is, increasing the number of threads for those groups heavily weighted, decreasing those with lesser load. Therefore, deciding what kind of information has to be gathered, is going to be one important issue in the implementation. Assuming that the gathering mechanism is available, it has to be defined a procedure to infer the work distribution and link it to a policy to distribute the threads accordingly. At this point, it is necessary to have some evaluating process, pointing out the improvement that is going to be obtained from the new thread distribution. A function predicting the gains has to be implemented, but under some threshold control, in order to avoid

unnecessary changes in the distribution. Notice that not having such mechanism, would open the system to undesirable effects like constantly moving threads across the groups or even thread ping-pong.

Figure 3 shows our general framework for computing the optimal thread distribution. From what has been introduced, 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.



**Figure 3. Framework design overview**

## 3.2 Implementation

This section describes the implementation decisions and 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, two main decisions have to be taken: what is the runtime system going to measure and how this data is going to be treated to obtain a description of the parallelism structure. In our implementation, execution time is the preferred metric, but of course, the reliability of the measurements depends on the selected places to place the probes.

#### What to sample, where to sample

Our runtime system places time probes at the beginning and end of the execution of each thread. This allows for a good approximation of the work each group of threads is doing, unless the runtime system introduces unreasonable overheads that distort the measurements. Notice that the way the probes are placed makes the accumulated time include all the overheads related to thread creation/termination, possible barrier synchronizations and so on.

A better, but somewhat complex, approach is to put the probes at the beginning and before the implicit barrier at the end of each work sharing construct. Accumulating the time in each work sharing we 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 an initial description of the parallelism structure. The deviation in those measurements is going to report the unbalance degree that has been obtained since the last applied thread distribution. The measurements are normalized to the minimum one. That is, the runtime computes the thread with the minimum execution time and it divides the rest of values by this minimum. This normalization erases the small deviations of the sampling 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 approximated, 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 González et al.[10]. Figure 4 shows the code algorithm written in Fortran90 syntax. Variable **weight** contains the proportions previously mentioned. Variable **ngroups** and **howmany** refers to the number of threads devoted to the execution of the parallelism in the outermost level. Variable **howmany** specifies the number of threads to be used for the execution of the parallelism in the inner levels. Variable **num\_threads** refers to the total number of available threads. 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.**

### 3.3 Validation of a thread distribution

After a thread distribution has been computed by the appropriate policy, a number of filters may be used to validate that a certain benefit is going to be obtained from applying the new distribution.

### **Critical path validation filter**

This filter allows to discard those cases where moving threads, which has some penalty (data movement between caches), it is not predicted to have a reasonable increase of performance (to overcome the penalty).

