Support of Multidimensional Parallelism in the OpenMP Programming Model

Haoqiang Jin and Gabriele Jost*
NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000
<hjin,gjost@nas.nasa.gov>

Abstract

OpenMP is the current standard for shared-memory programming. While providing ease of parallel programming, the OpenMP programming model also has limitations which often effect the scalability of applications. Examples for these limitations are work distribution and point-to-point synchronization among threads. We propose extensions to the OpenMP programming model which allow the user to easily distribute the work in multiple dimensions and synchronize the workflow among the threads. The proposed extensions include four new constructs and the associated runtime library. They do not require changes to the source code and can be implemented based on the existing OpenMP standard. We illustrate the concept in a prototype translator and test with benchmark codes and a cloud modeling code.

1. Introduction

OpenMP [11] was introduced as an industrial standard for shared-memory programming with directives. It has gained significant popularity and wide compiler support. The main advantage of the OpenMP programming model is that it is easy to use and allows the incremental parallelization of existing sequential codes.

OpenMP provides a fork-and-join execution model in which 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 lexically enclosed by the parallel construct. 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` and `ORDERED` regions. Thread synchronization of all threads is required at the end of `PARALLEL` constructs.

An existing code can be easily parallelized by placing OpenMP directives around time consuming loops which do not contain data dependences, leaving the source code unchanged.

---

* Computer Sciences Corporation, M/S T27A-1, NASA Ames Research Center.

The ease of programming is a big advantage over a parallelization based on data distribution and message passing, as it is required for distributed computer architectures. There are, however, limitations to the programming model which can affect the scalability of OpenMP programs. Comparative studies (i.e. [7]) have shown that high scalability in message passing based codes is often due to user optimized work distribution and process synchronization. When employing OpenMP the user does not have this level of control anymore. For example, a problem arises if the outer loop in a loop nest does not contain a sufficient number of iterations to keep all of the threads busy. In [1] and [5] it has been shown that directive nesting can be beneficial in these cases. Even though the OpenMP standard allows the nesting of directives, this feature is not supported by most of the commercial compilers. Another issue is the workflow between threads. Data dependences may require point-to-point synchronization between individual threads. The OpenMP standard requires barrier synchronization at the end of parallel regions which potentially destroy the workflow, especially when nested parallel regions are used.

The SPMD programming model is a way to mimic the data distribution and workflow achieved by message passing programs. In this model the programmer expresses manually the data and work distribution among the threads. The data is copied to small working arrays which are then accessed locally by each thread. The work is distributed according to the distribution of the data. The parallelization of the loop nests requires to compute explicitly which threads executes which iteration. The bounds for each loop have to be calculated based on the number of threads and the identifier of each thread, therefore introducing changes to the source code. The programming style allows the programmer to carefully manage the workflow, but completely defeats the advantages of the OpenMP programming model.

We are proposing extensions to the OpenMP programming model, which allow the automatic generation of SPMD style code based on user directives. Our goal is to remove some of the performance inhibiting limitations of OpenMP while preserving the ease of programming. In the current work we are addressing the issue of work distribution in multiple dimensions and point-to-point thread synchronization.

The rest of the paper is structured as follows. In Section 2 we discuss related work regarding multidimensional parallelization and present our own approach in comparison. In Section 3 we give a description of our prototype implementation. In Section 4 we present two case studies to demonstrate the proposed concept and conclude in Section 5.

2. Multidimensional Parallelism

The OpenMP standard allows the nesting of the `OMP PARALLEL` directive. According to the standard, a new team of threads is created when an inner `OMP PARALLEL` directive is encountered. The new team is joined at the inner `OMP END PARALLEL`. The `OMP DO` directive cannot be nested without nesting the `OMP PARALLEL` directive.

At this point not many compilers support this type of directive nesting. For example, the SGI compiler does not support nested OMP PARALLEL directives, but provides some support of multidimensional work distribution. The SGI compiler accepts the NEST clause on the OMP DO directive [10]. The NEST clause requires at least two variables as arguments to identify indices of subsequent DO-loops. The identified loops must be perfectly nested and no code is allowed between the identified DO statements and the corresponding END DO statements. The NEST clause on the OMP DO directive informs the compiler that the entire set of iterations across the identified loops can be executed in parallel. The compiler can then linearize the execution of the loop iteration and divide them among the available single level of threads.

