Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study

Eduard Ayguadé, Xavier Martorell, Jesús Labarta,
Marc Gonzàlez and Nacho Navarro
European Center for Parallelism of Barcelona, Computer Architecture Department (UPC),
cr. Jordi Girona 1-3, Mòdul D6, 08034 - Barcelona, Spain

Abstract

Most current shared–memory parallel programming environments are based on thread packages that allow the exploitation of a single level of parallelism. These thread packages do not enable the spawning of new parallelism from a previously activated parallel region. Current initiatives (like OpenMP) include in their definition the exploitation of multiple levels of parallelism through the nesting of parallel constructs. This paper analyzes the requirements towards an efficient multi–level parallelization and reports some conclusions gathered from the experience in the parallelization of two benchmark applications. The underlying system is based on: i) an OpenMP compiler which accepts some extensions to the original definition and ii) a user-level threads library that supports the exploitation of both fine–grain and multi–level parallelism.

1. Introduction

Parallel processing is being accepted by the computer industry as the path to increase the computational power of low–end workstations and even personal computers. Parallel architectures, ranging from multiprocessor workstations (with 2 to 4 processors) to medium scale shared–memory systems (up to 64 processors) are becoming more and more affordable and common. However, making these parallel machines truly usable requires easy–to–understand and portable programming models that allow the exploitation of parallelism out of applications written in standard high–level languages. They usually offer new mechanisms or extensions to the language to express the available parallelism of the application.

These extensions are usually offered by means of high–level directives and language constructs (the effort done within the OpenMP initiative [13] or the HPF High Performance Fortran Forum [9]) or by a set of services offered by a user–level thread package. Programming models in the first group offer a loosely synchronous programming model in which parallel jobs can be executed fully in parallel and synchronize at global points (by means of barriers or critical sections). Services included in most user–level thread packages allow a more general exploitation of parallelism (either at subroutine call level and at loop level) but at the expenses of higher programming effort.

Most current systems (compilers and run–time threads support) are based upon the exploitation of a single level of parallelism around loops (for example, the current version of the SGI MP library, the SUIF compiler infrastructure [6] or the MOERAE portable thread–based interface for the Polaris compiler [8]). Exploiting a single level of parallelism means that there is a single thread (master) that produces work for other processors (slaves). Once parallelism is activated, new opportunities for parallel work creation are ignored by the execution environment. Exploiting this parallelism may incur in low performance returns as one increases the number of processors to run the application.

Multi–level parallelism enables the generation of work from different simultaneously executing threads. Once parallelism is activated, new opportunities for parallel work creation result in the generation of work for all or a restricted set of processors. We believe that multi–level parallelization will play an important role in new scalable programming and execution models. Nested parallelism may provide further opportunities for work distribution, both around loops and sections; however, new issues may arise in order to attain high performance. OpenMP [13], jointly defined by a group of major computer hardware and software vendors, includes in its definition the exploitation of multi–level parallelism through the nesting of parallel constructs.

Previous work on supporting multi–level parallelism focused on providing some kind of coordination support to allow the interaction of a set of program modules in the framework of data parallel programs for distributed memory architectures. Some of them combine the use of two programming models and interfaces. For example, [3] proposes a library–based approach that provides a set of functions for coupling multiple HPF tasks to form task–parallel computations. Other alternatives [2, 5, 16] proposed a small set of Fortran directives to integrate task and data parallelism parallelism also in an HPF framework.

The Illinois–Intel Multithreading library [4] targets shared–memory systems. It also supports multiple levels of general (unstructured) parallelism. Application tasks are inserted into work queues before execution, allowing several task descriptions to be active at the same time. Kuck and Associates, Inc. has made proposals to OpenMP to support multi-level parallelism through the WorkQueue mechanism [10], in which work can be created dynamically, even recursively, and put into queues. Within the WorkQueue model, nested queuing permits a hierarchy of queues to arise, mirroring recursion and linked data structures. These proposals offer multiple levels of parallelism but do not support the logical clustering of processors in the multilevel structure, which may lead to better work distribution and data locality exploitation.

The approach presented in this paper takes Fortran applications fully annotated with directives to be parallelized by the compiler targeted to shared–memory architectures. Some extensions have been included in the OpenMP definition to enable an efficient exploitation of multiple levels of parallelism in numerical applications. Although the compiler may identify additional parallelism in the application through data and control dependence analysis, this aspect is out of scope for this paper. From the analysis of the program, the compiler generates an intermediate representation of the parallel application taking the form of a *Hierarchical Task Graph* (HTG [15]). The HTG representation captures parallelism information at different levels of granularity. An efficient user–level threads library allows the compiler to map the parallelism structure of the application into a Fortran code with calls to the services offered by the library.

