Dual-Level Parallelism Exploitation with OpenMP in Coastal Ocean Circulation Modeling

Marc González[1], Eduard Ayguadé[1], Xavier Martorell[1], Jesús Labarta[1] and Phu V. Luong[2]

[1] European Center for Parallelism of Barcelona (CEPBA)
Technical University of Catalunya (UPC), Barcelona, Spain

[2] University of Texas
Engineer Research and Development Center
Major Shared Resource Center
Vicksburg, MS 39180.

Abstract. Two alternative dual-level parallel implementations of the Multiblock Grid Princeton Ocean Model (MGPOM) are compared in this paper. The first one combines the use of two programming paradigms: message passing with the Message Passing Interface (MPI) and shared memory with OpenMP (version called MPI-OpenMP); the second uses only OpenMP (version called OpenMP-Only). MGPOM is a multiblock grid code that enables the exploitation of two levels of parallelism.

The MPI-OpenMP implementation uses MPI to parallelize computations by assigning each grid block to a unique MPI process. Since not all grid blocks are of the same size, the workload between processes varies. OpenMP is used within each MPI process to improve load balance. The alternative OpenMP-Only implementation uses some extensions proposed to OpenMP that defines thread groups in order to efficiently exploit the available two levels of parallelism. These extensions are supported by a research OpenMP compiler named NanosCompiler.

Performance results of the two implementations from the MGPOM code on a 20-block grid for the Arabian Gulf simulation demonstrate the efficacy of the OpenMP-Only versions of the code. The simplicity of the OpenMP implementation as well as the possibility of using and simply defining policies to dynamically change the allocation of OpenMP threads to the two levels of parallelism is the main result of this study and suggests to consider this alternative for the parallelization of future applications.
Keywords: OpenMP and MPI implementations, multiple levels of parallelism, multiblock grid, coastal ocean circulation model.

1 Introduction

In recent years, OpenMP [1] has emerged as an industrial library for parallel programming in shared-memory computers. Parallel performance is achieved without significantly sacrificing execution time when it is ported across a range of shared-memory platforms. Moreover, its simplicity makes the conversion of a sequential code to a parallel code for improving performance much easier and without major code modifications.

One of the key features which is not currently exploited by most current commercial and experimental systems is the use of OpenMP for multiple levels of parallelism. In general, OpenMP can be used to exploit the multi-level parallelism in most scientific and engineering numerical applications. This technique, however, has not been fully applied to any of these applications because they achieve satisfactory speed-ups when executed in mid-size parallel platforms or because most current systems supporting OpenMP (compilers and associated thread--level layer) sequentialize nested parallel constructs. This has originated the current practice of exploiting multiple levels of parallelism through a combination of different programming models and interfaces, MPI [2] coupled with OpenMP for example. In such cases, MPI is usually used for communication at the outer levels between subdomains or between block grids (multiblock grid), while OpenMP is used to parallelize the inner levels within each subdomain or block.

In addition to the possibility of nesting parallel constructs, OpenMP offers the possibility of controlling the number of threads usable at each level of parallelism (clause `NUM_THREADS` available in OpenMP v2.0). Some extensions have been proposed to OpenMP in order to allow a cleaner and effective control over the work distribution when dealing with multiple levels of parallelism through the definition of thread groups [4]. This proposal is the one used to express the parallelism in the MGPOM application and will be discussed in Section 2. Other proposals consist in offering work queues and an interface for inserting application tasks before execution-for example, the Illinois--Intel Multithreading library [5] or the WorkQueue mechanism [6] proposed by KAI, in which work can be created dynamically, even recursively, and put into queues. The proposal used in this paper is simpler and allows finer control over the allocation of threads to the multiple levels of parallelism in the application.

In this study, OpenMP is used for both outer as well as inner levels. Details of the extensions to OpenMP are discussed in Section 2. Its application to the MGPOM on a 20-block grid of the Arabian Gulf simulation is presented in Section 3. Parallel performance results are presented in Section 4. The conclusions of this study are in Section 5.

2 NanosCompiler Extensions

The NanosCompiler [3] and runtime library are serving as a research platform for proposing and evaluating extensions to the OpenMP language definition. One of these extensions consists of the dynamic creation of thread groups and the definition of the actual composition of groups at runtime. Groups can be created to exploit both loop- and task-level parallelism.