An example of a research platform that supports true nested OpenMP parallelism is the OpenMP NanosCompiler [3]. The OpenMP NanosCompiler accepts Fortran-77 code containing OpenMP directives and generates plain Fortran-77 code with calls to the NthLib thread library [9] currently implemented for the SGI Origin. In contrast to the SGI MP library, NthLib allows for multilevel parallel execution such that inner parallel constructs are not being serialized. The NanosCompiler programming model supports several extensions to the OpenMP standard to allow the user to control the allocation of work to the participating threads. The NanosCompiler extension to multilevel parallelization is based on the concept of *thread groups*. A group of threads is composed of a subset of the total number of threads available in the team to run a parallel construct. In a parallel construct, the programmer may define the number of groups and the composition of each one. 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 the number of groups. The rest of the threads are used to support the execution of nested parallel constructs. In other words, the definition of groups establishes an allocation strategy for the inner levels of parallelism. To define groups of threads, the NanosCompiler supports the GROUPS clause extension to the PARALLEL directive. The NanosCompiler also provides the PRED/SUCC extensions [4] in order to allow point-to-point synchronization between threads.

We propose the following extensions to OpenMP for support of multidimensional parallelism, for short MOMP directives. We introduce two new directives, TMAP and MDO, for mapping a team of threads to a grid of multiple dimensions and for distributing work in the multiple dimensions. The concept is illustrated for a two-dimensional grid. It can easily be extended to higher dimensions.

TMAP(ndim,sfactor1,sfactor2)

Maps a team of threads to a grid of multiple dimensions. ndim is the dimension of the mapped thread grid, currently 1 or 2; sfactor1 and sfactor2 are the shape factors for the grid, i.e. the ratio of the two factors is proportional to that of the grid sizes in each

dimension. For example, (2,1,1) defines a squared grid; (2,N,0) indicates that the number of threads mapped to the first dimension should not exceed N. See Figure 1 for an illustration.

MDO(idim[,gplow,gphigh])

Binds or distributes a worksharing DO loop to the 'idim' dimension of the thread grid. The optional parameters gplow and gphigh can be used to specify additional *ghost iterations* to be assigned on the low and high ends of the bound loop. This is to mimic the ghost points concept used in a message-passing program. For the (2,1,1) mapping in Figure 1, if loops K and J are bound to idim=1 and 2, respectively, threads 0,1,2,3 will be bound to the same iterations of loop K, while threads 0,4,8,12 will be bound to the same iterations of loop J. By default, threads are not synchronized at the end of an MDO loop, as opposite to the implicit synchronization at the end of an OMP DO loop. This selection reflects closer to the SPMD coding style. To enforce synchronization, the user needs to use explicit synchronization directives.

To support flexible synchronization among threads in the thread grid, we include two more directives, TSIGNAL and TWAIT.

TSIGNAL(idir[,...])

Sends a signal to the direction 'idir': -1 lower-neighbor in the first dimension, 1 higher-neighbor in the first dimension, -2 lower-neighbor in the second dimension, 2 higher-neighbor in the second dimension (see Figure 1). Multiple directions can be listed.

TWAIT(idir[,...])

Waits a signal from the direction 'idir': -1 lower-neighbor in the first dimension, 1 higher-neighbor in the first dimension, -2 lower-neighbor in the second dimension, 2 higher-neighbor in the second dimension. Multiple directions can be listed.

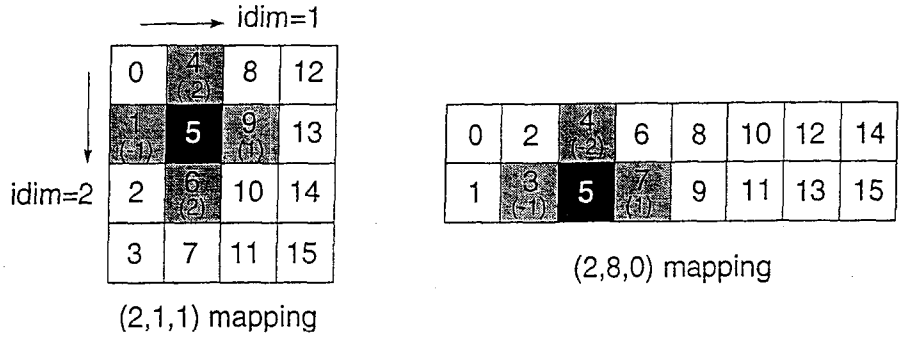

Figure 1: Examples of thread mapped grids with TMAP for 16 threads. The left (2,1,1) mapping defines a squared shape, while the right (2,8,0) mapping limits the number of threads mapped to the first dimension to 8, which then gives an 8×2 topology. The light shaded boxes indicate the neighboring threads of a given thread (5 in the figure) corresponding to idir=-1,1,-2,2.