The rest of the paper is organized as follows: Section 2 briefly describes the applications used along the paper as a case study. Section 3 presents the OpenMP programming model used in our environment and the thread–level support currently provided by run–time libraries. Section 4 presents some extensions proposed towards a more flexible and efficient multi–level parallelization and the thread–level support that is required. Section 5 analyzes the multi–level parallelization for both applications (running on top of an SGI Origin2000 platform). Finally, Section 6 concludes the paper.

2. Two SPEC95FP applications

In this paper two different applications have been selected from the SPEC95 benchmark set: Hydro2D, which computes the movement of galactical jets using Navier Stokes' equations for a variable number of points in the space; and Turb3D, which simulates isotropic homogeneous turbulences in a cube with periodic boundary conditions.

2.1. Hydro2D

Hydro2D works primarily with four different two–dimensional matrices (`RO`, `EN`, `GR` and `GZ`) of 402x160 elements. The main work of the application is done in function *advnce*, called directly from a timestep loop. This loop is repeated 200 times when the reference input is used. Each call to this function performs two steps over these four matrices.

In the first step, these matrices are used to calculate three sets of new matrices. The latter ones are combined, producing four intermediate matrices, onto which the *fct* is computed. The results of the four *fct* are the new versions of matrices `RO`, `EN`, `GR` and `GZ`, which are then used to perform the second step. The structure of the second step is very similar to that of the first one, calculating the final values for the matrices.

The central part in Figure 1 shows the parallelism structure for one of the two steps in function *advnce*. First of all, the computation of the three sets of matrices from the primary matrices can be done in parallel (nodes 2, 3 and 4). The functions involved in these computations are *corix*, *stagx1* and *stagx2*, where x stands for f or g, depending on the step. They contain both section and loop–level parallelism inside, as shown in the right part of Figure 1. After that, functions *trans1* and *trans2* are invoked (nodes 5 and 6),

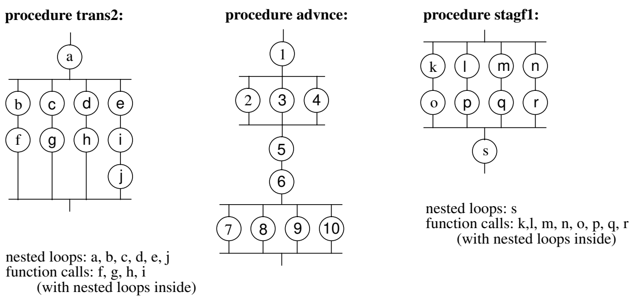


Figure 1. Parallelism structure for the SPEC95 Hydro2D application.

with the parallelism structure shown in the left part Figure 1. Again, they contain parallelism at the level of sections and loops. The computation of the four *fct* that follows *trans2* (nodes 7–10) can also be performed in parallel. Each *fct* contains function calls and nested loops with loop–level parallelism inside.

The application structure and data used in each node of the task graph enables the definition of processor groups. Each of the four sections that appear in different parts of the application is devoted to the computation of one of the above mentioned matrices. If one ensures that the same processors are always used for the execution of related sections, locality exploitation will be improved. This is the purpose of the extensions to OpenMP that we present in Section 4.

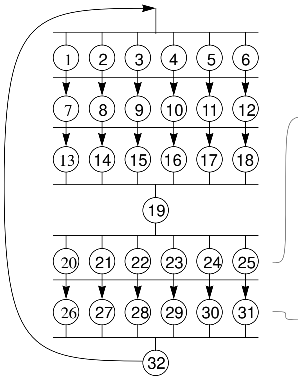
For this application, current parallelizing compilers, like PFA for the SGI Origin 2000 architecture [17] or SUIF [6] only detect the loop–level parallelism described above. In particular, the performance reported in the SPEC CFP95 summaries [18] for this program (using PFA) shows an speed–up of 3.93 and 4.28 for 8 and 16 processors[1], respectively.

2.2. Turb3D

The structure of the main loop in the application is shown in the left part of Figure 2: it consists of an iterative loop that alternates a series of Fourier–to–physical space and physical–to–Fourier space FFTs over six three–dimensional arrays (`U`, `V`, `W`, `OX`, `OY` and `OZ` of size 66x64x64) with the computation of a non–linear term on the physical space in between, and a time stepping phase before the end of each iteration. Each node is a loop or nested loop that iterates over one or several dimensions of the above mentioned arrays.

