Performance Characteristics for OpenMP Constructs on Different Parallel Computer Architectures

Rudolf Berrendorf, Guido Nieken
Central Institute for Applied Mathematics, Research Centre Jülich
52425 Jülich, Germany
{ r.berrendorf*, g.nieken } @fz-juelich.de

Abstract

OpenMP is emerging as a quasi-standard for shared memory parallel programming on small SMP-systems. To serve as a common programming interface in shared memory parallel programming, scalability to a larger number of nodes and support for different shared memory architectures has to be proven. In this paper we investigate how well the basic constructs of OpenMP are implemented on different parallel computer architectures.

1 Introduction

OpenMP is proposed as the uniform Application Programming Interface (API) for portable shared-memory parallel programming. The API is intended to provide a parallel programming model that is portable across shared memory architectures from different vendors. To serve as a base platform for developing applications on top of it, highly efficient implementations of OpenMP are necessary on shared-memory architectures. The efficiency must be proven on different types of shared memory implementation, and must hold for a larger number of processors.

In this paper, we analyze the efficiency of OpenMP implementations on different hardware platforms. To do this in an application independent manner, we analyze the basic primitives of the application programming interface. Parallelism is started in OpenMP with a parallel region construct. An important issue in scalability considerations of parallel programs is the efficiency of parallel loops, as parallel loops are the major source of parallelism in scientific programs. Explicit synchronization of threads can be done in OpenMP with barriers, critical sections, locks, atomic operations, and reduction operations. For implicit synchronization through memory reads/writes, OpenMP has the *FLUSH*-directive to ensure a consistent view of certain variables in all threads in a thread team.

* corresponding author

We analyze performance numbers for most of the mentioned basic OpenMP constructs on different target platforms: vector-parallel computers (SGI/CRAY T90, SGI/CRAY J90), cc-NUMA computer (SGI Origin2000) and SMP-computer (IBM R50). The main parameter in all of the benchmarks is the number of threads participating in each construct, as scalability is one of the main aspects in this study.

The paper is organized as follows. After giving a short overview of OpenMP in section 2, we specify in section 3 the parallel computers we used in our study. Section 4 gives the performance results for the OpenMP constructs we investigated. In section 5 we discuss related work and conclude our work in section 6.

2 OpenMP

OpenMP [DM98] is an API aimed for portable shared-memory parallel programming which defines directives/pragmas, functions, and environment variables as an interface to the system. Currently, language bindings exist for Fortran, C, and C++.

OpenMP is based on the fork-join programming model. The parallel region construct forks a number of threads executing (redundantly) the code inside the region in parallel. The number of spawned threads may be influenced by several factors (e.g. settings done through subroutine calls, environment variables). The mapping of threads to processors, the number of processors assigned to the program, and whether the processors are available during a whole parallel region is system dependent. Work sharing inside a parallel region may be done in a parallel loop or in a parallel section. There are several options which influence the behaviour of parallel loops, e.g. the scheduling strategy. By default, a barrier synchronization is done at the end of a work sharing construct, but this can be changed by the option *NOWAIT* given to the closing directive of that construct. Combined constructs exist to specify a parallel region with a single parallel loop or parallel section. There are several constructs to synchronize threads: barrier, critical section, atomic operation, data lock, reduction opera-

tion. To correctly implement synchronization through memory operations (i.e. writing and reading values to/from memory locations by different processors), the use of the *FLUSH*-directive is necessary which synchronizes a threads' view of memory with the global memory seen by other threads.

Currently, the OpenMP-specification misses constructs to control which processors access which data objects, e.g. through data distributions [Hig97] or work distributions [BG95]. While this might be of less concern on a dual-processor system or a UMA (*Uniform Memory Access*) system, processor locality of data is one of the key aspects for good performance on cache based or non-uniform memory access parallel computers with a large number of processors. SGI has therefore added for their Origin2000-systems of type cc-NUMA (*cache coherent Non Uniform Memory Access*) with up to 256 processors a number of directives to their OpenMP-implementation to control the association of processors and data.

The OpenMP Fortran-API version 1.0 (at the time of writing this is the actual version) leaves many things unclarified. There is another document [Ope99] published by the OpenMP Forum with tries to clarify some of the open questions.

