Scientific Programming 11 (2003) 125–131
IOS Press

An OpenMP compiler benchmark

Matthias S. Müller
HLRS, University of Stuttgart, Allmandring 30, D-70550 Stuttgart, Germany
Tel.: +49 711 685 8038; Fax: +49 711 6787 626; E-mail: mueller@hlrs.de

Accepted January, 2002

Abstract. The purpose of this benchmark is to propose several optimization techniques and to test their existence in current OpenMP compilers. Examples are the removal of redundant synchronization constructs, effective constructs for alternative code and orphaned directives. The effectiveness of the compiler generated code is measured by comparing different OpenMP constructs and compilers. If possible, we also compare with the hand coded "equivalent" solution. Six out of seven proposed optimization techniques are already implemented in different compilers. However, most compilers implement only one or two of them.

1. Introduction

Implementations of OpenMP are available on almost every shared memory platform. The portability of applications with OpenMP directives is therefore one of the strong points of this standard. Others are the advantages of shared memory programming in general: it allows an incremental approach to a fully parallel program, and it is also possible to switch between serial and parallel execution during runtime. This avoids the overhead introduced by the parallelism whenever a serial execution is faster.

A disadvantage of OpenMP is, that you need a compiler to generate the parallel code. Since the goal of parallel programming is to achieve higher performance, the further acceptance of OpenMP will strongly depend on compiler optimization techniques especially in the field where OpenMP has its possible benefits as described above. The importance of benchmarks to measure performance is reflected by many application benchmarks [8,10,9,11] and also the well known OpenMP Microbenchmarks [1]. To fully evaluate the quality of optimizing compilers a combination of both has to be used [3]. Microbenchmarks measure specific constructs directly, and can thus be used to drive the development of OpenMP environments. The focus of the OpenMP Microbenchmarks [1] is the runtime library, while this benchmark concentrates on the OpenMP compiler itself. It tries to avoid the architectural dependency of a direct measurement of synchronization and scheduling times and measures isolated OpenMP related optimizations directly, without introducing the complex behavior of a complete application. The focus of a runtime library benchmark [1,7] are questions like the scaling behavior of a barrier or reduction directive. Within this compiler benchmark the focus is whether overhead due to unnecessary synchronization or parallelization can be totally avoided.

2. Benchmarks

To test whether an optimization technique is already integrated in current compilers and to judge the efficiency of the proposed manual solutions, several compilers have been used. It should be noted that the major motivation for this survey is not to judge the compiler quality, but to check whether is is reasonable to expect the incorporation of the proposed techniques into the compiler. Due to the rapid development the results presented here are just an incomplete snapshot and results of different compiler versions may vary strongly.

The compilers from PGI [6] (version 3.4), Hitachi, NEC, SGI (version 7.30) and SUN (cc version 5.3, f90 version 6.2) are compilers producing native code, whereas the Omni [4] (Version 1.3) and guide [2] (Version 4.0) compiler are front ends to native compilers.

With one exception all constructs use the work load of Table 1. The fields `a`, `b` and `c` are double precision

