Toward Enhancing OpenMP's Work-Sharing Directives

Barbara M. Chapman[1], Lei Huang[1], Haoqiang Jin[2],
Gabriele Jost[3], and Bronis R. de Supinski[4]

[1] University of Houston, Houston TX 77004, USA
[2] NASA Ames Research Center, USA
[3] Sun Microsystems, Inc., USA
[4] Lawrence Livermore National Laboratory, USA
{chapman, leihuang}@cs.uh.edu, hjin@nas.nasa.gov,
gabriele.jost@sun.com, bronis@llnl.gov

Abstract. OpenMP provides a portable programming interface for shared memory parallel computers (SMPs). Although this interface has proven successful for small SMPs, it requies greater flexibility in light of the steadily growing size of individual SMPs and the recent advent of multithreaded chips. In this paper, we describe two application development experiences that exposed these expressivity problems in the current OpenMP specification. We then propose mechanisms to overcome these limitations, including thread subteams and thread topologies. Thus, we identify language features that improve OpenMP application performance on emerging and large-scale platforms while preserving ease of programming.

1 Introduction

OpenMP supports portable, high-level shared memory parallel programming and has been successfully deployed on small-to-medium shared memory systems (SMPs) and large-scale distributed shared memory platforms (DSMs). Its current version 2.5 [14] merges C/C++ and Fortran bindings and clarifies some concepts, especially with regard to the memory model. OpenMP 3.0 is expected to follow, and to consider a variety of new features. Among the many open issues are some tough challenges including extending OpenMP to SMP clusters and supporting other new architectures.

Several architectural trends to which we collectively call Chip MultiThreading (CMT) provide support for the simultaneous execution of two or more threads within one chip. It may be implemented through several physical processor cores in a chip (Chip MultiProcessor, CMP) [13], a single core with replication of features to maintain the state of multiple threads simultaneously (Simultaneous multithreading, SMT) [17] or their combination [9,10]. A hierarchical multithreading architecture results from using several of these chips in a single SMP. OpenMP was not designed for such hierarchical parallelism, nor to enable a programmer to assign different workloads to sibling threads in order to avoid resource contention. Traditionally, OpenMP targets computationally intensive, loop-based applications. CMT will probably dramatically increase the usage of OpenMP. Programmers will need language mechanisms that facilitate scalable parallel programming for these hierarchical systems, including flexibility in the assignment of work to threads.

W.E. Nagel et al. (Eds.): Euro-Par 2006, LNCS 4128, pp. 645–654, 2006.

In this paper, we describe two application development experiences from different domains that exposed problems with the expressivity of the current OpenMP specification. The first example involved porting an industrial seismic data processing application to OpenMP in order to create an easy-to-maintain version that exploited SMPs with hyperthreading. The language extensions we designed based on this effort turned out to have a much wider applicability. The second example comes from experiences gained while building scalable scientific applications on a large distributed shared-memory platform. Here too, the extensions facilitated an appropriate mapping of work to threads and led to a scalable parallel code. In each case, our inability to assign work to subsets of threads in the current thread team, and to orchestrate the work of different threads, in OpenMP 2.5 artificially limited performance. To overcome this, we propose a new clause for worksharing constructs that assigns the work to a subteam of the existing threads. Further, we introduce the notion of a topology, which gives a subteam a shape, and library routines to support these concepts. Finally, we also propose new constructs for improved work coordination between threads. We outline these applications and our proposed OpenMP extensions that facilitate programming them in the next two sections. Then, we discuss related work briefly before summarizing our findings.

2 Thread Subteams

Our experiences with commercial seismic data processing software initially motivated our thread subteam concept. Kingdom Suite from Seismic Micro-Technology, Inc. is an integrated geosciences interpretation software package for Windows systems used by the energy industry in the search for oil. OpenMP was applied to TracePak, an I/O-intensive module of Kingdom Suite to analyze and to process two-dimensional (2-D) and three-dimensional (3-D) post-stack seismic data [16]. Our goal was a parallel version for Windows-based SMPs with hyperthreading enabled. This version must be as close as possible to the original sequential code to simplify its maintenance, a common industrial requirement. Although our example could be programmed in a low-level style using thread IDs explicitly, this would require significant changes in the source code. In contrast, the suggested directives require only a minimal, localized modification of the source code and maintain the ease of programming that makes OpenMP a desired programming model. The subteam concept proposed here has been implemented in the OpenUH compiler [15]. It is comparatively straightforward, requiring less implementation effort than nested parallelism. WE are currently implementing our other proposals. Due to space limitations, implementation details will be addressed in a separate paper.