3 Parallel Computers

OpenMP relies on a shared memory implemented by the underlying system either in software (e.g. [ACD+96]) or in hardware. Hardware implementations of a shared memory across processors might be done in different ways, and there is a long history in doing this. We used in our investigation different types of hardware implementation to evaluate the scalability of the OpenMP programming model on that systems. Below we will give a short overview on the systems we used.

3.1 SGI/CRAY T90

The CRAY T90 is a uniform memory access (UMA) vector parallel machine with up to 32 processors accessing the (fast) global memory through a sophisticated multistage interconnection network. All processors see the same latency accessing any memory location. We used a 10 processor T90 with 475 Mhz processors. OpenMP is implemented in the Cray CF90 Fortran compiler (version 3.3). We used '*-O3*' as the compiler option.

3.2 SGI/CRAY J90

The CRAY J90 is a UMA-type machine similar to the CRAY T90, but with a slower DRAM memory. We used a machine with 16 processors running at 100 MHz (*J90 classic*). The same Fortran compiler and the same compiler option was used as on the CRAY T90.

3.3 SGI Origin2000

The Origin2000 is a cache-coherent non-uniform memory access (cc-NUMA) multiprocessor with up to 256 processors in the latest configurations. The building blocks are dual-processor boards with 4 MB L2 cache, local memory, and a hub implementing the global address space across all nodes of the computer. Memory for shared variables is allocated on one node (e.g. following the first touch strategy) and might be copied on a cache-line base to another node referring a location within that cache line. The machine we used had 128 processors running at 300 MHz. OpenMP is implemented in the latest Fortran Compiler (MIPSpro version 7.2) of SGI. The compiler options we used were '*-mp -O3*'.

3.4 IBM R50

The IBM R50 is a SMP-type parallel machine with 8 processors (dual processor boards) of type PPC 604e (200 MHz) and a 2 MB large L2 cache per board. The main memory is accessed through a crossbar switch. The operating system in use was AIX 4.3, and the compiler (xlf95_r) has release level 6.1. We used '*-qsmp=omp -O4*' as compiler switch.

4 Performance

Unless otherwise noted, we used for all measurements the default system settings, e.g. the default hold time for processors. We repeated all measurements 5 times and took the best of these 5 measurement as the result. The timing was done in an environment like the following:

```
!$OMP BARRIER
  t0 = get_time();
  OpenMP construct
  time = get_time()-t0-time_overhead;
```

All figures shown have a log-log-scale.

4.1 Parallelism

The basic parallelism model in OpenMP is to fork a number of threads (*team*) in a parallel region. These threads execute the code within the region in parallel. Inside the region a SPMD-type model might be used where all threads work on their own data, or a work sharing programming model might be used where the threads work cooperatively e.g. on a parallel loop, or a task parallel approach with the use of the parallel section construct, or a combination of these.

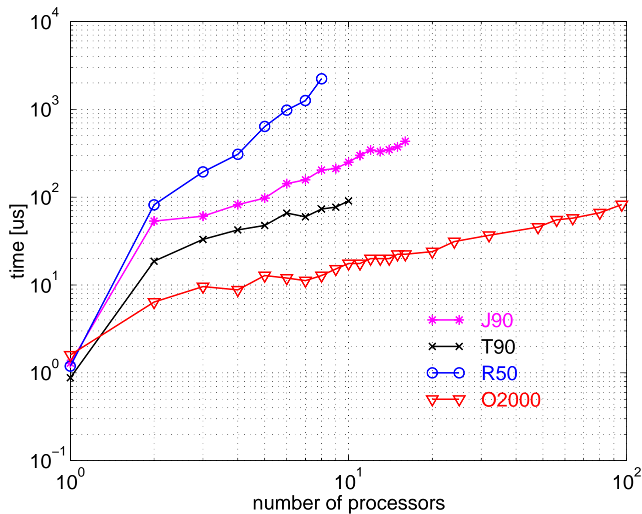

Figure 1: Performance of parallel region construct.

4.1.1 Setting up a Region

The parallel region construct forks a number of threads working in parallel on a section of code. Fig. 1 shows the performance of the parallel region construct as a function of the number of participating processors.
As can be seen in the figure, the overhead is very small on the Origin2000 system with a good scalability up to 100 processors. Rather costly is a parallel region on the IBM R50 system even for a small number of processors. In OpenMP it is possible to use work sharing directives in the dynamic extent of a parallel region (i.e. a parallel loop in a subroutine called in a parallel region; *orphaned directives*) therefore allowing parallel regions of larger (code) extent. This might help in compensating the overhead involved with the setup of a parallel region and therefore the overhead for a parallel region might be of less concern for those applications which have rather large parallel regions.