ISSN 1058-9244/03/$8.00

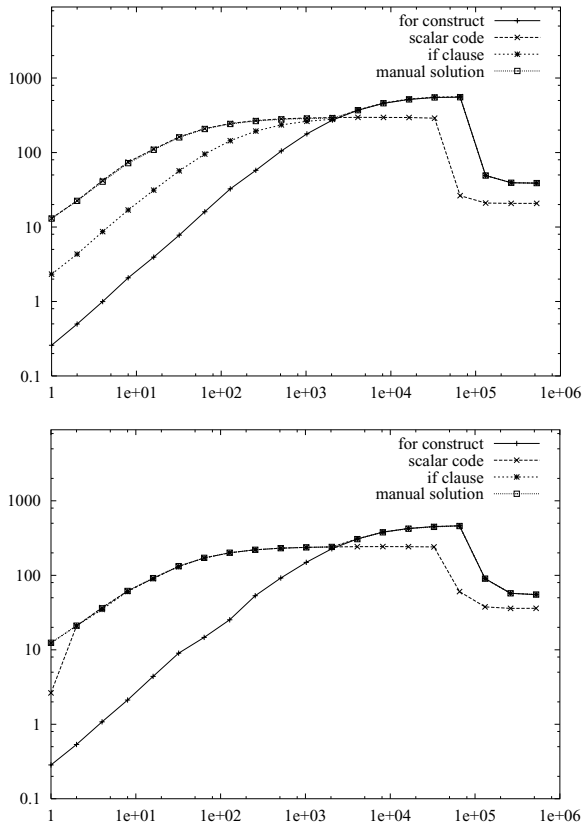

Fig. 1. Performance of alternative code in million iterations per second vs. loop length. The codes of the Guide compiler delivers the same performance as the manual solution with Fortran (bottom). With C the performance of the compiler generated code is worse (top).

arrays. The outer loop count is adjusted to guarantee a minimum runtime with a minimum execution count of ten repetitions. Some magic tries to avoid that the optimizer removes the outer loop. Here a zero is added to some array elements. The value of `offset` is read from file or passed as command line argument. Care has been taken that the optimization is not affected by the different aliasing prerequisites of Fortran and C. The inner loop count `length` is changed to measure the size dependent performance. This is not only important for the constructs that allow alternative code generation for small loop counts, but also provides useful informations for other situations. It shows where the parallelization efforts pays of, and whether or not there is a memory bottleneck. Where parallel and scalar performance is compared, the inner loop count is fixed, i.e. the iterations per thread vary. Since only relative performance is relevant, the performance is given in million loop iterations per second in this paper (see Figs 1 and 3). Where the overhead is more interesting than the performance, the absolute time spent inside the outer loop is measured (see Fig. 2). As long as the overhead is large enough, this time should reach a constant value for a short inner loop length. This time is taken as the overhead time. A comparison with the simple scalar loop gives an estimation of the clock accuracy.

Table 1
Workload of the benchmark loops

C

```
for (n = 0; n < count; n + +) {
      for (i = 0; i < length; i + +) {
        a [i] = b [i] + c [i];
      }
      /* fake the optimizer: */
      b [n] + = offset* a [n];
      c [n] + = offset* a [n];
}
```

Fortran

```
      DO 10 n = 1, count
            DO 20 i = 1, length
                  a (i) = b (i) + c (i)
      20 CONTINUE
C       fake the optimizer:
            b (n) = b (n) + offset* a (n)
            c (n) = c (n) + offset* a (n)
10       CONTINUE
```

2.1. Overhead of thread start-up

This part of the benchmark first of all checks whether Fortran and C compiler use the same thread implementation. It also checks how expensive it is to re-open a parallel region, providing hints whether threads are closed at the end of a parallel section or put into some form of waiting state. Most implementations seem to use the later approach and create the threads at the start of the program or first parallel section, and close them at the end of program. The SUN C compiler is the only example, where re-opening a parallel region is much more expensive than synchronizing and re-using existing threads.

In principle there should be no difference in performance between the two languages. Table 2 shows, that the overhead of thread creation is mostly dominated by the architecture. There are only small differences between Fortran and C compilers. Under certain circumstances the Hitachi compiler fails to create efficient solutions for parallel constructs, which makes some of the later tests not applicable. E.g. there is no benefit from parallel region merging, since two `parallel do` construct are more efficient than two `for` constructs merged in one parallel region.

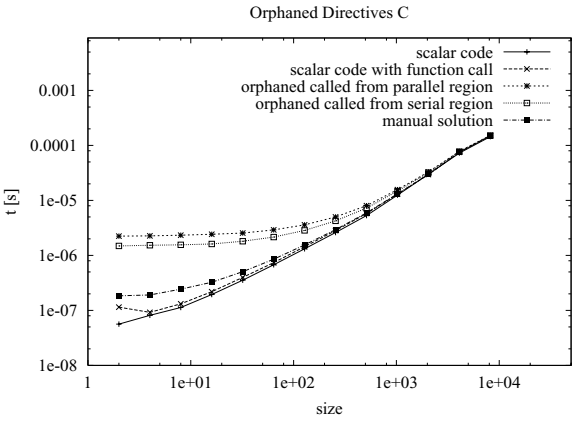

Fig. 2. Overhead of orphaned directives for the SGI compiler.