TSIGNAL/TWAIT must always be used in a matching pair; else a deadlock will occur. We could easily include constructs for signal/wait of a particular thread if required.

A sample code using MOMP directives is given in Figure 2. The TMAP clause is added to the beginning of a parallel region to define a 2-D thread grid. The first MDO distributes the work of the K loop, while the second MDO distributes the work of the J loop. The MOMP extensions can work seamlessly with most of the OpenMP directives.

Serial code	Code with MOMP directives

DO K=1,NZ
  ZETA = K*0.1
  DO J=1,NY
  do more work...
  ENDDO
ENDDO
	!$OMP PARALLEL **TMAP**(2,NZ,0)
	!$OMP **MDO**(1)
	  DO K=1,NZ
	  ZETA = K*0.1
	!$OMP **MDO**(2)
	  DO J=1,NY
	  do more work...
	  ENDDO
	  ENDDO
	!$OMP END PARALLEL

Figure 2: A sample code using the MOMP directives.

In Figure 3 we compare code examples using MOMP to the Nanos and SGI extensions for multidimensional parallelization. The Nanos extensions support true multilevel OpenMP parallelism. The inner parallel region requires forking and joining of thread teams. The SGI extensions merely support linearization of the loop nest and require tightly coupled loops. They would not be applicable to the code example from Figure 2.

Code with MOMP directives	OpenMP with Nanos Extensions	OpenMP with SGI extensions

!$OMP PARALLEL
!$OMP& **TMAP**(2,NZ,0)
!$OMP **MDO**(1)
  DO K=1,NZ
!$OMP **MDO**(2)
  DO J=1,NY
  do more work...
  ENDDO
  ENDDO
!$OMP END PARALLEL
	!$OMP PARALLEL **GROUPS**(NZ)
	!$OMP DO
	  DO K=1,NZ
	!$OMP PARALLEL DO
	  DO J=1,NY
	  do more work...
	  ENDDO
	!$OMP END PARALLEL DO
	  ENDDO
	!$OMP END PARALLEL
		!$OMP PARALLEL DO
		!$SGI+**NEST** **ONTO**(NZ,*)
		  DO K=1,NZ
		  DO J=1,NY
		  do more work...
		  ENDDO
		  ENDDO
		!$OMP END PARALLEL DO

Figure 3: Code examples comparing the MOMP directives with Nanos and SGI extensions.

3. Prototype Implementation

To illustrate the concept of using the MOMP directives described above for multidimensional parallelization, we implemented a prototype translator and the supporting runtime library. The translator simply translates all the MOMP directives into appropriate runtime calls and leaves the other OpenMP directives intact. The resulting code is a standard OpenMP code and can be compiled with any OpenMP compiler and linked with the MOMP runtime library.

In Table 1 we summarize the translation of MOMP directives into the runtime functions for FORTRAN codes; the concept applies to C as well. The translation of `TMAP`, `TSIGNAL` and `TWAIT` is straightforward, simply mapping into the corresponding runtime functions. The `MDO` directive is translated into a call to `momp_get_range` that computes the new loop limit after the associated loop is distributed in the defined grid dimension, and the loop is then replaced with the new limit. For simplicity we only consider the block distribution which also simplifies the implementation of `momp_tsignal()` and `momp_twait()`. The ghost iterations are translated into the overlap of the iteration space between two neighboring threads. When no ghost iteration is involved, `(gplow,gphigh)=(0,0)` is used.

MOMP directive	*Runtime function*
`TMAP(ndim,sfactor1,sfactor2)`	`call momp_create_map(ndim,` `&  sfactor1,sfactor2)`

`MDO(idim,gplow,gphigh)`
`  DO lvar=low,high,step`
	`call momp_get_range(idim,`
	`&  gplow,gphigh,low,high,`
	`&  step,new_low,new_high)`
	`DO lvar=new_low,new_high,`
	`&  step`

`MDO(idim)`	`call momp_get_range(idim,` `&  0,0,...)`
`TSIGNAL(idir[,...])`	`call momp_tsignal(idir)` `[call momp_tsignal(...)]`
`TWAIT(idir[,...])`	`call momp_twait(idir)` `[call momp_twait(...)]`

Table 1: Translation of MOMP directives into the runtime functions.