In the fork/join execution model defined by OpenMP, a program begins execution as a single process or thread. This thread executes sequentially until a PARALLEL construct is found. At this time, the thread creates a team of threads and it becomes its master thread. All threads execute the statements enclosed lexically within the parallel constructs. Work-sharing constructs (DO, SECTIONS and SINGLE) are provided to divide the execution of the enclosed code region among the members of a team. All threads are independent and may synchronize at the end of each work-sharing construct or at specific points (specified by the BARRIER directive). Exclusive execution mode is also possible through the definition of CRITICAL regions.

The SECTIONS directive is a non-iterative work-sharing construct which specifies the enclosed sections of code (each one delimited by a SECTION directive) are divided among threads in the team. Each section becomes a task which is executed once by a thread in the team. The DO work-sharing construct is used to divide the iterations of a loop into a set of independent tasks, each one executing a chunk of consecutive iterations. Finally, the SINGLE work-sharing construct informs that only one thread in the team is going to execute the work.

In this study, a *group of threads* is composed of a subset of the total number of threads available in the team to run a parallel construct. The threads participating in a parallel construct are identified following the active numeration inside the current team (from 0 to omp_get_num_threads()-1). In a parallel construct, the programmer may define the number of groups and the composition of each group. When a thread in the current team encounters a parallel construct defining groups, the thread creates a new team and it becomes its master thread. The new team is composed of as many threads as groups are defined; the rest of the threads are reserved to support the execution of nested parallel constructs. In other words, the groups definition establishes the threads that are involved in the execution of the parallel construct and the allocation strategy or scenario for the inner levels of parallelism that might be spawned. When a member of this new team encounters another parallel construct (nested to the one that caused the group definition), it creates a new team and deploys its parallelism to the threads that compose its group.

The `GROUPS` clause allows the user to specify thread groups. It can only appear in a `PARALLEL` construct or combined `PARALLEL DO` and `PARALLEL SECTIONS` constructs.

```
C$OMP PARALLEL [DO|SECTIONS] [GROUPS(gspec)]
```

Different formats for the groups specifier `gspec` are allowed [4]. In this paper we only comment on the two more relevant. For additional details concerning alternative formats as well as implementation issues, please refer to this publication.

```
GROUPS(ngroups,weight)
```

In this case, the user specifies the number of groups (`ngroups`) and an integer vector (`weight`) indicating the relative amount of computation that each group has to perform. Vector `weight` is allocated by the user in the application address space and it has to be computed from information available within the application itself (for instace iteration space, computational complexity or even information collected at runtime). The runtime library determines, from this information, the composition of the groups. The algorithm assigns all the available threads to the groups and ensures that each group at least receives one thread. The main body of the algorithm is shown in Figure 1 (using Fortran90 syntax).

```
howmany(1:ngroups) = 1
do while (sum(howmany(1:ngroups)) .lt. nthreads)
   pos = maxloc(weight(1:ngroups)/
                    howmany(1:ngroups))
   howmany(pos(1)) = howmany(pos(1)) + 1
end do
masters(1) = 0
do i = 1, ngroups-1
   masters(i+1) = masters(i) + howmany(i)
end do
```

Fig. 1. Skeleton of the algorithm used to compute the composition of groups.

The library generates two internal vectors (`masters` and `howmany`). In this algorithm, `nthreads` is the number of threads that are available to spawn the parallelism in the parallel construct containing the group definition.

The most general format allows the specification of three parameters in the group definition:

```
GROUPS(ngroups, masters, howmany)
```

The first argument (`ngroups`) specifies the number of groups to be defined and consequently the number of threads in the team that is going to execute the parallel construct. The second argument (`masters`) is an integer vector with the identifiers (using the active numeration in the current team) of the threads that will compose the new team. Finally, the third argument (`howmany`) is an integer vector whose elements indicate the number of threads that will compose each group. The vectors have to be allocated in the memory space of the application, and their content and correctness have to be guaranteed by the programmer. Notice that this format must be used when the default mapping explained before does not provide the expected performance.

3 Application of NanosCompiler Extensions to MGPOM

MGPOM is a standard Fortran77 multiblock grid code. A parallel version of MGPOM uses MPI asynchronous sends and receives to exchange data between adjacent blocks at the interfaces. OpenMP has been used as a second level of parallelization within each MPI process to improve the load balance in the simulation of the Arabian Gulf [7]. This area is extended from 48 East to 58 East in longitude and from 23.5 North to 30.5 North in latitude (left part of Figure 2). The computational 20-block grid shown on the right part of Figure 2 (also used in [7]) is used in this study.