Table 2
Overhead of `parallel for` construct, compared to `for` construct with or without barrier

Language	C		
	parallel for	for + barrier	for
Compiler	[μs] (cycles)	[μs] (cycles)	[μs] (cycles)
PC PGI	2.2 (2200)	2.5 (2500)	1.5 (1500)
PC Omni	3.0 (3000)	1.8 (1800)	1.2 (1200)
HP guide	15 (8250)	7.0 (3850)	4.0 (2200)
SGI	11 (3200)	11 (3500)	9 (2650)
SUN	180 (140000)	4.2 (3200)	3.3 (2500)
SR8K	1.9 (490)	2.4 (600)	1.7 (560)
SX5	–	–	–
Language		Fortran	
	parallel for	for + barrier	for
Compiler	[μs] (cycles)	[μs] (cycles)	[μs] (cycles)
PC PGI	2.6 (2600)	2.4 (2400)	1.5 (1500)
PC Omni	3.0 (3000)	1.6 (1600)	1.1 (1100)
HP guide	14 (7700)	6.7 (3700)	3.7 (2000)
SGI	10 (3000)	11 (3300)	8.7 (2600)
SUN	8.3 (6200)	4.4 (3300)	3.2 (2400)
SR8K	2.0 (500)	3.2 (800)	2.4 (600)
SX5	22 (5500)	10 (2500)	7.4 (1850)

2.2. Alternative code

Once the overhead to create threads is known, the programmer can decide to avoid the overhead of parallel execution for small work load. For this purpose OpenMP contains the `if` clause. A parallel region with the `if` clause is only executed in parallel if the condition is true, otherwise the code is serialized. This dynamic behavior can be used to maximize performance. The goal is to have the performance of the serial code if it is faster than the parallel. To decide whether this goal is achieved the performance of the serial and parallel version is supplied. In addition, a manual solution is shown in Table 3. It is expected that the `if` clause is not much more than a convenient short hand notation for this manual solution. However, the exact definition

Table 3
Manual solution for alternative code

C

```
    if    (condition)    {
#pragma    omp    parallel
          {
             /*    code    */
          }
    }
    else    {
       /*    code    */
    }
```

Fortran

```
          IF    ( condition )    THEN
!$OMP  PARALLEL
C              CODE

!$OMP  END  PARALLEL
       ELSE
C              CODE
       END    IF
```

in the OpenMP standard [5] is that the code is "serialized": it should behave as if the construct is executed by a team of threads of size one. The difference between serial and serialized execution affects the behavior of OpenMP runtime library calls and outlining issues like `private` variables. This may introduce some overhead and the compiler needs to analyze to what extend the code makes use of constructs that behave different in serial and serialized execution.

Many compilers don't perform this optimization (see Tables 6 and 7. One example are the KAI compilers, where the KAI Guidef77 compiler produces code that is competitive with the manual solution, whereas the Guidec compiler produces code that is much slower (see Fig. 1).

One disadvantage the manual solution is the resulting code bloat. However, the same argument is true for techniques like inlining, where methods have been developed to balance speed and code size.

2.3. Orphaned directives

An orphaned directive is an OpenMP directive that does not appear in the lexical extent of a parallel construct, but lies in the dynamic extent. If such a worksharing construct is not enclosed dynamically within a parallel region the OpenMP standard states "it is treated as though the thread that it encounters it were a team of size one". This allows the user to write code that can be used in a parallel and serial context. This could be useful for library routines that can run in parallel if called from within a parallel region. One question is,

Table 4
Manual solution for orphaned directives

C

```
if (omp_in_parallel()) {
#pragma omp for private (i)
  for (i=0; i<length; i++) {
    a [i] = b [i] + c [i];
  }
}
else {
  for (i=0; i<length; i++) {
    a [i] = b [i] + c [i];
  }
}
```

Fortran

```
IF ( omp_in_parallel() ) THEN
! $OMP DO
      DO 10 i=1, length
        a (i) = b (i) + c (i)
10    CONTINUE
! $OMP END DO
  ELSE
    DO 20 i=1, length
      a (i) = b (i) + c (i)
20  CONTINUE
  END IF
```

whether there will be a overhead if the code is called from a serial context.