In addition, an environment variable `MOMP_SHAPE` is used to control the grid shape externally. `MOMP_SHAPE` takes a value like "`S1xS2`," which is equivalent to specifying `sfactor1=S1` and `sfactor2=S2` to `TMAP`. This value overwrites the values given in `TMAP`. It allows a user to freely change the grid shape at runtime without recompiling the code. If `S1xS2=N` where `N` is the total number of threads, the shape "`S1xS2`" defines the current grid topology.

An example of the translation of the code given in Figure 2 is shown in Figure 4. The translated code is a standard OpenMP code with a proper list of private variables for the new loop limits and can be compiled with any OpenMP compiler.

Code with MOMP directives	Translated OpenMP code
!$OMP PARALLEL !$OMP& **TMAP**(2,NZ,0) !$OMP **MDO**(1) DO K=1,NZ ZETA = K*0.1 !$OMP **MDO**(2) DO J=1,NY do more work... ENDDO ENDDO !$OMP END PARALLEL	!$OMP PARALLEL PRIVATE(K_NLOW, !$OMP& K_NHIGH,J_NLOW,J_NHIGH) CALL **MOMP_CREATE_MAP**(2,NZ,0) CALL **MOMP_GET_RANGE**(1,0,0,1, & NZ,1,K_NLOW,K_NHIGH) DO K=**K_NLOW,K_NHIGH** ZETA = K*0.1 CALL **MOMP_GET_RANGE**(2,0,0, & 1,NY,1,J_NLOW,J_NHIGH) DO J=**J_NLOW,J_NHIGH** do more work... ENDDO ENDDO !$OMP END PARALLEL

Figure 4: Translation of a sample MOMP code to the standard OpenMP code with MOMP runtime calls.

4. Case Study

In this section we show examples for using our proposed MOMP directives. We will describe the usage of the directives and discuss the performance of the resulting code. All tests were run on an SGI Origin 3000 with 400MHz R12000 CPUs and 2GB local memory per node. We have parallelized two codes (BT and LU) from the NAS Parallel Benchmark suite [2]. We used the baseline implementation as described in [5]. In addition we have applied our extension to a full-scale cloud modeling code.

4.1. The BT Benchmark

BT is a simulated CFD application. It uses an implicit algorithm to solve the 3D compressible Navier-Stokes equations. The x, y, and z dimensions are decoupled by usage of an Alternating Direction Implicit (ADI) factorization method. The resulting systems are block-tridiagonal with a block size of 5x5. The systems are solved sequentially along each dimension. All of the nested parallel loops are at least triple nested. The structure of the loops is such that the two outer most loops can be parallelized and enclose a reasonably large amount of computational work. We applied the TMAP and MDO directives as shown in Figure 2 throughout the code. In Figure 5 we show timings obtained for the BT benchmark class A, which corresponds to the problem size of 64 grid points in each dimension. We compare timings achieved for different numbers of threads

running in various topologies. We denote by n_1 the first dimension of the thread topology. In our experiments n_1 is the total number of threads divided by the second dimension of the thread grid. For example, if we are running on 32 CPUs employing 32 threads, then the topology of $n_1 \times 4$ corresponds to employing 8 threads on the first and 4 threads on the second dimension.

The timings show that for more than 64 threads distributing the work in multiple dimensions can improves the performance significantly. The loop length resulting from 64 points does not provide enough iterations in one dimension to keep more than 64 threads busy. Distributing the work in two dimensions allows the exploitation of additional parallelism. The ratio of L2 cache misses vs. floating instructions per thread increases as more threads assigned in the second dimension due to the large stride memory access, which causes the increase in running time. But the positive impact of better workload balance is stronger than the negative impact of a lack of data locality in the run with 128 threads.