OpenMP with NanosCompiler Extensions is used to parallelize the serial version of the MGPOM code at the outer levels (block to block) as well as at the inner levels (within each block). The number of threads used to exploit the inner level of parallelism depends on the size of each grid block. Figure 3 shows the use of OpenMP directives and `GROUPS` construct implemented into the main program and a subroutine of the serial MGPOM code version. This figure shows a version in which the runtime library, using the default algorithm described in Section 2, determines the composition of the groups. The `GROUPS` clause has as input arguments the number of blocks (`maxb`) and a vector with the number of grid points in each block (`work`).

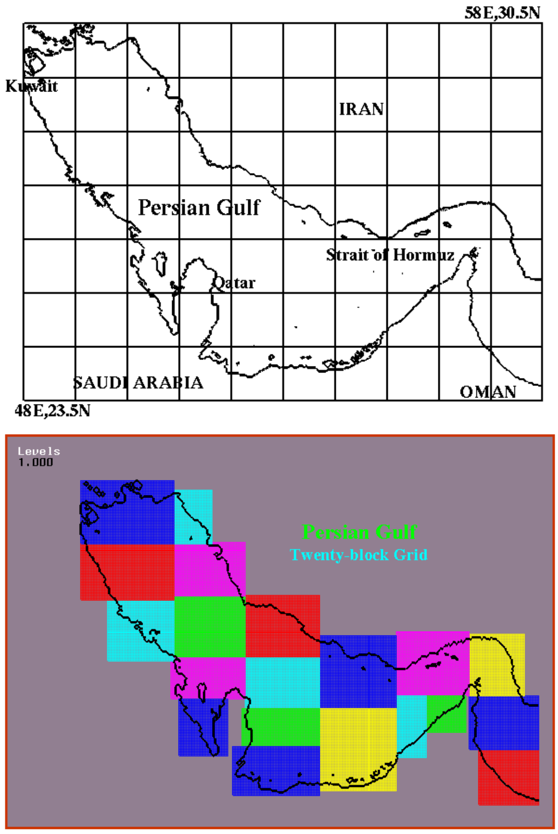

Fig. 2. The Persian Gulf coastline (left). The 20-block grid with blocks of different size (right).

Notice that the `GROUPS` clause is the only non-standard use of OpenMP. The exploitation of multiple levels of parallelism is achieved through the nesting of `PARALLEL DO` constructs.

```
        PROGRAM main
        ...
C$OMP PARALLEL DO PRIVATE(n) GROUPS(maxb, work)
        DO 414 n = 1, maxb
          ...
          IF (mode .ne. 2) THEN
            CALL baropg (drhox, drhoy, drx2d, dry2d, n)
            ...
          ENDIF
414   CONTINUE
        ...
        END

        SUBROUTINE baropg (drhox, drhoy, drx2d, dry2d, nb)
        ...
C$OMP PARALLEL DO PRIVATE (i,j,k)
        DO 200 j = 1, jm
          DO 200 k = 1 kb
            DO 200 I = 1, im
              rho(i,j,k,nb) = rho(i,j,k,nb) - rmean(i,j,k,nb)
200     CONTINUE
        ...
        RETURN
        END
        ...
```

Fig. 3. Excerpt of the OpenMP implementation.

Table 1 shows the number of grid points in each block for the 20-block grid. This is the composition of the `work` vector. The runtime library supporting the code generated by the NanosCompiler would generate the allocation of threads to groups shown in Tables 2 and 3 assuming 20 and 30 processors, respectively. `Who` and `Howmany` are the two internal vectors generated by the library with the master of each group (`who`) and the number of threads to be used in each group (`howmany`). With 20 processors, the critical path is determined by the largest block (i.e. block number 8), yields a theoretical speedup of 12.5. With 30 processors, the critical path is determined by the block with the largest ratio `size/howmany` (i.e. block number 17), with a theoretical speed-up of 19.4.

Table 1. Number of grid points in each block for the twenty-block case.

Block	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Size	1443	1710	1677	2150	700	989	2597
Block	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Size	2862	2142	1836	1428	1881	1862	2058
Block	**15**	**16**	**17**	**18**	**19**	**20**	-
Size	1470	1280	1848	2318	999	2623	-

Table 2. Default allocation of threads to groups with 20 processors.

Block	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Howmany	1	1	1	1	1	1	1	1	1	1
Who	0	1	2	3	4	5	6	7	8	9
Block	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Howmany	1	1	1	1	1	1	1	1	1	1
Who	10	11	12	13	14	15	16	17	18	19

Table 3. Default allocation of threads to groups with 30 processors.