For instance, nodes 1, 7, 13, 20 and 26 perform different parts of the computation over matrix `U`. From the point of view of parallelization, nodes 7 and 26 may be parallelized in the loops that access the second dimension of array `U` while nodes 13 and 20 in the loops that access the third dimension of the same array. Node 1 can be parallelized in any of the loops that access the three different dimensions. These computations for each array can run completely in parallel. However, nodes 19 and 32 update some of the previously computed arrays with contributions from the rest of the arrays; for instance, node 19 updates arrays `OX`, `OY` and `OZ` with contributions from `U`, `V`, `W`, `OX`, `OY` and `OZ`. Deeper in the hierarchy of tasks (and through and intricate set of procedure invocations), nodes 7–

[1]These speed–ups are for a SGI Origin2000 system with R10k processors at 250 MHz, 4 Mb of secondary cache and 2 or 4 Gb of main memory, respectively.




```
ncpus = nthf_cpus_actual()
nsect = min(6, ncpus)
C$OMP PARALLEL GROUPS(nsect)
C$OMP SECTIONS
C$OMP SECTION ONTO(0)
C$OMP PARALLEL DO SCHEDULE(STATIC)
      DO  K=1,NZ
         CALL XYFFT(K,1,1,-1)
      ENDDO
...
C$OMP SECTION ONTO(5)
C$OMP PARALLEL DO SCHEDULE(STATIC)
      DO  K=1,NZ
         CALL XYFFT(K,1,6,-1)
      ENDDO
C$OMP END SECTIONS
...
C$OMP SECTIONS GROUPS(nsect)
C$OMP SECTION ONTO(0)
C$OMP PARALLEL DO SCHEDULE(STATIC)
      DO  J=1,NY
         CALL ZFFT(J,1,1,-1)
      ENDDO
...
C$OMP SECTION ONTO(5)
C$OMP PARALLEL DO SCHEDULE(STATIC)
      DO  J=1,NY
         CALL ZFFT(J,1,6,-1)
      ENDDO
C$OMP END SECTIONS
C$OMP END PARALLEL
```

Figure 2. Parallelism structure for the SPEC95 Turb3D application.

18 and 20–31 contain calls to direct and reverse two–dimensional FFTs.

The application presents multiple levels of parallelism. On the one side, the application offers parallelism at the level of sections that perform the same computation over 6 different arrays. On the other side, each section reveals two nested levels of loop parallelism (through several procedure invocations) and other nodes reveal loop–level parallelism. Some compilers, like PFA only detect the innermost level of parallel loops; this parallelization strategy does not report any speed–up, even for a small number of processors. In particular, the performance reported in the SPEC CFP95 summaries shows an slow–down of 0.8 and 0.75 for 8 and 16 processors[1], respectively. Other compilers, like SUIF are able to detect the outermost level of parallel loops; as we will report at the end of the next section, this parallelization strategy performs much better than the previous one.

3. OpenMP and thread–level support

The programming model used in this paper is based on OpenMP [13], the application program interface proposed to offer a programming model for portable parallel programming across shared memory architectures from different vendors. Fortran directives are translated by a compiler to code (based on a highly optimized thread interface) directly injected into the high–level Fortran code.

OpenMP offers a set of parallel, work–sharing and synchronization constructs to specify the parallelism structure of the application. It allows the definition of multi–level parallelism through the nesting of `PARALLEL` constructs. However, most current execution environments serialize inner parallel constructs because the supporting threads implementation does not support nested parallelism. For instance, the current implementation of the SGI MP library provides a very efficient mechanism based on a unique work descriptor, located at a fixed memory location, from where all processors determine the work to be executed. The descriptor contains, among other information, the pointer to the procedure that encapsulates the work to be done. This descriptor cannot be reused until all processors finish the work assigned.

In OpenMP, the static iteration scheduling policies specified in the parallel `DO` work–sharing directive do not assume any specific assignment of chunks of iterations to processors. In general, chunks are assigned to processors following their lexicographical order, starting from the chunk that contains the lower bound of the iteration space which is assigned to the first processor in the group of processors involved in the parallel execution. This assignment is a consequence of the unique work descriptor mechanism used to generate work. The descriptor contains the lower and upper bounds and step of the whole iteration space; each thread determines from its own thread identifier the chunk or chunks of iterations that has to execute. Similarly, in the `SECTIONS` work–sharing construct, the assignment of code segments parceled out by each `SECTION` directive is not predefined by the OpenMP model. However, its usual conversion to a parallel loop that conditionally branches to each part also establishes a default lexicographical order.

In order to conclude this section, we summarize the performance results for both applications when a single level of loop parallelism is exploited; Section 5 performs the complete evaluation for several parallelization strategies. In order to generate these results, we have generated an OpenMP version of the parallelization strategies proposed by the SUIF compiler, as included in the SUIF–SPEC95 distribution [18]. The strategy corresponds to the parallelization of the outermost loops. Figure 3 shows the speed–up obtained for this parallelization. Notice that in both cases this parallelization provides a relatively good performance up to a certain number of processors (between 16 and 32). Since the parallelization efficiency decreases as the number of processors is increased (diminishing returns are obtained as the number of processors is increased), one may argue if using a small degree of task parallelism (if exists) together with the loop parallelism above would contribute to get higher returns when the number of processors is increased.