2.1 Seismic Data Processing on an SMT Platform

Fig. 1 shows the structure of the sequential program. This code iteratively reads data from an input file, processes it using different transform functions in a specified order, and then writes the results to an output file. The amount of seismic data typically handled in a job is quite large, ranging from 100MB to 100GB, and reading and writing consume considerable time.

Since OpenMP does not support parallel I/O, we decided that the best strategy to parallelize the code of Fig. 1 is to overlap the sequential I/O operations (lines 2 and 7)

```
1. for (i=0; i<N; i++) {
2.     ReadFromFile(i,...);
3.     for (j=0; j<ProcessingNum; j++)
4.         for (k=0; k<M; k++) {
5.             ProcessData(); //processing involves several
                              //different seismic functions
6.         }
7.     WriteResultsToFile(i);
8. }
```

Fig. 1. A sequential pseudo-code fragment for seismic data processing

with the parallelized computation (line 5), as illustrated by the timeline view in Fig. 2. A simple way to parallelize the computation is to enclose the innermost loop (k-loop) between threads in an "omp parallel for" directive. This approach, however, does not overlap the computation and I/O, and moreover, frequently entering and leaving parallel regions degrades performance. A dependence between the seismic data processing functions prevents parallelization of the outer loop (j-loop). In order to overcome these deficiencies, we enclose the entire loop nest in a parralel region as shown in Fig. 3. This version preloads the data needed for the first iteration of the i-loop (line 6). Then, we use "omp single nowait" and "omp for schedule(dynamic)" to enclose and to overlap the I/O operations and computation. One thread reads the data for the next iteration and another thread writes the results to an output file. The remaining threads share the work of the j loop (line 11 of Fig. 3). The dynamic schedule enables the threads performing I/O to subsequently join the computation.

The innermost, work-shared loop includes an implicit barrier at its end. Unfortunately, we cannot simply remove it since the data processing functions must follow a specific sequential order: each iteration uses results from the previous one. Thus although plenty of computation remains, the computing threads must wait at the implicit barrier until the I/O has completed, as shown in Fig. 4. Thus I/O operations and computation are not fully overlapped. Unfortunately, exchanging the order of the loops in the nest would, if possible, require a complete rewrite. However, a parallelization strategy that requires major code reorganization is unacceptable, as previously discussed.

2.2 Performance Improvement

In a normal run, the ratio of I/O and computation is about 1.2:1, where the I/O takes slightly longer than the computation. Thus, including the I/O threads in the barrier limits the overlap of I/O with computation. To determine how much removing this limitation

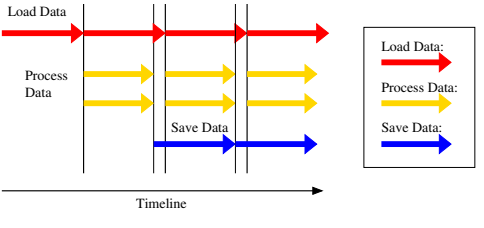

Fig. 2. Overlapping I/O with computation in the parallel seismic program

```
1.  #pragma omp parallel
2.  {   #pragma omp single
3.      {   //preload data to be used in the first iteration of the i-loop in line 6
4.          ReadFromFile(0,...);
5.      }
6.      for (i=0; i<N; i++) {
7.          #pragma omp single nowait
8.          {   //preload the data for next iteration of the i-loop
9.              ReadFromFile(i+1...);
10.         }
11.         for (j=0; j< ProcessingNum; j++)
12.             #pragma omp for schedule(dynamic)
13.             for(k=0; k<M; k++) {
14.                 ProcessData(); //user configurable data processing functions
15.             } //here is the barrier
16.         #pragma omp single nowait
17.         {
18.             WriteResultsToFile(i);
19.         }
20.     }
21. }
```