Block	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Howmany	1	1	1	2	1	1	2	2	2	2
Who	0	1	2	3	5	6	7	9	11	13
Block	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Howmany	1	2	2	2	1	1	1	2	1	2
Who	15	16	18	20	22	23	24	25	27	28

According to the allocation of threads to groups shown in Table 3, the average work per thread has a large variance (from 1848 in block 17 to 700 in block 5). This variance results in a noticeable load imbalance. To reduce this load imbalance, several blocks could be gathered into a cluster such that the work distribution is equally divided among groups. To achieve this, the user can define its own composition of groups, as shown in Figure 4.

```
        PROGRAM main
        ...
        CALL compute_groups(work, maxb, who, howmany)
        ...
C$OMP PARALLEL DO PRIVATE(n) GROUPS(maxb, who, howmany)
        DO 414 n = 1, maxb
          ...
          IF (mode .ne. 2) THEN
            CALL baropg (drhox, drhoy, drx2d, dry2d, n)
            ...
          ENDIF
414     CONTINUE
        ...
        END
```

Fig. 4. Excerpt of the OpenMP implementation with user-defined groups.

User function `compute_groups` decides how many OpenMP threads are devoted to the execution of each block and which OpenMP thread is going to be the master in the exploitation of the inner level of parallelism inside each block. This function has as input arguments the number of blocks (`maxb`) and a vector with the number of grid points in the block (`work`). It returns two vectors with the master of each group (`who`) and the number of threads to be used in each group (`howmany`). This is the information that is later used in the `GROUPS` clause.

Table 4 shows the allocation of the threads to groups as well as the identities of blocks in a cluster. A cluster of two blocks with 20 processors is shown in this case. In this case, the work imbalance is noticeably reduced.

Table 4. Allocation of threads to groups with 20 processors after clustering.

Cluster	**1**	**2**	**3**	**4**	**5**
Blocks	1, 9	2, 13	3, 12	4, 11	5, 8
Howmany	2	2	2	2	2
Who	0	2	4	6	8
Cluster	**6**	**7**	**8**	**9**	**10**
Blocks	6, 20	7, 19	10, 17	14, 15	16, 18
Howmany	2	2	2	2	2
Who	10	12	14	16	18

4 Performance Results

In this section we evaluate the behaviour of two parallel versions of MGPOM. The `MPI-OpenMP` version exploits two levels of parallelism by combining the two programming paradigms: MPI to exploit the inter-block parallelism and OpenMP to exploit the intra-block parallelism. The `OpenMP-Only` version exploits the two-levels of parallelism using OpenMP and the extensions offered by the NanosCompiler and supporting OpenMP runtime system. For the compilation of the multilevel OpenMP version we use the NanosCompiler to translate from extended OpenMP Fortran77 to plain Fortran77 with calls to the supporting runtime library NthLib. We use the native `f77` compiler to generate code for an SGI Origin2000 system [8]. The flags are set to `-mips4 -64 -O2`. The experiments have been performed on system with 64 R10k processors, running at 250 MHz with 4 Mb of secondary cache each.

Table 5. Speed-up with respect to the sequential execution for the multi-level OpenMP and the mixed MPI/OpenMP versions.

	OpenMP-Only	**MPI-OpenMP**
10	9.6	
20	15.3	14.7
30	18.7	
40	23.9	22.2
50	26.7	
60	27.8	23.0

Table 5 shows the speed-up achieved by the two versions of the program. The MPI-OpenMP version uses the same number of threads per MPI process. For this reason performance numbers are available for only 20, 40 and 60 processors.

In order to understand these results, we have instrumented the application with the –P option of the NanosCompiler and generated a trace suitable for being analyzed with Paraver [9]. Each iteration of the application time step loop can be divided in 5 different parts: P1, P2, P3, P4 and P5. Basically, parts P1 and P4 perform independent computations over the blocks. However, parts P2, P3 and P5 are responsible for updating the boundaries of the blocks; these parts require communication in the MPI version and are the source of cache coherence overheads in the OpenMP version. Table 6 shows the average speed-up of each of these individual parts for the OpenMP-Only version.

Table 6. Speed-up (relative to the execution with 10 processors) of each part of the time step loop for the OpenMP-Only version.

	P1	P2	P3	P4	P5
10	1	1	1	1	1
20	1.8	1.2	1.3	1.8	1.2
30	2.2	1.6	1.8	2.3	1.4
40	3.2	1.7	1.9	3.5	1.5
50	4.1	2.0	2	4.5	1.7
60	4.9	2.1	2.1	5.3	1.8

Notice that parts P1 and P4 scale much better than the rest of the parts. The parallelization of parts P2, P3 and P5 (update of the boundaries) perform accesses to memory with no data locality with respect to the access pattern performed in parts P1 and P4. This causes a large number of secondary cache misses and cache invalidations, which add overhead to the overall parallel execution time. In addition to that, parts P2, P3 and P5 include a large number of small loops whose parallelization is not efficient, due to the size of the iteration space traversed and due to the barriers that have to be performed to enforce the correct update of the boundary elements.