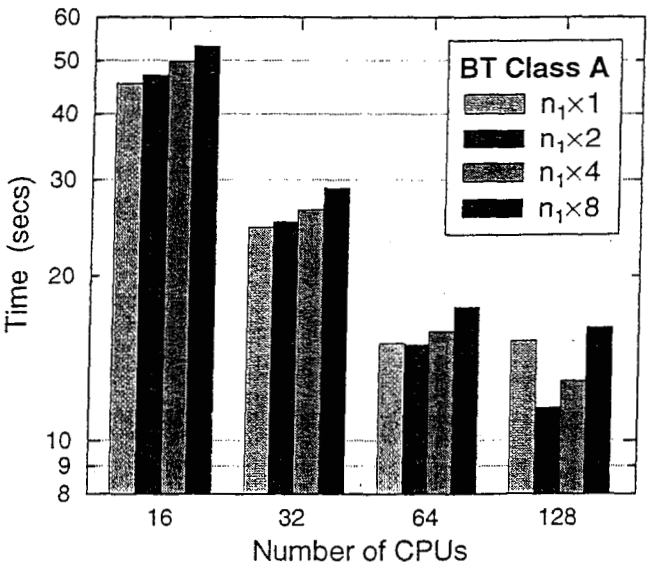

Figure 5: MOMP timing comparison for various numbers of threads running in different topologies. The first dimension n_1 corresponds to the number of CPUs divided by the second topology dimension.

In Figure 6 we compare timings achieved for different thread topologies for the MOMP version with timings obtained for the nested OpenMP version using the NanosCompiler. For the NanosCompiler we use the GROUPS clause extension as shown in Figure 3. The timings for the MOMP version seam to be slightly better than for the Nanos version, but not significantly. The extra barrier synchronization points at the end of inner parallel regions do not introduce considerable overhead to the NanosCompiler generated code, which is due to the very efficient NthLib thread library. We also noticed that the thread scheduling applied by the NanosCompiler yields a somewhat better workload balance than our approach. This indicates that additional performance might be gained for MOMP with an optimized runtime library.

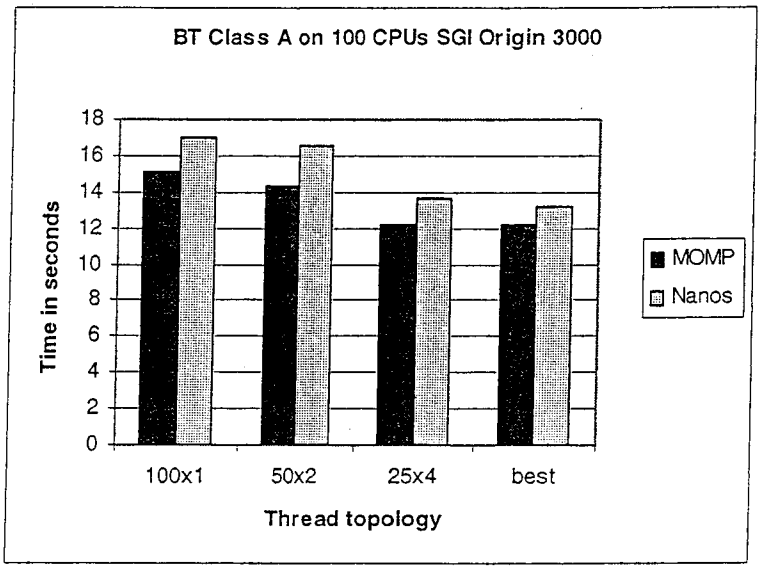

Figure 6: Time comparisons between Nanos and MOMP for different thread configurations. The columns in category *best* indicate the best time over all tested topologies.

The SGI `NEST` clause is not applicable to some of the time consuming loops in the BT benchmark because they are not tightly nested. More details on that can be found in [5].

4.2. The LU Benchmark

The LU application benchmark is a simulated CFD application that uses the symmetric successive over-relaxation (SSOR) method to solve a seven band block-diagonal system, resulting from finite-difference discretization of the 3D compressible Navier-Stokes equations by splitting it into block lower and block upper triangular systems. All of the loops involved carry data dependences that prevent straightforward parallelization. There is, however, the possibility to exploit a certain level of parallelism by using software pipelining which requires explicit synchronization of individual threads. We use the MOMP `TMAP` and `MDO` directives to distribute the work in multiple dimensions. Distributing the work in multiple dimensions requires a two-dimensional thread pipeline. We use the MOMP `TWAIT` and `TSIGNAL` directives to set up threaded pipeline execution in multiple dimensions. An example of the source code for the 2D pipeline implemented using the MOMP directives is shown in Figure 7

At this point we note that a two-dimensional pipeline involving nested OpenMP parallel regions runs into the problem that the inner parallel region imposes an extra barrier synchronization point which inhibits the 2D pipeline.