Fig. 3. The OpenMP code for seismic data processing kernel

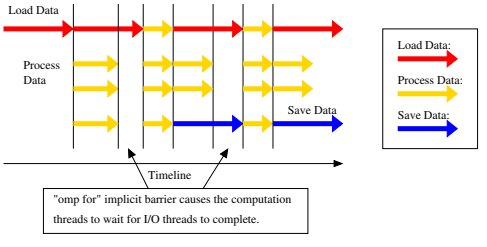

Fig. 4. Execution behavior of OpenMP seismic code

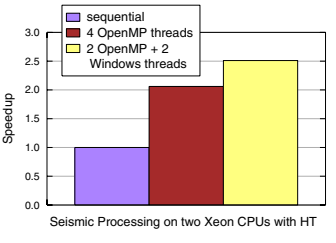

Fig. 5. Performance comparison: OpenMP vs. hybrid OpenMP and Windows API codes

would improve performance, we combined OpenMP with Windows threads for reading and writing files and achieved much greater overlap than with pure OpenMP. Fig. 5 shows results on an HP XW8200 with dual Xeon 3.4 GHz CPUs, 1MB L2 cache, 3GB memory, Intel extended memory 64, and hyperthreading technology. The compiler used was Microsoft Visual C++ in Visual Studio 2005 with OpenMP support. The hybrid version was 25% faster than standard OpenMP on four threads.

To achieve similar results with pure OpenMP, we require mechanisms to separate the computational threads from the data handling threads, and to synchronize their activities in the desired manner. We can achieve this with three parallel sections: read, write, and computation. The computation section would create a nested parallel region and share the work among its threads. We either prefetch data in the previous iteration, as in the code of Fig. 3, or use critical regions and arrays of variables. Unfortunately, each iteration of the outer i-loop requires a new parallel region if we are to retain the sequential program structure and the overheads for these are potentially high.

2.3 Thread Subteam as a Solution

Nested parallelism can dynamically create, exploit and terminate teams of threads and is well-suited to codes with needs that change over time. Our code structure is static. The relative amount of data and computation does not vary, and we expect the number of participating threads and their roles to remain the same. Nested parallelism is more powerful than we require. Thus, we propose a simpler mechanism that allows us to bind the execution of a worksharing or barrier construct to a subset of threads in the current team. Only the threads in the specified subteam participate in its work, including any barrier operations encountered. To synchronize the actions of multiple subteams, we may use existing OpenMP constructs and take advantage of the shared memory.

To realize this idea, we define an "onthreads" clause for worksharing and barrier directives. In contrast to nested parallelism, it refers only to existing threads. This clause permits us to specify that a worksharing directive is applied to a *subteam* of threads: participation in the associated work is restricted to the specified members. In particular, implicit and explicit barriers within the code it encloses do not block threads that are not part of the subteam. This clause would require minimal change to the current specification. In addition we can define an "onthreads" directive that could enclose arbitrary structured block of code within a parallel region. Work in the block would be carried out by the specified subteam of threads.

Using the thread subteam notation, we can rewrite the example code in Fig. 3 to that in Fig. 6. Line 5 and line 14 use the "onthreads" clause to limit the I/O to individual threads, while line 7 defines a subteam of threads to process the data. The integer expressions in parentheses use OpenMP's thread-ids and array section notation to specify the desired subset of threads. The implicit barrier at line 12 applies only to the threads defined in the subteam from line 7.

Additional syntax could enable the programmer to name these subsets. New run-time library routines would be provided to get the number of threads in a (named) subteam and a subteam-internal consecutive thread number. A programmer might also want to permute the order of threads in a subteam to specify schedules that enforce a certain work distribution, e.g. to support data reuse. Although none of these (except possibly the

```
1.  #pragma omp parallel
2.  {   #pragma omp single
3.          ReadFromFile(0,...); //preloads data for first iteration of i-loop
4.      for (i=0; i<N; i++) {
5.          #pragma omp single onthreads(0)
6.            ReadFromFile(i+1...); //preload data for next iter. of i-loop
7.          #pragma omp onthreads ( 2:omp_get_num_threads()-1 )
8.          for (j=0; j< ProcessingNum; j++)
9.            #pragma omp for schedule(dynamic)
10.           for (k=0; k<M; k++) {
11.              ProcessData(); //user configurable data processing functions
12.           } //here is the group-internal barrier
13.         #pragma omp barrier  //this ensures we are ready for next iter.
14.         #pragma omp single onthreads(1)
15.           WriteResultsToFile(i);
16.     }
17. }
```