Using the time samples and the new thread distribution, this filter computes an estimation of the critical path that is going to be obtained if the new distribution was applied. This is done by dividing the time samples by the number of assigned threads in the new distribution. If the maximum value is greater than the one obtained with the current thread distribution the distribution is discarded. Figure 5 shows the described algorithm in Fortran90 syntax.

```
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.**

### **Ping-pong effect**

Based on the history of thread distributions used, this filter would be in charge on detecting a ping-pong situation, where a number of distributions are applied cyclically without any real gain. When the filter detects this situation 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 about the number of chunks of work that are going to be defined in the inner levels of parallelism. The parallel regions in the inner levels might offer different degrees of parallelism translated in chunks of work to be distributed among the threads. In that sense, it is possible that one parallel region offers enough chunks to feed all the threads, while others do not. Our implementation is sensitive to that, but we expect that the sampled execution times are going to reflect this situation. Thus, the thread distribution algorithm can redress it.

## **4 Evaluation**

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

## 4.1 Execution environments

### 4.1.1 NANOS environment

Benchmarks evaluated with the NANOS runtime library have been executed in a Silicon Graphics Origin2000 system [20] with 64 R10k processors, running at 250 MHz with 4 Mb of secondary cache each. The compilation was done using the NANOS compiler which performs all the OpenMP transformations. This compiler is a source-to-source compiler. We finally build binaries with 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 have been run in a p690 32-way Power4 [21] machine at 1.1 Ghz with 128 Gb of RAM. We used the IBM's XLF compiler with the *-O3 -qipa=noobject -qsmp=omp*, and the operating system was AIX 5.2.

## 4.2 Applications

We have evaluated two applications from the NAS Multizone benchmark suite [17]. The applications are: 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 formed by blocks. The computation treats one block after the other, and then some data is propagated between the blocks. Parallelism appears at two levels. At the outermost level, all the blocks can be treated in parallel. At the innermost level, the computation in each block is 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 SP-MZ works with a data structure where all blocks are of the same size.

### 4.2.1 BT-MZ class A

BT-MZ in class A works with an input data formed by 16 three-dimensional blocks. For each block, the computation goes along different phases, all of them implementing 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. Because of data dependences, in only two phases the nest of the loops is structured placing the loop running on the J-dimension in the outermost level and also parallelized.

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. The reader can see that 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 in the block. The loops over the K-dimension and J-dimension (this last one only in two phases) have to be parallelized. Again the performance is going to be limited. 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. This means that when the loop executing on the K-dimension is parallelized, only 16 threads can obtain work to execute in parallel, while when the loop executing on the J-dimension is parallelized, up to 58 threads can execute in parallel, depending on the block. Therefore, best option is to use a two-level strategy, combining the *inter* and *intra* block parallelism.

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

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

#### 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 compose by 16 three-dimensional blocks, all of them of the same size: 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 the BT-MZ, a single level strategy can not be efficiently applied. Notice that the option of just exploiting a the *inter* block parallelism is not unbalance at all, but suffers from the same limitation regarding the number of threads to be used. Only 16 threads can obtain work in a such strategy as only 16 blocks form the input data. Thus, again, a two level strategy is the best option: combine the *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 an algorithm performing data and thread distribution. As it is possible to activate/deactivate each type of distribution, different benchmark versions have been evaluated. The comparisons have been 