4.1.2 Parallel Loops

Parallel loops are typically the major source of parallelism in fork-join type scientific programs. Therefore, it is important to efficiently schedule iterations of such a parallel loop to processors. There are 4 scheduling alternatives in OpenMP how to distribute the iterations to the threads in the team working on the parallel loop:

- nothing specified: implementation dependent
- *static*: Do a block distribution of iterations to threads. With an additional chunk size parameter, block cyclic distributions are possible.
- *dynamic*: Self scheduling, where each idle thread gets a chunk of iterations. The default chunk size is 1.
- *guided*: Guided self scheduling, where the number of tasks a thread gets shrinks during subsequent requests, i.e. the first thread gets more iterations than later arriving idle threads. Again, an optional chunk size is possible.

In real application programs running on a larger number of processors, data access and processor locality might be the dominating performance factor. In our tests we were rather interested in the basic overhead of a parallel loop and the scalability on a given architecture. We tested three version with every scheduling strategy: constant amount of work in each parallel iteration, increasing amount (e.g. iteration 1 takes 40 μs, iteration 2 takes 41 μs etc.), and decreasing amount of work. We used the default chunk size in all cases. The total number of parallel iterations was in all tests 100.000, and the total time of the sequential reference loop was kept on each architecture at approx. 20 seconds for up to 16 processors and 40 seconds for more than 16 processors. Inside the loops only private data was accessed.
With the exceptions discussed below, the loop results on all systems show nearly linear speedup as no data conflicts exist and the number of processors is moderate with the exception of the Origin2000.
The static scheduling scheme has the lowest overhead as every thread is able to calculate from the known loop boundaries and the number of participating threads its own iteration space. Neither communication nor synchronized access to a global variable is necessary. On the other hand, static scheduling works only if there is an equal amount of work in the blocks the threads work on (leaving the aspect of data communication out of the discussion). Therefore, the cases with increasing and decreasing amount of work in the iterations for the static loop do not perform well.
Fig. 2 shows some interesting cases with dynamic scheduling algorithms where problems have shown up for a larger number of processors. For a small number of processors all speedup curves are nearly identical.
The guided self scheduling algorithm [PK87] calculates the number of iterations x_i the i-th idle thread receives based on a recursive formula: $x_i = \frac{1}{p} R_i$ where R_i is the number of remaining iterations and p is the number of processors (resp. threads). On CRAY J90/T90 the guided self scheduling algorithm is implemented in a slightly modified manner where instead of p the value $1 - \frac{p-1}{p}$ is taken and the denominator is adjusted to a power of 2. The combination of the modified algorithm and the fact that in a loop with a decreasing amount of work, iterations with more work are done first, results in the curve shown in Fig. 2 where no more improvements for that loop can be seen in using more than 8 processors.
The dynamic scheduling scheme is usually implemented with atomic accesses to a shared variable which holds the loop

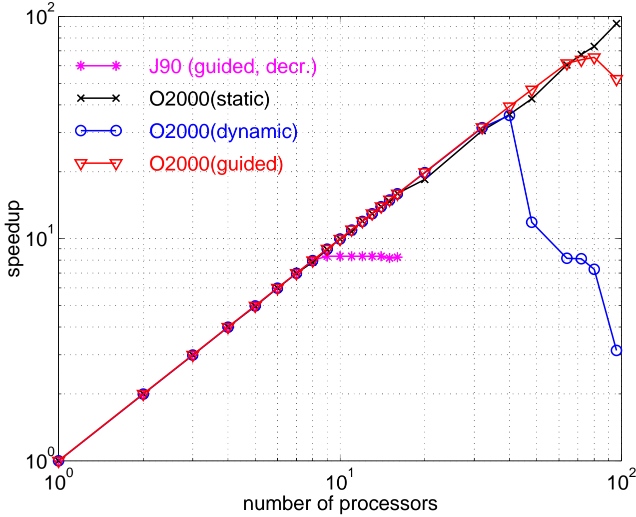

Figure 2: Selected loop results.