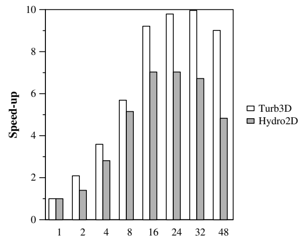


Figure 3. Speed–up for the loop parallelization in Hydro2D and Turb3D.

4. Efficient multi–level parallelization

In this section we describe the set of extensions to the OpenMP programming model oriented towards the definition of processor groups. In addition to that, we will also analyze the functionalities needed at the thread–level library.

4.1 Extensions to OpenMP

A group of processors is defined by a 'master' thread and a number of 'slave' threads. The definition of groups may be originated in any parallel construct. Once defined, work–sharing constructs inside the parallel construct will assign work to the master threads (instead of assigning the work to all the threads available). The slave threads will cooperate with the master in the exploitation of any additional parallelism inside these work–sharing constructs.

The extensions proposed allow the definition of the groups (i.e. the number of master threads and the number of slave threads assigned to each master). Once defined, other extensions allow the particular assignment of work to the groups. This allows the user to control the allocation of work and may result in a more efficient exploitation of multi–level parallelism; the appropriate assignment of work to groups may improve data locality and reduce load unbalance.

4.1.1 `GROUPS` clause

The `GROUPS` clause can be applied to any parallel construct. It establishes the groups of processors that will execute any work–sharing construct inside and nested parallel constructs:

```
C$OMP PARALLEL [DO|SECTIONS] [GROUPS(gdef[,gdef])]
  ...
C$OMP END PARALLEL [DO|SECTIONS]
```

where each group definer `gdef` has the following form:

```
[name:]ncpus
```

The `name` attribute is optional and is used to identify the group. The `ncpus` attribute is used to determine, from the number of currently available processors, the number of processors that will be assigned to the group. By default, groups are numbered from 0 to an upper value; this upper value is the number of groups defined within the clause `GROUPS` minus one.

A shortcut is available to specify the simplest group definition: `GROUPS(number)`. In this case, the user specifies the definition of `number` groups, each one receiving the same number of processors.

For instance, assume that the following definition of groups is provided in a `PARALLEL` construct: `GROUPS(a:2,b:3,one:1,two:2)`. In this case, four groups are set. Processors 0–1 would constitute the first group (`a`), processors 2–4 would constitute the second group (`b`), processor 5 would constitute the third group (`one`) and processors 6–7 would constitute the fourth group (`two`).

If the number of processors available at the time of reaching the parallel construct is different than the sum of processors specified in the clause, the numbers specified are considered as proportions; the runtime system has to be able to distribute (in the more fair way and with minimum overhead) the total number of processors according to these proportions.

4.1.2 `ONTO` clause

The default assignment of iterations in a `DO` work–sharing construct or individual sections in a `SECTIONS` work–sharing construct to processors can be changed by using the `ONTO` clause.

```
C$OMP DO [ONTO(expr)]
```

When this clause is used, `expr` specifies the group of processors that will execute a particular chunk of iterations (in fact, only the master of each group will execute the work). If the expression contains the loop control variable, then the chunk number (numbered starting at 0) is used to perform the computation; otherwise, all chunks are assigned to the same processor. In all group computations, a 'modulo the number of active groups' operation is applied. If not specified, the default clause is `ONTO(i)`, being `i` the loop control variable of the parallel loop.

For example, assume the previous definition of groups and the following `ONTO` clause:

```
C$OMP DO SCHEDULE(STATIC,4) ONTO(2*i)
        do i = 1, 1000
           ...
        enddo
C$OMP END DO
```

In this case, only groups `a` and `one` will receive work from this work–sharing construct. In particular, the master of the first group (processor 0) will execute iterations 1:4,9:12, ... and the master of the third group (processor 5) will execute iterations 5:8,13:16, ...

For example, for the same definition of groups, an `ONTO(2*k)` clause would specify that the master processor of the group 2*k (modulo the number of active groups) would execute all the iterations of the loop.

For the `SECTIONS` work–sharing contruct, the `ONTO(expr)` clause is attached to each `SECTION` directive to specify the group that would execute each section. Each expression `expr` can be different and is used to compute the group that will execute the statements parceled out by the corresponding `SECTION` directive. If the `ONTO` clause is not specified the compiler will assume an assignment following the lexicographical order of the sections.