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. A first one where the data distribution algorithm has been deactivated, and the zones are distributed among the thread groups following a *STATIC* scheme. A second one where the data distribution is defined by the existing algorithm in the benchmark. In each case, four different versions have been tested, depending on the applied thread distribution. The *automatic* version dynamically computes the thread distribution. This corresponds to the one using the automatic mechanisms proposed in this paper. The *uniform* version defines a uniform thread distribution among the groups. The *manual* version performs a thread distribution computed by the existing algorithm in the benchmark. Finally, the *preassigned* version executes under the final thread distribution reached by the *automatic* version. The *preassigned* version allows for having an approximate measurement of the effects of the automatic thread distribution mechanisms. Notice that those mechanisms rely on an initial phase where the application evolves looking for a stable thread distribution. It is this thread distribution that is used in the *preassigned* version. Thus, the comparison of the *automatic* and *preassigned* versions gives us an estimation of the effects of the mentioned initial phase.

### 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, allows for checking that the proposed mechanisms adapt to this situation, leading to the definition of a uniform thread distribution across the thread groups.

Two versions of the benchmark have been evaluated. The *automatic* version works with the automatic thread distribution mechanism. This version starts the execution with a uniform thread distribution, and its execution allows us to check if the proposed mechanism detect the absence of unbalance, and does not change to a non-uniform thread distribution. The *manual* version executes under a constant uniform thread distribution. The adaptive mechanisms are deactivated. The comparison of this two versions allows us to check the presence of unreasonable overheads, or even deficiencies in the thread distribution algorithm that lead to wrong thread distributions.

## 4.4 Evaluation in the NANOS environment

### 4.4.1 BT-MZ class A

The results for the BT-MZ benchmark are displayed in figure 6 and all of them are expressed by means of speed-up.

Graphics in figure 6(a) shows the performance numbers for all the versions under a *STATIC* zone distribution (the data distribution mechanism is deactivated). The *automatic* version obtains 18.01, 19.79 and 27.24 for 4, 8 and 16 groups. Compared with the *uniform* version (speed-ups of 13.84, 11.76 and 10.96), it is clear that the thread distribution has to be computed according to the work unbalance caused by the differences between the zones. The same result is obtained by comparing the *uniform* and *manual* versions. The numbers for the *manual* version are 19.35, 20.84 and 23.02. Again, the *manual* version outperforms the *uniform* version.

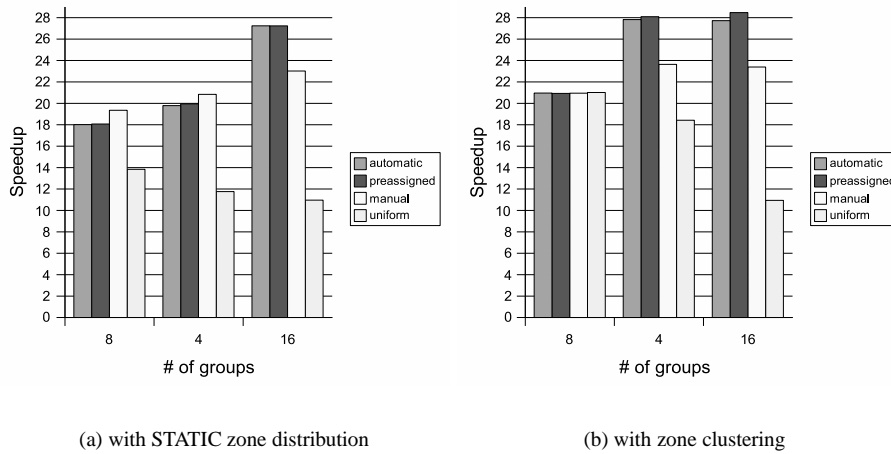


Figure 6. BT-MZ class A speedups using 32 cpus (NANOS environment)

The comparison between the *manual* and the *automatic* versions allows for a main conclusion. Both versions perform similarly, unless for the case of 16 groups, where the *automatic* version outperforms in 4 points of speed-up. Thus, there is no need of including the thread distribution mechanism in the application, while this can be done by the runtime with as much precision as a programmer's hand coded approach. The differences in performance between these two versions are explained through the thread distribution. Table 2 shows the thread distribution for each version. Remember that the zones are distributed under a *STATIC* scheduling. Thus, this work distribution introduces some work unbalance that has to be reflected in the thread distribution. In the case of 4 groups defined, we observe a considerable difference in the threads devoted to the most weighted group (in computational terms). For the *manual* version 14 threads are assigned to this group, while 18 are in the *automatic* version. For the 8 and 16 groups cases, similar situations arrive. It is because of these differences that the *automatic* version outperforms. A first conclusion is that deriving the thread distribution from information gathered at runtime allows for better distributions.

Version	Groups 4	Groups 8	Groups 16
<i>automatic</i>	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
<i>manual</i>	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)**

Finally, the comparison between the *automatic* and *preassigned* versions allows for determining the effects of the initial phase that the *automatic* spends in reaching a stable thread distribution. The *preassigned* performs with 18.07, 19.95 and 27.23 points of speed-up with 4, 8 and 16 groups respectively. Clearly, very similar to the *automatic* version, what allows us to conclude that the automatic mechanisms can be reasonably afforded by the application.