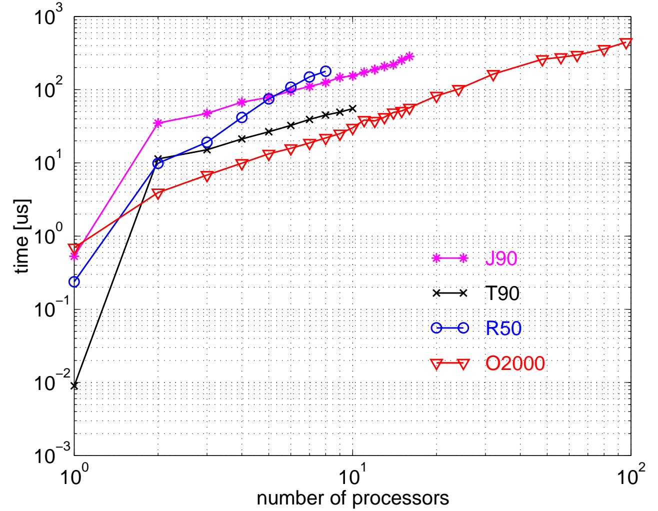

Figure 3: Performance of barrier operation.

count. For a (default) chunk size of 1 this means that every iteration involves an atomic access to this variable. Fig. 2 shows for the dynamic scheduling (with a constant amount of work) a nearly linear curve up to 43 processors. After that, the performance curve steeply drops. The reason for this effect is that for that number of processors the atomic access to the shared loop variable gets the bottleneck as many processors compete for the access. The exact number where the curve drops is influenced by the amount of work in the iterations and the number of processors, i.e. how frequently the shared variable needs to be accessed. Putting more work in our test loop moves the turning point to a larger number of processors.
The guided self scheduling strategy on the Origin2000 shows also this type of degradation but as not every loop iteration causes an access to the shared variable (as with dynamic scheduling and default chunk size 1), the degradation point is moved to a larger number of processors and the degradation is not that steeply.

4.2 Synchronization

We tested barriers, critical sections, locks, and reduction operations.

4.2.1 Barrier

Efficiency of a barrier implementation plays an important role in the OpenMP programming model, as (unless overwritten) after every work sharing construct and every parallel region a barrier is done implicitly. Fig. 3 shows the performance of the barrier operation on the various computers.

operation	T90	J90	O2000	R50
lock init	0.587	1.083	1.108	0.088
lock destroy	0.802	1.574	0.966	0.065
lock set	1.279	2.108	0.314	0.491
lock unset	1.299	2.121	0.123	0.501
lock test-and-set	1.183	1.900	0.234	0.395
critical section	3.367	8.528	0.412	0.880

Table 1: Performance of lock routines without contention (times in μs)

The T90, J90, and Origin2000 show similar curves but only with a different offset. Only the R50 has a steeper curve as the number of processors grows.

4.2.2 Critical Sections and Locks

All measurements for synchronization constructs where only one thread is involved (i.e. without contention) were calculated doing 10.000 operations of the corresponding type by a single thread and dividing the elapsed time by 10.000. The time to initialize, destroy, set, test-and-set, and unset a lock without contention for that lock can be done within a small amount of time on all systems (Tab. 1). The same is true for going through a critical section without contention. The Cray systems have a fairly large overhead for the non-competing synchronization operations.
Fig. 4 shows performance numbers when processors are competing for a lock. Here, all processors execute in a parallel region a loop with 10.000 locks/unlocks (each thread). The time shown is the total time divided by that 10.000, i.e. how long it takes for a processor in average to execute one op-

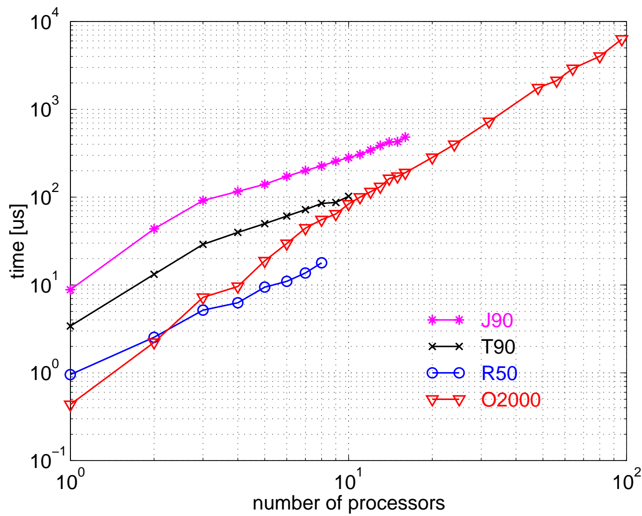

Figure 4: Performance of lock routines under contention.