In order to conclude this section, Table 7 shows the relative performance (w.r.t the execution with 20 processors of the OpenMP-only version in Table 5) for three OpenMP versions of the application: "no_groups" exploits two levels of parallelism but does not define groups (similar to using the NUM_THREADS clause), "groups" performs a homogeneous distribution of the available threads among the groups, and "weighted" performs a weighted distribution of the available threads among the groups. Notice that the performance of both "groups" and "weighted" versions is greater than "no_groups". Also, the "weighted" performs better due to load unbalance that exists among blocks. In summary, additional information is needed in the NUM_THREADS clause to boost the performance of applications with multiple levels of parallelism.

Table 7. Speed-up with respect to OpenMP-only with 20 processors for three different OpenMP versions.

	no_groups	groups	weigthed
20	0.44	1	1
40	0.33	1.5	1.7
60	0.26	1.51	1.9

5 Conclusions

The main purpose of this paper has been to examine the performance achievable when exploiting nested parallelism in two equivalent parallel versions of a coastal ocean circulation modeling application. One of the versions relies on mixing two programming paradigms to exploit the parallelism: MPI and OpenMP. The other version uses only OpenMP (using the possibility offered by the language definition of nesting parallel constructs and some extensions offered by the research OpenMP NanosCompiler).

The paper summarizes the extensions to OpenMP proposed to efficiently exploit nested parallelism. The extensions are mainly based on the definition of thread groups. The composition of the groups can be dynamically decided by the supporting OpenMP runtime system (using a predefined allocation strategy based on the amount of work to be performed by each group) or by the user using his/her own algorithm for deciding the allocation. The paper presents results in which the default allocation is overridden by the user in order to better balance the work distribution by overlapping the execution of several grid blocks on the same group of threads.

Regarding the parallelization of MGPOM, the main conclusion from this study is that OpenMP alone is able to achieve a similar (or even better) performance than the one that could be achieved mixing two programming paradigms (such as MPI and OpenMP). The use of a single programming paradigm makes the development, tuning and maintenance process of parallel applications simpler. Close analyses of the results show that the scalability of the application is limited by parts of the application that update the boundaries of the blocks. The parallelization with OpenMP performs a static scheduling of loop iterations that degrade the locality of the memory hierarchy. In addition to that, the parallelization with OpenMP forces a large number of synchronizations that add extra overheads to the parallel execution time. We are currently investigating alternative parallelization strategies for these parts that may improve the efficiency of the parallel execution.

6 Acknowledgments

This work has been supported by the Spanish Ministry of Science and Technology and the European Union FEDER program under contract TIC2001-0995-C02-01, and by the European Center for Parallelism of Barcelona (CEPBA).

References

1. OpenMP Organization. OpenMP Fortran Application Interface, v. 2.0, www.openmp.org, June 2000.
2. M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra. MPI – The Complete Reference: Volume 1, the MPI Core. MIT Press, Cambridge, 1998.
3. E. Ayguadé, M. Gonzalez, J. Labarta, X. Martorell, N. Navarro, J. Oliver, NanosCompiler: A Research Infrastructure for OpenMP Extensions. 1st European Workshop on OpenMP (EWOMP'99), Lund (Sweden). September/October 1999.
4. M. Gonzalez, J. Oliver, X. Martorell, E. Ayguade, J. Labarta and N. Navarro. OpenMP Extensions for Thread Groups and Their Runtime Support. In Workshop on Languages and Compilers for Parallel Computing, August 2000.
5. 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.
6. S. Shah, G. Haab, P. Petersen and J. Throop. Flexible Control Structures for Parallelism in OpenMP. In 1st European Workshop on OpenMP, Lund (Sweden), September 1999.
7. P. Luong, C.P. Breshears and L.N. Ly, Application of Multiblock Grid and Dual-Level Parallelism in Coastal Ocean Circulation Modeling. Journal of Applied Mathematical Modelling, submitted for publication.
8. Silicon Graphics Computer Systems SGI. Origin 200 and Origin 2000. Technical Report, 1996.
9. European Center for Parallelism of Barcelona. Paraver and Instrumentation Packages Reference Manual. http://www.cepba.upc.es/paraver.