In figure 6(b) shows the performance numbers for all the benchmark versions under a the zone distribution algorithm provided by the application. Similar conclusions can be obtained under this zone distribution scheme. The *automatic* and *manual* versions are the ones that perform the best. Again, the thread distribution solves the work unbalance introduced by the native algorithm in the application. Surprisingly, the obtained results are better than the ones obtained in the executions under a STATIC zone distribution. We suspect that this is related to the zone clustering that the application includes. Notice that, as much as work unbalance is concerned, the automatic thread distribution is conditioned by the unbalance degree, the number of groups and the number of available threads. First, notice that the number of groups determines the number of remaining threads to distribute. It is possible that a huge unbalance degree, combined with the definition of a particular number of groups (not leaving over enough threads to balance the application), forbids to establish an appropriate thread distribution. In that sense, the native zone clustering algorithm reduces the work unbalance, increasing the possible benefits from the thread distribution.

Table 3 shows the differences between the obtained thread distributions for the *automatic* and *manual* versions. Those differences explain the different behavior of both versions in terms of speed-up.

Version	Groups 4	Groups 8	Groups 16
<i>automatic</i>	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
<i>manual</i>	8 8 8 8	6 4 4 3 4 3 4 4	5 4 4 2 2 2 2 1 2 1 1 1 1 1 1 1

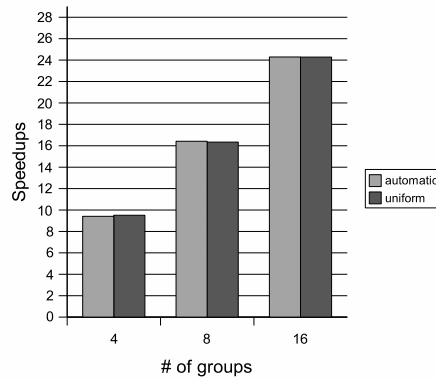
**Table 3. BT-MZ class A thread distributions with zone clustering (NANOS environment)**

Finally, the comparison between the *automatic* and *preassigned* versions shows that the possible overheads coming from the examining phase, where the application evolves to the final thread distribution, can be afforded by the application. Both versions perform similarly.

#### 4.4.2 SP-MZ class A

Figure 7 shows the performance numbers for all the evaluated versions. The *manual* version obtains 9.51, 16.34 and 24.28 points of speed-up with 4, 8 and 16 groups, executing 32 processors. 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 case appears with 4 groups where 4 zones are assigned to each group. In the case of 16 groups, the zones are 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 the same 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 has been checked and any number of groups, the *automatic* version always determines a uniform thread distribution along its execution. In the case of 4 groups, 8 threads are assigned per group. With 8 and 16 groups, 4 and 2 threads are assigned to each group, respectively. The main conclusion is that the proposed mechanisms are able to correctly adapt to a well balanced work distribution, without introducing unreasonable overheads. Finally, the differences in performance related to the number of groups need for 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 have been explored.

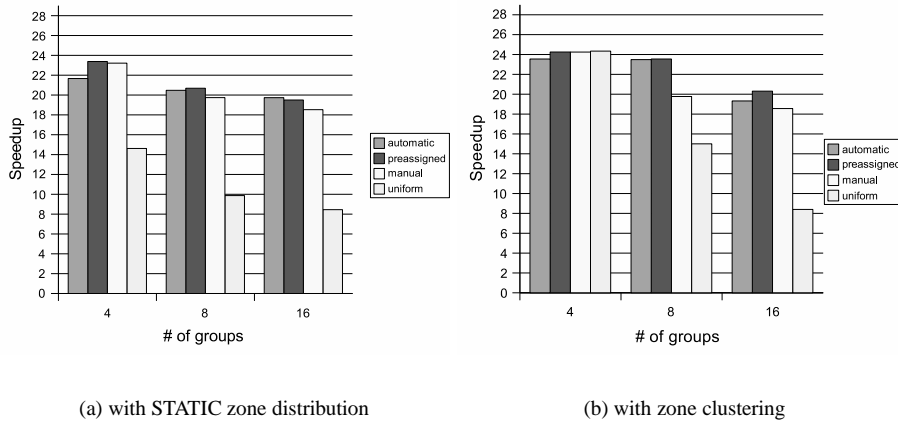


**Figure 7. SP-MZ class A speedups using 32 CPUs (NANOS environment)**

## 4.5 Evaluation in the IBM XL environment

### 4.5.1 BT-MZ class A

In figure 8(a), we can see results for the evaluation of BT-MZ with an *STATIC* zone distribution. Both *automatic* and *manual* versions outperform the *uniform* version, which is heavily unbalanced. The gains obtained with the *manual* version range from 59%, with 4 groups, to a 119%, with 16 groups. With the *automatic* version the gains go from 48% with 4 groups to a 134%, with 16 groups. This means, both algorithm are computing balanced thread distributions. For 8 and 16 groups, the *automatic* version slightly outperforms the *manual* version. But, when executed with 4 groups the *automatic* version obtains a better speedup. But, if we look at the *preassigned* version, which uses the distribution computed by the *automatic* version, we can see that obtains the same performance as the *manual* version. Table 4 shows the distributions used by both versions.



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

This means that, there are some overheads in the library, that reduce the gain from a good thread distribution.

Also note that for all the different versions, as the number of groups increases it decreases the speedup obtained decreases.

Version	Groups 4	Groups 8	Groups 16
<i>automatic</i>	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