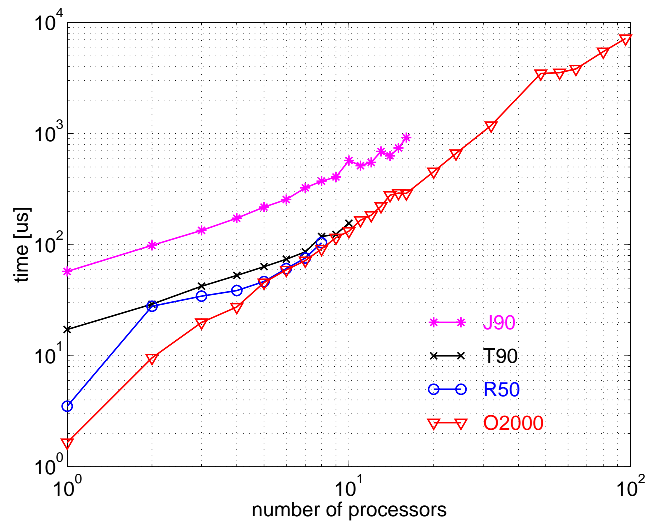

Figure 5: Performance of reduction operation (10 scalars).

eration under contention. Performance numbers for going through a critical section under contention are very similar to those shown in Fig. 4.

Here, the R50 shows a good performance over the whole range of processors, while the Origin2000 system has more problems as the number of processors grow. Also, on the Origin2000 there is an increase going from 2 processors on a board to 3 processors which have to communicate through the interconnection network.

4.2.3 Reductions

OpenMP allows only the reduction of scalar variables, a severe restriction to programmers as already pointed out in [BS98]. We tested the reduction of 1 scalar variable and the reduction of 10 scalar variables in a simple parallel loop (statically scheduled) where the number of iterations equals the number of threads available in the parallel region. Both reduction loops were executed 10.000 times and the resulting time for each loop divided by 10.000. Fig. 5 shows the results for 10 scalar variables, the results for 1 scalar variable are similar with the number of processors.

4.3 Data Handling

4.3.1 Firstprivate

To initialize private variables with the values of the original object, the *FIRSTPRIVATE* clause may be given to certain constructs. We specified a 10 KB large variable to be initialized in a parallel region. Fig. 6 shows the timings for that construct (including the overhead for the parallel region).

Comparing that with Fig. 1 (which shows the overhead for a parallel region without *FIRSTPRIVATE*) one can see that

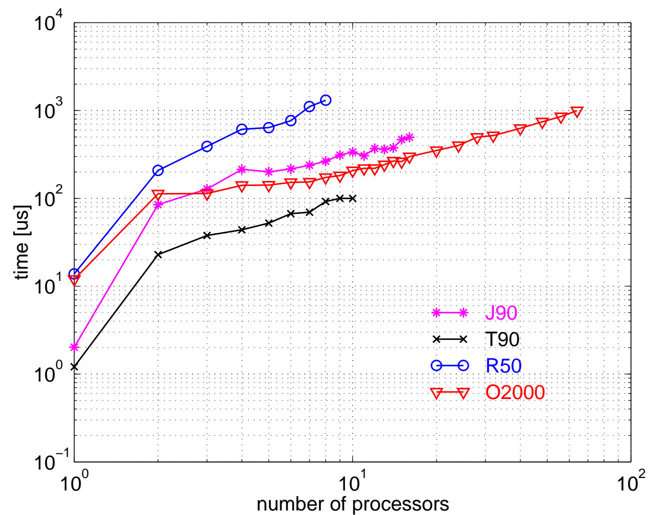

Figure 6: Parallel region with *FIRSTPRIVATE*-initialization of a variable (10 KB).