Fig. 6. OpenMP seismic data processing kernel with the "onthreads" directive

library routines) are essential, they would greatly increase the expressive power of this construct. Interactions between subteams could be made explicit by providing notation for communication between subteams. This might help a programmer reason about the structure of this communication and avoid programming errors such as deadlock. The same construct might also enable point-wise synchronization between threads in a single subteam to avoid barriers. In the code fragment of Fig. 7, a post-wait notation does this succinctly and we have named the thread team, whose order is a permutation of the original thread numbers (used here only to illustrate the concept).

```
#pragma omp parallel
{
   #pragma omp team CompthreadsReordered = threads(omp_get_num_threads()-1:2:-1)
   for (i = 0; i < N; i++) { //executed by all threads
       #pragma omp single onthreads(0)
       {  ReadFromFile(i);
          #pragma omp post (dataready[i])  //signals reading is complete
       } //thread(0) independently does this reading and posting
       ........
       #pragma omp on CompthreadsReordered
       {   //subteam starts to work
          #pragma omp wait (dataready[i])  //after data is ready
```

Fig. 7. Excerpt from OpenMP code with named subteam and post/wait

The ability to divide work among subteams of threads, and thus to have different subteams working concurrently and independently, seems to be a fairly natural extension to the current API and it has a variety of potential uses. It would likely simplify the use of OpenMP within third party libraries. It also enables the specification of multidisciplinary code ensembles and permits components written in traditional programming languages to interact without the need to provide external file-based interactions. It supports the simpler case of multilevel parallelism with a fixed team of threads without the extra overheads and burden of nested parallelism.

3 Worksharing and Synchronization Across Loop Nests

Scientific and engineering computations must exploit large numbers of threads, not only in emerging, very large shared-memory systems, but also in smaller SMPs with CMPs. Writing scalable code requires special care. Two of the authors previously proposed a set of language features to enable the parallelization of multiple levels of loop nests [8]. These features specify an appropriate execution schedule and assign threads to loop levels, as well as additional synchronization that enables a pipelined execution scheme in the LU benchmark from the NAS Parallel Benchmarks [2]. They addressed scalability limitations in several applications despite the presence of sufficient inherent parallelism.

3.1 The LU Example

The LU application benchmark uses the symmetric successive over-relaxation (SSOR) method to solve a seven band block-diagonal system. Figure 8 illustrates its lower

triangular phase. References to values of elements of array v in line 4 create dependences between loop iterations that prevent straightforward parallelization. However, a wave-front or a pipelined technique can enable considerable levels of parallelism to be exploited, since the value of an element of v can be computed once the new values are available from the previous iteration in each of the three dimensions.

A wave-front restructuring of the code reveals parallelism that can be expressed with the existing OpenMP parallel directive to update points on a diagonal plane concurrently. However, this method suffers from poor cache utilization. A pipelined approach, in which data are partitioned as blocks in selected dimensions, usually gives better cache performance. We illustrate the differences between wave-front and pipelined parallelism in Fig. 9. Expression of the parallelism in two dimensions would reduce the cost of pipeline startup and shutdown, and support good cache performance for this kernel. However, OpenMP currently can only successfully exploit parallelism in one dimension. Parallelization in multiple dimensions requires nested parallelism, which results in multiple one-dimensional pipelines and incurs high overheads [7].

```
1. for (k = 1; k < nz; k++) {
2.   for (j = 1; j < ny; j++) {
3.     for (i = 1; i < nx; i++) {
4.       v[k][j][i] =
           v[k][j][i] +
           a*v[k][j][i-1] +
           b*v[k][j-1][i] +
           c*v[k-1][j][i];
5.       . . .
6.     }
7.   }
8. }
```

Fig. 8. The LU computational kernel

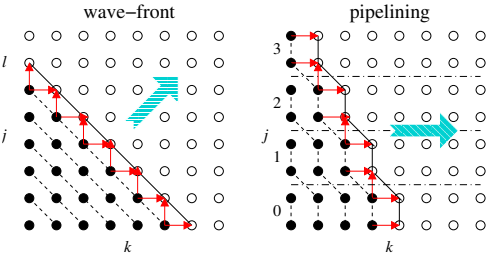

Fig. 9. Wave-front and pipelined algorithms. j,k are data dimensions. l in the left panel indicates a diagonal plane. Numbers in the right panel indicate data blocks mapped to different threads.