Code with MOMP directives

```
!$OMP PARALLEL TMAP(2,NZ,0)
    DO K=2,NZ
!$OMP WAIT(-1,-2)
!$OMP MDO(1)
      DO J=2,NY
!$OMP MDO(2)
        DO I=2,NX
          V(I,J,K) = V(I,J,K)+V(I-1,J,K)+
     &               V(I,J-1,K)+V(I,J,K-1)
          . . .
        ENDDO
      ENDDO
!$OMP SIGNAL(1,2)
    ENDDO
!$OMP END PARALLEL
```

Figure 7: Code example for the implementation of a 2D thread pipeline using MOM directives.

We used the Paraver [12] performance analysis system to show the effect of the 2D pipeline on the workflow of the threads. In Figure 8 we show the time line view of the useful thread time during the forward substitution phase of LU for a 16 CPU run. Dark shades indicate time the threads spend in computation, light shades indicate time spent in synchronization or other OpenMP introduced overhead. The left image shows a 16×1 thread topology, corresponding to a 1D pipeline. The right image shows a 4×4 topology corresponding to a 2D pipeline. The use of the 2D pipeline decreases a pipeline startup time for the computations.

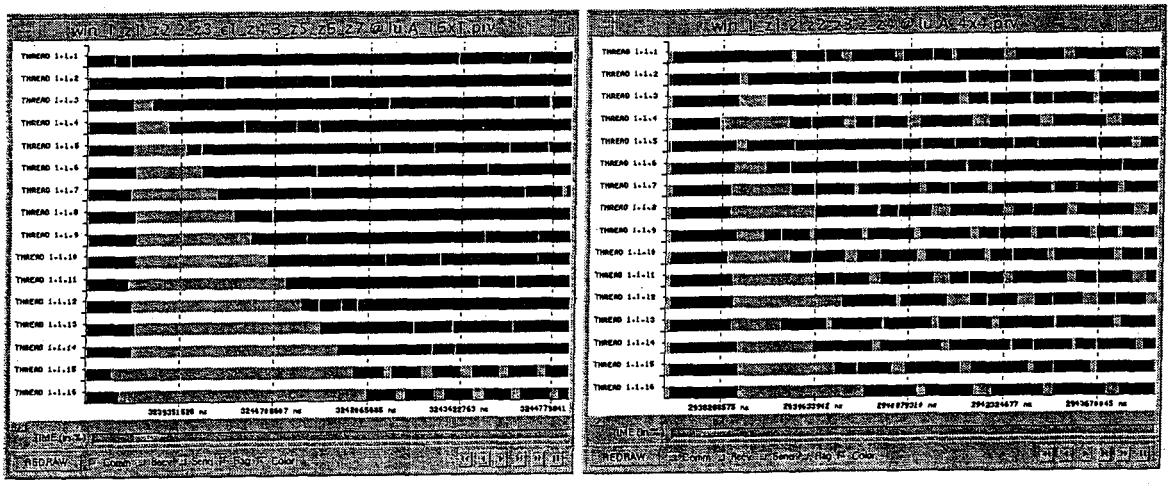

Figure 8: Time line views of the thread workflow during the forward substitution phase in LU running on 16 threads. Dark shading indicates time spent in computation, light shading indicates time spent in synchronization. The left images shows a 1D thread pipeline, the right image shows a 2D thread pipeline. The 2D pipeline decreases the pipeline startup time.

Just like in the case of BT, the LU benchmark provides only work for 62 threads on the outer loop level. Even though we were able to achieve a 2D pipelined thread execution, we did not gain any significant speedup by using extra threads and exploiting the inner loop for additional parallelism. Timings for LU benchmark class A problem are shown in Figure 9. A large increase in L2 cache misses for some of the threads lead to a very imbalanced workflow which decreased the performance. This is clearly indicated in Figure 10 where the effect of multidimensional parallelism in the LU benchmark on the value of various hardware counters is summarized.