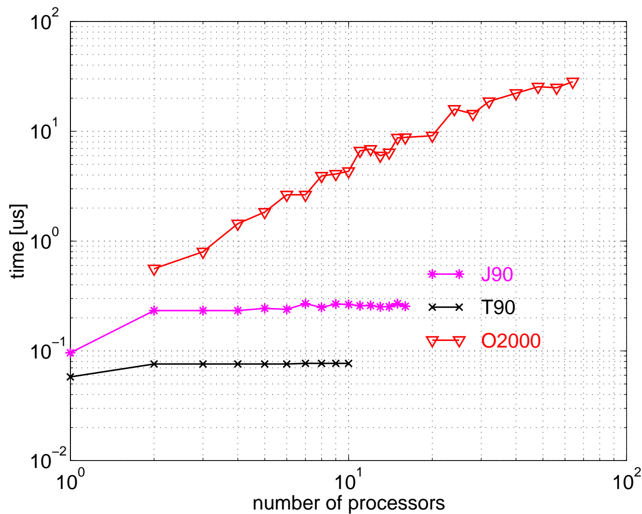

Figure 7: Flush of a variable (10 KB).

the vector machines (T90, J90) show only a small increase in time which can be contributed to the fast memory subsystem. On the R50, the overhead for a parallel region is already fairly high, such that the memory operations don't show too much effect. On the Origin2000, there is a large increase in time compared to the parallel region without the *FIRSTPRIVATE*-clause.

4.3.2 Memory Synchronization with Flush

The *FLUSH*-directive identifies synchronization points at which thread-visible variables synchronize with the global memory to ensure a consistent view of the memory. We specified at the *FLUSH*-directive a 10 KB large shared variable. A scetch of the code executed by each processor in a parallel region is shown here (this was actually iterated 10 times; *me* is the thread number):

```
data(me+1) = data(me)
data(me) = data(me-1)
!$OMP FLUSH(data)
```

On the IBM R50, the *flush*-directive is not implemented, therefore numbers for that system are missing. Fig. 7 shows the performance as a function of the number of processors.

On the UMA-machines (T90, J90) the flush operation is actually a no-op as no caches need to be flushed or data fetched from other processors' memory; if at all, only registers have to be written back to memory. On the cc-NUMA machine (Origin2000), the communication caused by the memory references to the shared variable combined with the *FLUSH* increases the time as the number of participating processors grow.

5 Related Work

As OpenMP is supported by the major computer vendors in their products only recently, little work has been published so far on performance results for OpenMP. In [BS98], first results on the NAS BT benchmark on a Origin2000 system were given.

6 Conclusion

OpenMP performs reasonably well on all tested systems. Some of the problems we have seen occurred only in special cases and might be solved with slight modifications in OpenMP library routines. The overhead for starting up a parallel region was fairly high with the exception of the Origin2000, and programs which fork a parallel region for every fine-grained parallel loop might have performance problems. Also, the scalability of the dynamic scheduling algorithms with fine-grained loops has shown to be a problem on the Origin2000.

Data placement and processor locality of data in non-UMA systems is an important aspect, which we have left out in our discussion as this aspect is highly application dependent. There are tools available to gather information on the memory performance [Ber99] which might help to optimize data locality although there are no language constructs in OpenMP to guide the compiler in generating processor locality.

During our tests we had to overcome some small but bothering problems with the OpenMP compilers we used. Some of the compilers had bugs, or didn't implement all directives/run time routines, or the name of environment variables were different than specified in the OpenMP API. We hope that future versions of all compilers implement the full API.

7 Acknowledgments

We would like to thank Reiner Vogelsang (SGI) for his support on SGI and CRAY systems.

References

[ACD+96] Christiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, pages 18–28, February 1996.

[Ber99] PCL – the performance counter library version 1.2. http://www.fz-juelich.de/zam/PCL/, July 1999.

[BG95] Rudolf Berrendorf and Michael Gerndt. SVM Fortran reference manual version 1.4. Technical Report KFA-ZAM-IB-9510, Research Centre Juelich, Germany, April 1995.

[BS98] Ragnhild Blikberg and Tor Sørenvik. Early experiences with OpenMP on the Origin 2000. In *Proc. 4th European SGI/CRAY MPP Workshop*, pages 166–177, IPP, Garching, Germany, September 1998.

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, pages 46–55, January 1998.

[Hig97] High Performance Fortran Forum. *High Performance Fortran Language Specification*, 2.0 edition, January 1997.

[Ope99] OpenMP Fortran interpretations version 1.0. http://www.openmp.org/, April 1999.

[PK87] Constantine D. Polychronopoulos and David J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Computers*, C-36(12):1425–1439, December 1987.