For instance, when defining a `SECTIONS` work–sharing construct with four `SECTION` inside, the programmer could specify that following clauses: `ONTO(a)` in the first section, `ONTO(one)` in the second section, `ONTO(b)` in the third section and `ONTO(two)` in the fourth section. In this case, processors 0, 5, 2 and 6 would execute the code parceled out by the sections, respectively.

For example, the right part of Figure 2 shows how these clauses are used to specify the multi–level parallelization strategy and processor groups described in Section 2 for the Turb3D application. Notice that the user is defining `nsect` groups in the outer parallel construct. This value is computed as the minimum between 6 and the number of currently available processors (returned by a service of the threads library). The master of each group will execute one of the sections, according to the number provided in the `ONTO` clause. Each master will encounter the inner loop–level parallel construct and spawn work for the processors available in his team (for instance `ncpus` divided by `nsect`, as indicated by the threads library through a specific call).

4.2. Supportive user–level threads library

In this section we briefly describe the main services required from the user–level threads library. Although the library has been defined to directly support the execution of the parallelism expressed by means of a *Hierarchical Task Graph* [15], this paper only focuses on the functionalities needed to support the OpenMP programming model.

Each parallel construct expressed inside the application is transformed in such a way that the original code is encapsulated in a function. In its place, the compiler inserts the specific code to generate the parallelism. Generating parallelism consists on building the description of the work to be executed and calling the user-level threads library to supply work to the participating processors.

The threads package has been designed to provide different mechanisms to spawn parallelism. Depending on the hierarchy level in which the application is running, the requirements for creating work are different. When spawning the deepest (fine grain) level of parallelism, the application only creates a work descriptor and supplies it to the participating processors. The mechanism is implemented as efficiently as the one provided by most current systems [11]. However, the design allows processors spawning parallelism to generate more than one descriptor and supply them to the slaves before they terminate with previous parallelism, thus supporting multiple levels of parallelism.

When the application knows that it is spawning coarse grain parallelism, not at the deepest level, it can pay the cost of supporting nested parallelism. Higher levels of parallelism, containing other parallel regions, are generated using a more costly interface that provides threads (called nanothreads) with a stack [12]. Owning a stack is necessary for the higher levels of parallelism to spawn an inner level, because the stack is used to maintain the context of the higher levels, along with the data structures needed for joining the parallelism, while executing the inner one. In addition, nanothreads have been designed so that the compiler can specify precedence relations among them.

An important aspect of the library design is the support for processor groups. This requires extra functionalities to set the current groups definition (used when clause `GROUPS` is found in a `PARALLEL` construct) and to get to actual groups composition (used to decide the subset of processors receiving work in each work sharing construct). All this information is maintained by the library in order to allow orphaned directives.

Processors waiting for work search first for work descriptors and then for nanothreads. Each entity is managed in a specific way: nanothreads are enqueued in ready queues; work descriptors are supplied to lists using a more efficient mechanism. Two different structures co–exist for holding these entities: global (accessed by all the processors) and local (per processor). They enable the compiler to decide, for each work–sharing construct, if locality or load balancing are the issues that need to be considered in its parallel execution. Local structures are needed to allow the execution model based on processor groups.

5. Experimental evaluation

This section evaluates the performance obtained for the two applications described in Section 2. The experimental framework includes the following components: i) an OpenMP compiler [1] developed on top of Parafrase–2 [14]; ii) the user–level threads library NthLib [12] developed on top of Quick–Threads [7]; iii) and a Silicon Graphics Origin2000 system with 64 R10k processors, running at 250 MHz with 4 Mb of secondary cache each. For all compilations we use the native `f77` compiler with flags set to `-64 -Ofast=ip27 -LNO:prefetch_ahead=1:auto_dist=on`.

5.1. Hydro2D benchmark

Two different versions of this benchmark have been executed using the reference input file, as provided by the SPEC definition. The sequential execution time is 154.7 seconds. The first one implements a single level parallelization strategy around loops. Figure 4 shows the speed–up obtained when this parallelization strategy is executed on top of NthLib, using up to 64 processors. Notice that the performance grows up to 16 processors and then declines. This is due to the fact that the parallelism available in the loops is not enough for such large number of processors.

The second parallel version implements the multi–level parallelization described in Section 2. In particular, one of the levels of parallelism has been intentionally omitted because it introduces an important loss of locality: nodes 2–4 in Figure 1 are executed sequentially. In addition to that, we define 4 processor groups. For instance, a group of processors is responsible for the execution of nodes b, f, k, o and 7; loop–level parallelism inside these nodes is executed only by the set of processors assigned to the group. All the processors collaborate in the execution of nodes where a single level of parallelism is available (like nodes a, 1, and s). Notice that this parallelization strategy returns better performance than the loop–level strategy when more than 16 processors are used, reaching a maximum speed–up of 9.1 with 48 processors.