Several versions are compared against each other:

scalar: Performance of scalar code.

scalar with function call: Performance of scalar code with function call. The called function contains the working loop. This is done, because many compiler put parallel regions inside functions. This version checks whether a potential performance reduction is caused by the introduced additional call.

orphaned parallel: The working loop contains orphaned OpenMP directives. The loop is called from a parallel region with a team of one thread.

orphaned serial: The working loop contains orphaned OpenMP directives. The loop is called from a serial region.

orphaned manual: Finally, the hand coded version of Table 4 is used.

All tests are performed with the number of threads set to one. Any performance differences between scalar and parallel execution should therefore be due to the overhead of the implementation. The compiler should generate code that is faster or equal to the manual solution, depending on whether or not a call to the OpenMP runtime library function `omp_in_parallel` is necessary to check if the routine is called from a parallel region.

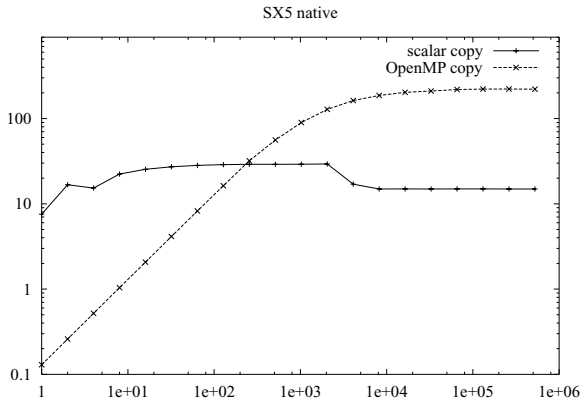

Fig. 3. Performance of copy operations in million iterations per second vs. loop length. OpenMP directives may help to optimize serial code. Both versions run with one thread. The OpenMP directive allows to vectorize the code.

With the PGI and SUN compilers it makes no difference whether the working function is called from a serial region or from a parallel region with a team of one thread. With the exception of the Hitachi compiler, the manual solution always results in the best performance, although there is an additional call to `omp_in_parallel`(). Figure 2 shows that the overhead of orphaned directives is always huge.

In principle the compiler could generate code without any overhead. For any function `foo` with orphaned directives, it could generate an additional function `foo_scalar` containing serial code and a function `foo_parallel` with parallel code. Depending whether the function `foo` is called inside or outside a parallel region the corresponding function call is substituted. The function `foo` with a solution equal to the manual solution could be provided to maintain link compatibility.

2.4. Removal of redundant synchronization

Three tiny examples check whether the OpenMP compiler removes redundant synchronization.

2.4.1. Parallel region merge

This is simply a concatenation of two `parallel for` directives. The execution time of one parallel region with two work-sharing constructs is compared to the time for two parallel regions with one construct each. An optimizing compiler should merge the two parallel regions. It is surprising that the Guide C compiler seems to implement this in contrast to the Guide F77 compiler.

Table 5
Overhead of orphaned directives in μs for various compilers

Compiler	scalar	scalar + f-call	orphaned from parallel	orphaned from serial	manual solution
PGI (C)	0.11	0.12	0.20	0.20	0.15
PGI (Fortran)	0.10	0.11	0.19	0.19	0.14
OMNI (C)	0.09	0.10	0.35	0.23	0.14
OMNI (Fortran)	0.10	0.11	0.25	0.35	0.16
HP guide (C)	0.05	0.04	2.24	0.47	0.19
HP guide (Fortran)	0.09	0.07	0.59	2.25	0.37
SGI (C)	0.06	0.11	2.24	1.48	0.18
SGI (Fortran)	0.12	0.13	1.49	2.28	0.21
SUN (C)	0.05	0.05	0.43	0.39	0.15
SUN (Fortran)	0.15	0.17	0.52	0.53	0.24
SR8K (C)	0.17	0.24	0.54	0.43	0.42
SR8K (Fortran)	0.17	0.30	0.36	0.40	0.40
SX5 (Fortran)	0.11	0.33	10.6	9.57	2.46