<i>manual</i>	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)**

Figure 6(b) shows the results obtained for the versions using the zone clustering algorithm embedded in the application. When using 4 groups, the zone clustering algorithm, produces a balanced distribution of zones. That is why all the different versions have the same speedup. In table 5 can also be seen as both the *manual* and *automatic* distribute the threads evenly between the groups. Using 8 and 16 groups, the *manual* version outperforms the *uniform* distribution of threads, obtaining a 32% and a 121% of improvement with 8 and 16 groups respectively. The *automatic* version outperforms, both the *uniform* and the *manual* versions. Compared to the *uniform* version, it obtains a 57% increase with 8 groups, and a 130% of increase with 16 groups. Compared to the *manual* version, the increase in performance is a 19% with 8 groups, and a 4% of increase with 16 groups. This increase is due to a better thread distribution as can be seen in table 5.

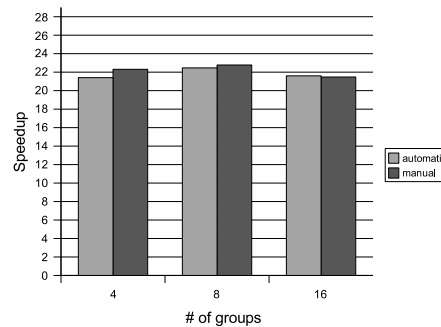
Version	Groups 4	Groups 8	Groups 16
<i>automatic</i>	8 8 8 8	7 4 5 3 4 3 3 3	7 4 4 2 2 2 2 1 1 1 1 1 1 1 1 1
<i>manual</i>	8 8 8 8	6 4 4 3 4 3 4 4	5 4 4 2 2 2 2 2 1 2 1 1 1 1 1 1

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

Is interesting to note, that, in general, versions using STATIC zone distribution obtain a lower speedup than those that use the zone clustering algorithm. The reason for this is that after clustering is applied a less unbalanced distribution exists, which can be better balanced. Being more balanced a better speedup is obtained.

Also, in all the cases, as more groups are used less performance is obtained. The main reason, for this is, that when you have more groups there are there are fewer threads to be moved between groups, as each groups needs to have at least one thread. This makes it difficult to get a *perfect* balance as it would imply distributing fractions of threads which is not possible.

#### 4.5.2 SP-MZ



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

Figure 9 shows the performance numbers for all the evaluated versions. In both approaches, *manual* and *automatic*, the obtained speedups are similar to the balanced version. This means that the overheads of the mechanism don't have a big impact in the performance. The *automatic* version determines, in all the cases, an uniform distribution of threads. Results also show that differences in performance due to the number of groups used are quite small (below 7%).

## 5 Related work

Nested parallelism is an active area of research in the OpenMP community. In several experiments [1, 2, 3, 4, 5, 6] mixing more than one programming model are presented as a possibility for exploiting nested parallelism. Typically, applications executing under such parallel strategy define a first level of parallelism using a distributed memory paradigm plus a second level of parallelism implemented with shared memory, usually programmed with OpenMP. The obtained results show that nested parallelism gives an opportunity to increase performance in front conventional single level strategies. The reason why hybrid programming models are used is that nested parallelism is not available in many of the vendor platforms.

Other proposals have been made, in a pure OpenMP programming model style. In several experiments [7, 8] with nested parallelism are presented, and the main problems to get performance are pointed out. The introduced solutions are based on the definition of *thread groups*, although there is not yet a general solution in the form of a proposal to extend the OpenMP programming model.

Different authors [10, 11, 12] have pointed out the necessity of a *thread distribution* for a performance wise nested parallelism exploitation. As a natural evolution of the ideas previously presented [7, 8], the thread grouping mechanism is

studied and organized as a proposal to extend the OpenMP language. Different constructs are defined to allow programmers specify the most appropriate thread distribution between the levels of parallelism. An optimal algorithm is presented to compute the thread distribution. All these works rely on some effort coming from the programmer, as they require information supplied by programmers in order to compute the thread distribution. Typically, the supplied information is based on two parameters: the number of parallel branches in the outermost 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 being used in each branch or other specific characteristic of the application (number of "elements" treated by a computational branch, number of iterations...).