3.2 Thread Topology

We introduce the notion of a thread topology to support pipelined algorithms. A thread topology does not create new threads; instead, it *reshapes* the thread (sub)team and associates a new naming scheme with existing threads. We can use the topology to specify a variety of new schedules for worksharing directives. Our syntax requires the programmer to provide the number of dimensions in the topology and the coordinates in each dimension. We will also need a default strategy for mapping the linearly numbered threads to a Cartesian grid. The basic syntax of specifying a topology is:

```
#pragma omp topology name(ndim,start,stop,stride,fixedorder)
```

where `name` defines a name of the topology. The `ndim` argument specifies the number of dimensions. The arguments `start`, `stop`, and `stride` are arrays with one entry per dimension to specify the topological shape. `fixedorder` is a Boolean variable that tells the compiler whether or not the default strategy for associating these threads with the linear thread numbers must be applied. If not, the system can choose

any mapping of threads to the topology. For example, if 16 threads exist, the directive can reshape threads into a $4 \times 2 \times 2$ grid with coordinates from `start[]`=(0,0,0) to `stop[]`=(3,1,1) and `stride[]`=(1,1,1) or any other numbering scheme we desire that has 16 threads. We can associate a defined topology with a worksharing construct using the "onthreads" clause. We use standard section notation to specify the target of the worksharing directive in each topological grid dimension. We use ":" to denote the entire dimension of an array. Dimensions not involved in the worksharing are marked via a dummy "*" and the computation is replicated in those dimensions. A runtime function "omp_get_coord(`name`,`idim`)" can obtain coordinates of a thread in the grid topology.

We illustrate the use of our topology notation in Fig. 10 for the LU computational kernel. We introduce a 2-D logical grid of threads with the same number of threads in each dimension. Our thread subteam clause maps the iterations of two different loops to threads using our grid topology through two worksharing constructs (this notation does not conform to current OpenMP rules). The 2-D topology is used to distribute the work in the i and j loop nests among threads.

Finally, we need a way to define synchronization between threads in a topology. We cannot use existing features of OpenMP, since the interaction required is not between iterations but threads. This is achieved here using `post` and `wait` directives with our 2-D thread-ids. In our example, each thread of the topology must wait for its neighbors to the left and below it to finish their computation except for where the thread does not have a neighbor. For instance, thread 0 does not have a neighbor and can start right away. Once its work is done, a thread signals its neighbors to the right and above that they can continue. The ability to synchronize between threads is very important for implementing the pipelined approach in the LU algorithm. In general, it enables loosely synchronous algorithms [12].

```
   mystart[0] = 0; mystart[1] = 0; ... // assign values to mystart[:] and mystop[:]
  #pragma omp parallel {
  #pragma omp topology grid(2,mystart,mystop,mystride,1)
                                          // arrange threads logically into a square called grid
   iam1 = omp_get_coord(grid,1);
   iam2 = omp_get_coord(grid,2); // my coords in grid
1. for (k = 1; k < nz; k++) {
     #pragma omp wait grid (iam1-1,iam2)  // wait for thread below to complete its portion
     #pragma omp wait grid (iam1,iam2-1)  // wait for thread on left to complete its portion
     #pragma omp for nowait onthreads(grid(:,*))  // share out to first dimension of grid
2.   for (j = 1; j < ny; j++) {
       #pragma omp for nowait onthreads(grid(*,:))  // share out to second dimension of grid
3.     for (i = 1; i < nx; i++) {
4.       v[k][j][i] = v[k][j][i] + a*v[k][j][i-1] +
                      b*v[k][j-1][i] + c*v[k-1][j][i];
5.       . . .
6.     }
7.   }
     #pragma omp post grid(iam1,iam2+1)  // indicate to thread on right that it is ready
     #pragma omp post grid(iam1+1,iam2)  // indicate to thread above that it is ready
8. }
}
```

Fig. 10. The multilevel LU computational kernel using thread topology

4 Related Work

The NanosCompiler team has proposed groups of threads in association with parallel regions [5,6]. Their notation permits the user to specify the number of independent teams of threads that will be created. Since these thread groups are associated with the parallel region, additional notation is required to assign work to the individual groups. They also propose extensions to express the precedence relations in pipelined computations. These extensions are also valid in the scope of nested parallelism and are based on the ability to name worksharing constructs and to specify a predecessor-successor relationship between them to support synchronization. Our topology simplifies specifying the desired target sets and is more intuitive than the predecessor-successor relationship. Furthermore, it does not rely on nested parallelism and the associated overhead.

There have been a variety of proposals for multilevel loop parallelism. The SGI compiler for the Origin [11] provides the SGI NEST clause on the OMP DO directive. The NEST clause requires at least two variables as arguments to identify indices of subsequent DO-loops, which must be perfectly nested. It informs the compiler that the entire set of iterations across the identified loops can be executed in parallel. The compiler can then linearize the iteration space and divide it among the threads. Intel has proposed a new directive to enable wavefront execution schema. Although this might sometimes be appropriate, we expect that it will be hard to achieve good data locality in most cases. Our proposal explicitly enables control of work distribution and, thus, enables the expression of data locality.