Table 6
Summary of optimization techniques of native compilers

Compiler	SUN		SGI		Hitachi		NEC	
Version	C	F77	C	F77	C	F77	C	F77
Optimization								
Alternative code competitive with manual solution	no	no	yes	yes	n.a.	n.a.	–	yes
Optimal Orphaned Directives	no	no	no	no	no	no	–	no
Orphaned Direct. competitive with manual solution	no	no	no	no	yes	yes	–	no
Parallel Region Merge	no	no	yes	yes	n.a.	n.a.	–	no
implicit NOWAIT due to independent blocks	no	no	no	no	no	no	–	yes
implicit NOWAIT due to end of region	yes	yes	no	no	n.a.	n.a.	–	yes
OpenMP directives used as optimization hint	no	no	no	no	yes	yes	–	yes

2.4.2. Removal of implicit `nowait` *at end of region*

The third is a `for` construct directly embedded in a `parallel` region. It is checked whether there is a difference between this version, a direct `parallel for` and a `for` construct with a `nowait` clause. An optimizing compiler should produce the same code for all three versions. Since there is no code between the for loop and the end of the parallel region, there is no need for more than one barrier. In the case of C++ there might be a lot of side effects, because variables go out of scope at the end of the `for` construct and destructors are called. For Fortran and C the question is, whether the information stored in these variables is still needed by other threads. Since all these variables are `private` by default, this is only possible if the content of a private variable is exposed to another thread. It is currently an open question whether this is legal in OpenMP. Nevertheless, the compiler still can analyze the code and remove the barrier if it is not required. The Guide and NEC Compiler show the desired behavior.

2.4.3. Removal of implicit `nowait` *between independent blocks*

It consists of two `for` constructs, that work on independent arrays. If the compiler detects that the two basic blocks are independent of each other, it may remove the barrier between the two loops. To increase the difference between a version with and without barrier a load imbalance is introduced in the first loop, that is balanced by a load imbalance in the second loop:

```
!$OMP DO PRIVATE(i)
      DO 20 i$=$1, length
         IF ( i .LT. length/2 ) THEN
C  Additional work for first half
            a(i) $=$ b(i)$+$c(i)
 &          $+$offset*sin(cos(b(i)))
         ELSE
            a(i) $=$ b(i)$+$c(i)
         END IF
 20   CONTINUE
!$OMP END DO
C  Same loop with additonal
C  work on second half
!$OMP DO PRIVATE(j)
      DO 30 j$=$1, length
         IF ( j .LT. length/2 ) THEN
            d(j) $=$ e(j)$+$f(j)
 &          $+$offset*sin(cos(e(j)))
         ELSE
            d(j) $=$ e(j)$+$f(j)
```

Table 7
Summary of optimization techniques of third part compilers

Compiler	Guide		Omni		PGI	
Language	C	F77	C	F77	C	F77
Optimization						
Alternative code competitive with manual solution	no	yes	no	no	almost	almost
Optimal Orphaned Directives	no	no	no	no	no	no
Orphaned Direct. competitive with manual solution	no	no	no	no	almost	almost
Parallel Region Merge	yes	no	no	no	yes	yes
implicit NOWAIT due to independent blocks	no	no	no	no	no	no
implicit NOWAIT due to end of region	yes	yes	no	no	no	no
OpenMP directives used as optimization hint	no	no	no	no	no	no

```
&          $+$offset2*sin(cos(e(j)))
        END IF
 30   CONTINUE
!$OMP END DO
```

The removal of this barrier requires a detailed code analysis. It is no surprise that the front end compilers (Omni and Guide) do not implement this, since it requires a detailed knowledge of the processor and architecture to perform possible optimizations. The NEC compiler shows that a native compiler can perform optimizations for this case, especially if the OpenMP directives are mapped to vendor specific directives, that will trigger parallelization and vectorization at the same time.