Thread distribution has been showed to be necessary and critical for achieving performance 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 so, based on the previous works of Duran et al. [13, 14]. Those works introduce the facility of sampling the execution time for each branch in the outermost level of parallelism. This measurement can be used as an approximation of the computational weight in the overall nest of parallelism. It is possible to obtain a suitable thread distribution based on the sampling.

## 6 Conclusions and Future work

This paper has explored the viability, in the context of OpenMP nested parallelism, of having 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 spent by each group doing useful work. Based on those 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 of a specific runtime implementation.

Results has shown our approach is able to get better performance than algorithms handcrafted by the application programmer. When applications work well with uniform distributions the mechanism has a low overhead which makes it well suited as a default option for nested parallelism.

Results has also shown 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 in the sampled times differences) a runtime library that uses the techniques described in the paper becomes more portable, from a performance point of view, across different platforms.

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

## References

- [1] 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 , Proceedings on the 3rd European Workshop on OpenMP (EWOMP2001).
- [2] Philippe Kloos, Fabrice Mathey, Philippe Blaise, OpenMP and MPI programming with a CG algorithm , Proceedings on the 3rd European Workshop on OpenMP (EWOMP2001).
- [3] D. May, From a Vector Computer to an SMP-Cluster - Hybrid Parallelization of the CFD Code PANTA , Proceedings on the 2nd European Workshop on OpenMP (EWOMP2001).
- [4] F. Mathey, P. Kloos, S. Blaise, OpenMP Optimisation of a Parallel MPI CFD Code , Proceedings on the 2nd European Workshop on OpenMP (EWOMP2000).
- [5] Lorna Smith, EPCC Development and Performance of a Hybrid OpenMP/MPI Quantum Monte Carlo Code , Proceedings of the 1st European Workshop on OpenMP (EWOMP99).
- [6] P. Lanucara and S. Rovida Conjugate-Gradient Algorithms: an MPI-OpenMP Implementation on Distributed Shared Memory Systems , Proceedings on the 1st European Workshop on OpenMP (EWOMP99).
- [7] E. Ayguadé, X. Martorell, J. Labarta, M. González and N. Navarro, Exploiting Multiple Levels of Parallelism in Shared Memory Multiprocessors: a Case Study. International Conference on Parallel Processing (ICPP99)
- [8] R. Blikberg, T. Sorevik, Nested Parallelism: Allocation of Processors to Tasks and OpenMP Implementation , Proceedings on the 2nd European Workshop on OpenMP (EWOMP2000).
- [9] M. Gonzalez, E. Ayguade, J. Labarta, X. Martorell, N. Navarro and J. Oliver. "NanosCompiler: A Research Platform for OpenMP Extensions", In First European Workshop on OpenMP, Lund (Sweden), October 1999
- [10] M. González, J. Oliver, X. Martorell, E. Ayguadé, J. Labarta and N. Navarro. OpenMP Extensions for Thread Groups and Their Runtime Support , In Workshop on Languages and Compilers for Parallel (LCPC00).
- [11] R. Blikberg, Parallelizing AMRCLAW by nesting techniques , Proceedings on the 4th European Workshop on OpenMP (EWOMP2002).
- [12] E. Ayguade, M. Gonzalez, X. Martorell and G. Jost, Employing Nested OpenMP for the Parallelization of Multi-Zone Computational Fluid Dynamics Applications, International Parallel and Distributed Process Symposium (IPDPS2004)

- [13] J. Corbalán, A. Duran, J. Labarta, Dynamic Load Balancing of MPI+OpenMP Applications, International Conference on Parallel Processing, (ICPP 2004)
- [14] A. Duran, R. Silvera, J. Corbalán, J. Labarta, Runtime Adjustment of Parallel Nested Loops, Workshop on OpenMP Applications and Tools (WOMPAT 2004)
- [15] 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", Proceedings of the 13th. ACM International Conference on Supercomputing (ICS99)
- [16] X. Martorell, E. Ayguadé, N. Navarro and J. Labarta, "A Library Implementation of the Nano-Threads Programming Model", Proceedings of the 2nd. Europar Conference 1996.
- [17] R.F. Van Der Wijngaart, H. Jin, NAS Parallel Benchmarks, Multi-Zone Versions, NAS Technical Report NAS-03-010, NASA Ames Research Center, Moffett Field, CA, 2003
- [18] "Fortran Language Specification , v2.0", <http://www.openmp.org>
- [19] M. Gonzalez, E. Ayguade, X. Martorell, J. Labarta and Phu V. Luong "Dual-level Parallelism Exploitation with OpenMP in Coastal Ocean Circulation Modeling", International Workshop on OpenMP: Experiences and Implementations (WOMPEI 2002)
- [20] J.Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server", Proceedings of the 24th. Annual International Symposium on Computer Architecture, (ISCA97).
- [21] IBM. POWER4 System Microarchitecture. <http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html>. 2001. October.