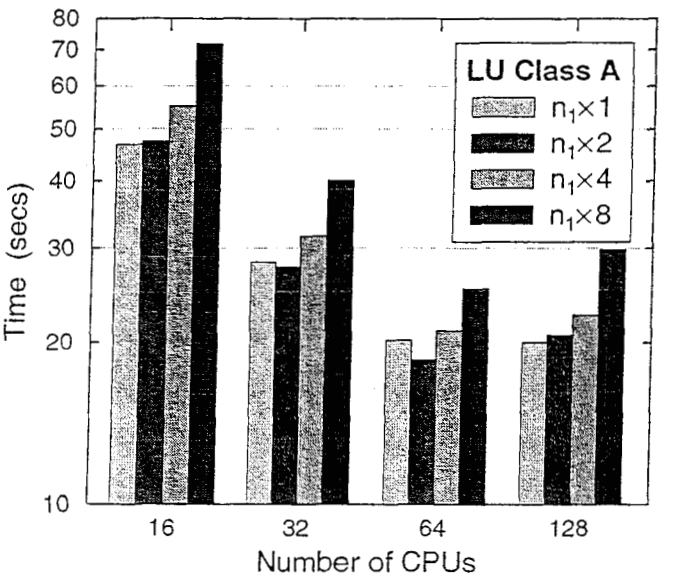

Figure 9: MOMP timings for various numbers of threads running in various topologies for LU benchmark class A. See Figure 5 for a note on the topology.

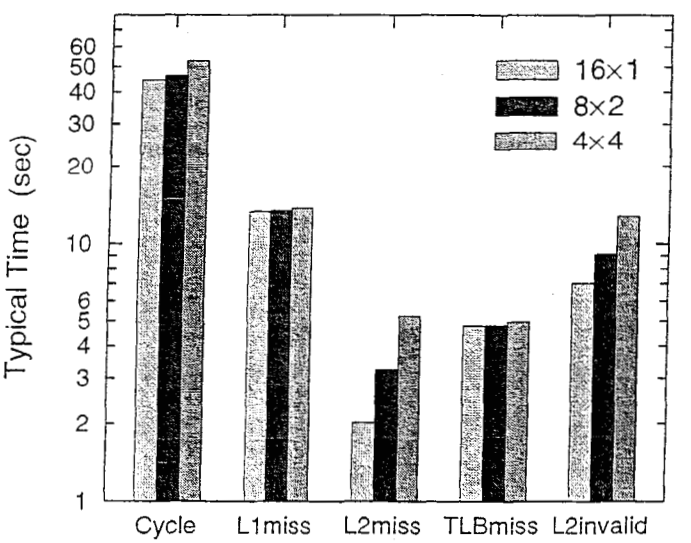

Figure 10: Effect of 2D work distribution on hardware counters for the LU benchmark.

4.3. The Cloud Modeling Code

The Goddard Cumulus Ensemble (GCE) code developed at the NASA Goddard Space Flight Center [13] is used for modeling the evolution of clouds and cloud systems under large-scale thermodynamic forces. The 3-dimensionsal version of this code, GCEM3D, was previously parallelized [6] using the standard OpenMP directives. The parallelization was performed on the outer loops (in many cases the vertical K dimension) to achieve coarse parallel granularity and little OpenMP overhead. However, due to the size limitation of the K dimension, the code does not scale beyond 32 CPUs even though the horizontal sizes are much larger (see [6]).

Use of the MOMP directives is a natural way to exploit parallelism in the horizontal dimension of the GCEM3D code. We use `TMAP` to define a 2-D thread mapping and `MDO` to bind the K loops to the first dimension of the thread grid and the J loops to the second dimension. The code requires the use of the ghost iterations in the `MDO` directives, as shown in Figure 11. The "`DO 200 J`" loop computes additional values for the `XY1` array in the low end so that `XY1(I,J-1)` can be used in the next J loop. Since the `XY1` array is declared as private, the use of "`MDO(2,1,0)`" ensures that each thread performs one extra calculation at the low end of the assigned iteration so that the `XY1(I,J-1)` value is available in the next J loop for each thread.

```
!$OMP PARALLEL PRIVATE(I,J,K,JM,XY1) TMAP(2,KLES-1,0)
!$OMP MDO(1)
      DO 100 K=2,KLES
!$OMP MDO(2,1,0)
        DO 200 J=1,JLES
          DO 200 I=2,ILES
            XY1(I,J)=...
  200   CONTINUE
!$OMP MDO(2)
        DO 300 J=2,JLES
          JM=J-1
          DO 300 I=2,ILES
            U1(I,J,K)=U1(I,J,K)+(XY1(I,JM)-XY1(I,J))
  300   CONTINUE
        ...
  100 CONTINUE
!$OMP END PARALLEL
```

Figure 11: Sample GCEM3D code that uses the ghost iteration feature of the MDO directive.

To test the effect of the MOMP directives, we applied them to the most time consuming routine 'fadvuw' in the GCEM3D code. The parallel code was run on 32 and 64 CPUs for a problem size of 258×258×34. The timing results are summarized in Table 2. As one can see, the single dimension parallelization (as indicated by 32x1 and 64x1) only improves the timing slightly

from 32 to 64 CPUs; the multidimensional parallelization with the MOMP directives reduces the timing on 64 CPUs from 51.1 seconds to 35.4 seconds.