2.5. Benefit of openMP directives for other optimizations

This is maybe the most interesting test. The idea behind it is, that OpenMP directives may help the compiler to generate better code because he knows that certain preconditions are fulfilled. In that sense OpenMP could serve as a kind of portable pragmas for optimization. As an example workload we use the loop

```
      DO 20 i$=$1, length
         a(idx(i)) $=$ a(idx(i))$+$b(i)
 20   CONTINUE
```

Because the compiler does not know, how `idx(i)` looks like, he cannot assume that the different iterations of the loop are independent. The situation is different, if there is an `#pragma omp parallel for`. If the loop can be executed in parallel it is also subject to optimization techniques like software pipelining or vectorization. For large loop counts the performance of the parallel version executed on one thread should therefore exceed the performance of the serial code. Of course, the compiler can only trust the promise of the directive if OpenMP support is activated by the user. An incorrect OpenMP directive could cause the sequential version to behave differently from the one thread parallel version. During the tests there were only two cases (native compiler on Hitachi SR8000 and NEC SX5) where the performance of the loop with OpenMP directives was increased (see Fig. 3). However, a possible further cause might be, that it is impossible to increase the performance of a loop with indirect addressing on the tested architecture.

3. Conclusion

This small benchmark contains a collection of various optimization techniques that might be implemented in OpenMP compilers. The focus was to avoid architecture dependent techniques on one hand and to concentrate on features that are crucial to achieve maximum performance, especially in areas where the goal is to avoid the parallel overhead whenever a scalar execution is faster.

Both Fortran and C compilers lack many of the optimization techniques presented in this paper. However, six from seven proposed optimization techniques are already implemented in different compilers. This shows that it is a realistic demand to ask for the implementation of these techniques. The only feature that is not currently implemented in any compiler is an optimal solution for orphaned directives. In the case of alternative code generation and orphaned directives, the definition of “serialized” in the OpenMP standard [5] is one reason that makes it difficult for the compiler to generate efficient code. In both areas huge improvements are necessary to increase the performance of applications and parallel libraries. With the upcoming multi-threaded processors and the resulting reduced general thread overhead, the elimination of unnecessary overhead will become even more important, because they open the possibility for fine grained multi-threaded programming.

Acknowledgments

I would like to thank Sanjiv Shah (Intel) and Holger Berger (NEC) for their helpful comments and fruitful discussions.

References

[1] J.M. Bull, Measuring synchronization and scheduling overheads in OpenMP, In *First European Workshop on OpenMP*, 1999.
[2] http://www.kai.com.
[3] Kazuhiro Kusano, Shigehisa Satoh, and Mitsuhisa Sato, Performance evaluation of the omni openmp compiler, In *WOMPEI 2000*, Tokyo, Japan, Oct. 2000.
[4] OMNI OpenMP compiler, http://pdplab.trc.rwcp.or.jp/Omni.
[5] OpenMP Architecture Review Board, *OpenMP Specifications*, http://www.openmp.org/specs.
[6] http://www.pgroup.com.
[7] Achal Prabhakar, Vladimir Getow, and Barbara Chapman. Performance comparisons of basic openmp constructs, in: *High Performance Computing, 4th International Symposium, ISHPC 2002*, Hans P. Zima, Kazuki Joe, Mutsuhisa Sata and Yoshiki Seo adn Massaki Shimasaki, eds, vol. 2327 of *Lecture Notes in Computer Science*, pp. 413–424, Springer, 2002.
[8] RWCP, Openmp version of NAS parallel benchmarks, http://pdplab.trc.rwcp.or.jp/Omni/benchmarks/NPB/index.html.
[9] Mitsuhisa Sato, Kazuhiro Kusano, and Sigehisa Satoh, Openmp benchmark using PARKBENCH, in: *Proceeding of Second European Workshop on OpenMP*, Edinburgh, Scotland, U.K., Sept. 2000.
[10] SPEC, SPEComp 2001, http://www.spec.org.
[11] Daisuke Takahashi, Mitsuhisa Sato and Taisuke Boku, Performance evaluation of the hitachi sr8000 using openmp benchmarks, in: *High Performance Computing, 4th International Symposium, ISHPC 2002*, Hans P. Zima, Kazuki Joe, Mutsuhisa Sata and Yoshiki Seo adn Massaki Shimasaki, eds, Vol 2327 of *Lecture Notes in Computer Science*, pp 390–400. Springer, 2002.