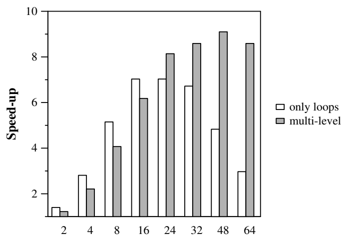


Figure 4. Speed–up for the Hydro2D program.

Figure 5 shows the behavior of the parallelization strategies in terms of secondary cache misses, which in a NUMA architecture may require access to remote memory, and external invalidations that occur per processor, as reported by the *perfex* tool. Notice that in general, the numbers for the multi–level strategy are higher that for the loop–level strategy. This is due to the data movement that happens when flowing from parts of the application where groups of processors concentrate in some nodes of the task graph to parts of the application where all processors collaborate in the execution of the same loops. This justifies the low speed–up reported for the multi–level strategy when less than 16 processors are used. When more than 16 processors are used, the additional overhead of the data movement is counteracted by the gain produced by a better distribution of work among groups of processors. Using more than 48 processors produces a decrease of the speed-up because of the poor distribution of work among processors within each group.

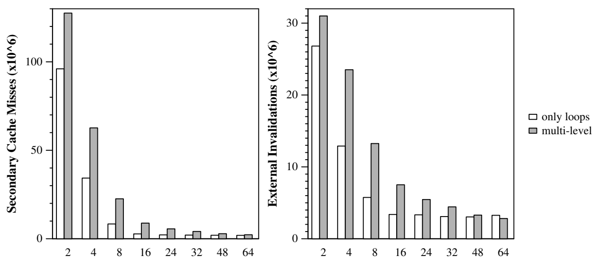


Figure 5. Average number of secondary cache misses and external invalidations (per processor) in Hydro2D.

5.2. Turb3D benchmark

For this application, the following parallelization strategies have been analyzed. The first one is the proposed by the automatic parallelizer PFA for the SGI Origin2000 architecture. In this case, the compiler suggests the parallelization of the innermost loops in the routines that compute the two–dimensional FFT; this parallelization will be referred in the rest of this section as 'inner'. The second parallelization analyzed is the one suggested by the SUIF compiler, consisting on the parallelization of the outer loops that compute 2D FFTs over different planes of three–dimensional matrices; this parallelization strategy will be referred as 'outer'. Finally, the third and fourth strategies studied are multi–level parallelizations where we parallelize the section–level parallelism described in section 2 and either the inner or the outer loops mentioned above; these two strategies will be named 'mlv–inner' and 'mlv–outer'.

We have conducted a set of experiments in order to analyze the behavior of the different parallelization strategies and number of processors. For the multi–level parallelizations, we first try to assign as many processors as possible to the outermost level (sections level) and the rest of processors to the innermost one; in this case, groups of processors concentrate on the execution of each individual section. Figure 6 summarizes the performance results.

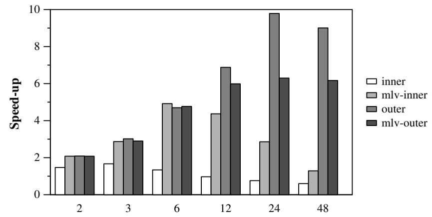


Figure 6. Speed–up for the Turb3D program.

First of all, notice that the 'inner' parallelization strategy fails for this application and that increasing the number of processors to exploit this level of parallelism actually results in a reduction of parallel performance. This is due to the small size of the two–dimensional FFTs (64x64) that are computed at the innermost level. Second, the 'mlv–inner' parallelization performs better than the 'inner' parallelization. The performance for this strategy increases when using up to six processors; this corresponds to exploiting a single level of parallelism: sections for nodes 1–18 and 20–31, and loops for nodes 19 and 32. When more than 6 processors are used (i.e. when processors are allocated to the execution of the innermost FFTs in nodes 1–18 and 20–31), the performance drops.

For this application, the 'outer' parallelization always outperforms the 'mlv–outer' parallelization. This is due to the additional data movement among processors that occur in the multi–level parallelization. This data movement can be observed in the amount of secondary cache misses and external invalidations that each processor suffers:

- The 'outer' strategy parallelizes the loops that traverse the second dimension of the arrays in nodes 1–12 and 26–32; for the rest of nodes, the loops that traverse the third dimension are parallelized. This means that, at each iteration of the outer loop, two transpositions of the six arrays occur.

- With up to six processors, in the 'mlv–outer' parallelization there is no data movement during the execution of any sequence of nodes that use the same array (for example nodes 1, 7 an 13). However, the averaging nodes (19 and 32) imply high data movement.