CPUs	32	64	
Topology	32×1	64×1	32×2
Time (secs)	59.3	51.1	35.4
Ratio	-	1.16	1.68

Table 2: Timing obtained for the most time consuming routine 'fadvuw' running with different topologies of the thread grid.

5. Conclusion

We have proposed extensions to the current OpenMP programming model which allow the user to easily distribute the work in multiple dimensions within nested DO loops and synchronize the work flow between threads. The purpose is to exploit parallelism in multiple dimensions within a nest of loops. We have demonstrated the feasibility of our approach in several case studies. The advantage of the MOMP directives proposed in this work is that they are simple and clean. They can be implemented within the current OpenMP programming model and do not require changes to the OpenMP standard. The proposed work distribution directives support the automatic generation of SPMD style parallelization of loop. They do not impose restrictions on the structure of the loop nest, such as being tightly nested, nor do they require the nesting of parallel regions. Using the MOMP directives it is trivial to create a 2-D pipeline in LU, which would be difficult when using nested parallel regions.

Our case studies also demonstrated that distributing the work in multiple dimensions introduces disadvantages: The data is accessed employing large strides which can lead to severe cache problems. This can be seen in the case of the LU benchmark. The lack of data locality is always an issue with the shared-addressing programming model on cache-based systems. This issue needs to be addressed in future work, for example by automatically restructuring the code for cache optimization. Other possible enhancements to the MOMP directives include "`MBARRIER`" for barrier synchronization and "`MREDUCTION`" for reduction on a selected grid dimension.

Acknowledgements

This work was partially supported by NASA contract DTTS59-99-D-00437/A61812D with Computer Sciences Corporation.

Reference

[1] E. Ayguade, X. Martorell, J. Labarta, M. Gonzalez and N. Navarro, "Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study," Proc. Of the 1999 International Conference on Parallel Processing, Ajzu, Japan, September 1999.

[2] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," *RNR-95-020*, NASA Ames Research Center, 1995. NPB2.3, http://www.nas.nasa.gov/Software/NPB/.

[3] M. Gonzalez, E. Ayguadé, X. Martorell, J. Labarta, N. Navarro and J. Oliver. "NanosCompiler: Supporting Flexible Multilevel Parallelism in OpenMP." Concurrency: Practice and Experience. Special issue on OpenMP. vol. 12, no. 12. pp. 1205-1218. October 2000.

[4] M. Gonzalez, E. Ayguadé, X. Martorell and J. Labarta. "Defining and Supporting Pipelined Executions in OpenMP." 2nd International Workshop on OpenMP Applications and Tools. July 2001.

[5] H. Jin, M. Frumkin, and J. Yan, "The OpenMP Implementations of NAS Parallel Benchmarks and Its Performance," NAS Technical Report NAS-99-011, 1999.

[6] H. Jin, G. Jost, D. Johnson, and W-K. Tao, "Experience on the Parallelization of a Cloud Modeling Code Using Computer-Aided Tools," NAS Technical Report NAS-03-006, NASA Ames Research Center, March 2003.

[7] H. Jin, G. Jost, J. Yan, E. Ayguadé, M. Gonzalez, X. Martorell, "Automatic Multilevel Parallelization Using OpenMP," 3rd European Workshop on OpenMP (EWOMP01), Barcelona, Spain, September 2001.

[8] G. Jost, H. Jin, J. Labarta, J. Gimenez and J, Caubet, "Performance Analysis of Multi-level Parallel Programs on Shared Memory Computer Architectures," Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS03), Nice, France, April 2003.

[9] X. Martorell, E. Ayguadé, N. Navarro, J. Corbalan, M. Gonzalez and J. Labarta. "Thread Fork/join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors." 13th International Conference on Supercomputing (ICS'99), Rhodes (Greece). pp. 294-301. June 1999.

[10] MIPSPro 7 Fortran 90 Commands and Directives Reference Manual 007-3696-03.

[11] OpenMP Fortran/C Application Program Interface, http://www.openmp.org/.

[12] Paraver, http://www.cepba.upc.es/paraver/.

[13] W.-K. Tao, "Goddard Cumulus Ensemble (GCE) Model: Application for Understanding Precipitation Processes, AMS Meteorological Monographs," Symposium on Cloud Systems, Hurricanes and TRMM, 2002.