New programming languages [1,3,4] are being proposed to facilitate high end application development in a multithreading environment. These languages address problems faced by levels of scaling that are far from those currently envisaged for hierarchical SMPs, and they provide a wealth of new ideas related to correctness, locality, efficiency of shared memory updates, and more. We will explore these ideas in the context of OpenMP.

5 Conclusions

OpenMP is a widely deployed shared memory programming API that offers the promise of performance and ease of use. It seems possible that the judicious addition of language features that increase the power of expressivity might also improve the achievable performance of a variety of OpenMP codes. In this paper, we introduced a unified notation for sharing work among subteams of threads and for flexibly executing multiple levels of loop nests in parallel. Table 1 lists the proposed new OpenMP constructs and clauses in the paper. This approach fits in well with existing features of the API. As our future work, we will conduct more detailed performance study of the proposed subteam concept implemented in the OpenUH compiler.

Table 1. Proposed new OpenMP Constructs and Clauses

Proposed OpenMP Directives/Clauses	Description
omp onthreads / onthreads (clause only)	Defines thread subteams for work sharing
omp topology name	Defines the thread topology
omp post / omp wait	Uses for point-wise synchronization

References

1. E. Allen, D. Chase, V. Luchangco, J-W. Maessen, S. Ryu, G.L. Steele Jr., S. Tobin-Hochstadt. "The Fortress Language Specification, Version 0.785." http://research.sun.com/projects/plrg/fortress0785.pdf
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.
3. Cray Inc., "Chapel Specification 0.4." http://chapel.cs.washington.edu/specification.pdf
4. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing." in the proceedings of OOPSLA '05, pp. 519-538, 2005
   K. Ebcioglu, V. Saraswat and V. Sarkar. " X10: Programming for hierarchical parallelism and nonuniform data access (extended abstract)." OOPSLA 2004), October 2004.
5. M. Gonzalez, E. Ayguade, X. Martorell and J. Labarta. "Defining and Supporting Pipelined Executions in OpenMP." in the proceedings of WOMPAT 2001, July 2001.
6. M. Gonzalez, J. Oliver, X. Martorell, E. Ayguade, J. Labarta, and N. Navarro. "OpenMP Extensions for Thread Groups and Their Run-time Support." in the proceedings of LCPC'2000, New York (USA), pp. 317-331, August 2000.
7. H. Jin, G. Jost, J. Yan, E. Ayguade, M. Gonzalez, and X. Martorell, "Automatic Multilevel Parallelization Using OpenMP," Scientific Programming, Vol. 11, No. 2, pp. 177-190, 2003.
8. H. Jin and G. Jost. "Support of Multidimensional Parallelism in the OpenMP Programming Model," WOMPEI2003, Tokyo, Japan, October 2003, in the Proceedings of the International Symposium on High Performance Computing (ISHPC-V).
9. R. Kalla, B. Sinharoy, and J. Tendler. "IBM POWER5 chip: a dualcore multithreaded processor", in IEEE Micro, 24(2): 40-47, 2004.
10. P. Kongetira. "A 32-way Multithreaded SPARC Processor", in Hot Chips 16, http://www.hotchips.org/archives/hc16/.
11. MIPSPro 7 Fortran 90 Commands and Directives Reference Manual 007-3696-03. http://techpubs.sgi.com/.
12. Z. Liu, B. Chapman, Y. Wen, L. Huang and O. Hernandez. "Analyses and Optimizations for the Translation of OpenMP Codes into SPMD Style," Proc. WOMPAT 03, LNCS 2716, 26-41, Springer Verlag, 2003.
13. K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The Case for a Single-Chip Multiprocessor", in Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, 1996, pp. 2-11.
14. OpenMP Application Program Interface, Version 2.5, May 2005. http://www.openmp.org/drupal/mp-documents/spec25.pdf
15. "The OpenUH compiler project", http://www.cs.uh.edu/ openuh
16. Sesimc Micro-Technology, Inc., TracePak Module, http://www.seismicmicro.com/Prod_Geo.htm.
17. D. Tullsen, S. Eggers, and H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", Intl. Symp. on Computer Architecture, pp. 392-403, 1995.