- When more than six processors are used in the 'mlv–outer' strategy, the additional transpositions within each group of processors introduce additional overhead.

Figure 7 shows the number of secondary cache misses and external invalidations that occur in the application per processor, as reported by the *perfex* tool. Notice that in general, the number of external invalidations for the 'mlv–outer' strategy are higher that for the 'outer' strategy. Also, when more than 6 processors are used in the application, the number of secondary cache misses of the 'mlv–outer' strategy is higher than 'outer'; this corresponds to the point where the performance of the 'mlv–outer' strategy saturates in Figure 6

In order to analyze the influence of the averaging computations in the 'mlv–outer' parallelization, we have generated a synthetic benchmark from the original SPEC application so that nodes 19 and 32 are executed at each iteration, once every 2, 5, 10, 25 or 50 iterations, or never executed. In this case we are reducing the data movement overhead due to them. Figures 8 and 9 show the speed–up and number of secondary cache misses and external invalidations, respectively, for different values of the `NAVG` variable. Notice that for this application, it would be enough to keep data locality inside the groups during two consecutive iterations to have a higher speed–up in the 'mlv–outer' parallelization. In particular, notice that in the 'never' situation the number of secondary cache misses and invalidations is always smaller in the 'mlv–outer' version than in the 'outer' version. In the '`NAVG=2`' situation, this is also true when using less than 24 processors; otherwise, the 'outer' version outperforms 'mlv–outer'.

Summary

In the Hydro2D benchmark, multi–level parallelism boosts the performance of the parallel execution; the comparison with the traditional loop–level parallelism gives a 30% improvement on 32 processors. Although

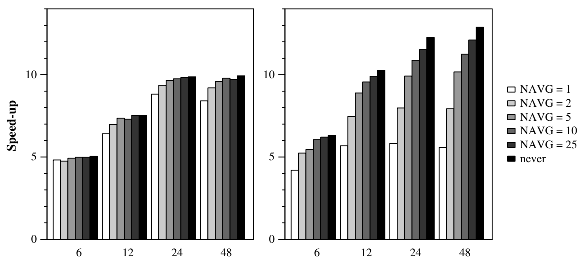


Figure 8. Speed–up for the synthetic Turb3D program for the 'outer' (left) and 'mlv–outer' (right) strategies.

the memory behavior is worse, the better work distribution that is achieved in the multi-level version raises the performance.

In the Turb3D benchmark the multi–level strategy does not improve the performance. We have generated a synthetic version in order to analyze the influence of having parts in the application that benefit from multiple levels of parallelism and parts that only have one level of loop parallelism. For instance, for 12 processors, the classical outer–level loop parallelizations reports an speedup of 6.41 while a multi–level parallelization obtains 5.68 when the degree of interaction is high (each iteration of the outer loop). When this degree of interaction decreases, the multi–level parallelization starts to outperform the classical one. When the interaction is performed half the number of times in the original program, the performance increases by 6%; when this degree of interaction is reduced to a minimum, the performance increases up to 36%.

Grouping of processors has also been proposed as an extension in the context of OpenMP. In this case we have proposed a clause in the directive to specify these groups. If no groups are specified, any additional parallelism that is created inside an active parallel region involves all the processors allocated to the application. For this application, the behavior of a 'mlv–outer'

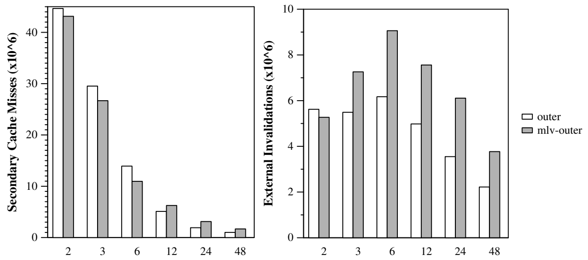


Figure 7. Average number of secondary cache misses and external invalidations (per processor) for the 'outer' and 'mlv–outer' parallelizations in Turb3D.

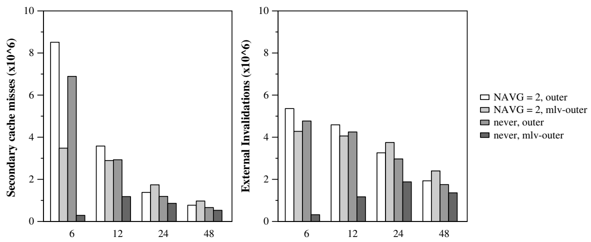


Figure 9. Average number of secondary cache misses and external invalidations (per processor) for the synthetic Turb3D program for the 'outer' (left) and 'mlv–outer' (right) strategies.

strategy without grouping is very close to the behavior of the 'outer' one; in this case, the outer level is created but then all processors are involved in the execution of the inner level.

6. Conclusions

In this paper we have presented some extensions to the current definition of OpenMP oriented towards the definition of processor groups when multiple levels of parallelism exist in the application. Most current compilers and run–time systems only support the exploitation of single–level parallelism around loops. In order to exploit multiple levels of parallelism, several programming models are combined (e.g. message passing and OpenMP). We believe that a single programming paradigm should be used and should provide similar performance. The paper also discuses the requirements and functionalities needed in the threads library.

The experimental evaluation shows that multi–level parallelism may play its role in increasing performance. We have analyzed a set of parallelization strategies for two SPEC95FP applications. The purpose of the evaluation has been to figure out situations in which multiple levels of parallelism are worth to be exploited. The two applications suffer from frequent patterns of interaction between parts of the application where multiple or single levels of parallelism exist. Although this does not benefit the exploitation of the multiple levels of parallelism, the results are encouraging and show some of the key factors that need to be addressed.

Acknowledgments

This research has been supported by the Ministry of Education of Spain under contracts TIC98-511 and TIC97-1445CE, the ESPRIT project 21907 NANOS (www.ac.upc.es/nanos) and the CEPBA (European Center for Parallelism of Barcelona).

References

[1] E. Ayguadé, X. Martorell, J. Labarta, M. Gonzalez and J.I. Navarro. Exploiting Parallelism Through Directives on the Nano-Threads Programming Model. In *10th Workshop on Languages and Compilers for Parallel Computing*, Minneapolis (USA), August 1997.

[2] I. Foster, B. Avalani, A. Choudhary and M. Xu. A Compilation System that Integrates High Performance Fortran and Fortran M. In *Scalable High Performance Computing Conference*, Knoxville (TN), May 1994.

[3] I. Foster, D.R. Kohr, R. Krishnaiyer and A. Choudhary. Double Standards: Bringing Task Parallelism to HPF Via the Message Passing Interface. In *Supercomputing'96*, November 1996.

[4] M. Girkar, M. R. Haghighat, P. Grey, H. Saito, N. Stavrakos and C.D. Polychronopoulos. Illinois-Intel Multithreading Library: Multithreading Support for Intel Architecture–based Multiprocessor Systems. *Intel Technology Journal*, Q1 issue, February 1998.

[5] T. Gross, D. O'Halloran and J. Subhlok. Task Parallelism in a High Performance Fortran Framework. *IEEE Parallel and Distributed Technology*, 2(3), Fall 1994.

[6] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, December 1996.

[7] D. Keppel. Tools and Techniques for Building Fast Portable Threads Packages. Tech. Rep. UWCSE 93-05-06, Univ. of Washington, 1993.

[8] S.W. Kim, M. Voss, and R. Eigenmann. MOERAE: Portable Interface between a Parallelizing Compiler and Shared-Memory Multiprocessor Architectures. Tech. Rep. Purdue Univ. ECE-HPCLab-98210.

[9] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele and M.E. Zosel. The High Performance Fortran Handbook. *Scientific Programming*, 1994.

[10] Kuck and Associates, Inc. WorkQueue Parallelism Model. http://www.kai.com, Fall 1998.

[11] X. Martorell, E. Ayguadé, J.I. Navarro, J. Corbalán, M. González and J. Labarta. Thread Fork/join Techniques for Multi–level Parallelism Exploitation in NUMA Multiprocessors. In *13th Int. Conference on Supercomputing*, Rhodes (Greece), 1999.

[12] X. Martorell, J. Labarta, J.I. Navarro, and E. Ayguadé. A library implementation of the nano-threads programming model. In *Euro-Par'96*, 1996.

[13] OpenMP Organization. Fortran Language Specification, v. 1.0, www.openmp.org, October 1997.

[14] C.D. Polychronopoulos, M. Girkar, M.R. Haghighat, C.L. Lee, B. Leung, and D. Schouten. Parafrase–2: An environment for parallelizing, partitioning, and scheduling programs on multiprocessors. *International Journal of High Speed Computing*, 1(1), 1989.

[15] C.D. Polychronopoulos. Nano-threads: Compiler driven multithreading. In *4th Int. Workshop on Compilers for Parallel Computing*, Delft (The Netherlands), December 1993.

[16] S. Ramaswamy. Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Computations. PhD Thesis. University of Illinois at Urbana–Champaign, 1996.

[17] Silicon Graphics Computer Systems SGI. *MIPSpro Fortran 77 Programmer's Guide*, 1996.

[18] SPEC Organization. *The Standard Performance Evaluation Corporation*, www.spec.org.