OpenMP Optimization Techniques: Comparison of Fortran and C Compilers

Matthias Müller
HLRS, University of Stuttgart, Allmandring 30, D-70550 Stuttgart, Germany,
mueller@hlrs.de

July, 2001

Abstract

The purpose of this benchmark is to compare the optimization techniques in current Fortran and C OpenMP compilers. Examples are the removal of redundant synchronization constructs and effective constructs for alternative code. Although all tests focus on language independent techniques, there are differences in the implementation of the compilers.

1 Introduction

In the last years OpenMP [5] has found wide spread acceptance as portable programming model. Since the first release of the Fortran binding in 1997 the standard has been under active development. Traditionally a new version of the Fortran binding was available before the C binding. The purpose of this paper is to check whether this has influence on the quality of available OpenMP compilers. Especially the extent to which different optimization techniques are implemented might be influenced by the maturity of the compiler.

The importance of benchmarks to measure performance is reflected by many application benchmarks (e.g. [7, 8]) and also the well known OpenMP Microbenchmarks [1]. The problem of comparing Fortran and C compilers is, that application performance will be dominated by the scalar code quality and the results of the Microbenchmarks reflect the implementation of the OpenMP runtime library as well as the hardware architecture. This benchmark [3] tries to avoid the architectural dependency of a direct measurement of synchronization and scheduling times. It measures isolated OpenMP related optimizations directly without introducing the complex behavior of a complete application.

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
```

Table 1: Workload of the benchmark loops.

2 Benchmarks

To test whether the proposed optimization techniques are already active 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 goal is not to judge the compiler quality, but to check whether is is reasonable to expect the incorporation of the proposed techniques inside the compiler. Due to the rapid development the results presented here are just an incomplete snapshot and results of different compiler version may vary strongly.

The compilers from PGI[6] (Version 3.2-4), Hitachi, NEC and SGI (Version 7.3.1.1) 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 Tab. 1. The fields `a b` and `c` are double precision 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. 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 low 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.

2.1 Overhead of Thread Start-up

C			
Compiler	parallel for [μs](cycles)	for + barrier [μs](cycles)	for [μs](cycles)
PC PGI	2.0 (2000)	2.4 (2400)	1.4 (1400)
PC Omni	2.9 (2900)	1.6 (1600)	1.1 (1100)
HP guide	15 (8250)	7.0 (3850)	4.0 (2200)
SGI	11 (2100)	11 (2100)	10 (1950)
SR8K	1.8 (450)	2.6 (650)	1.7 (425)
SX5	-	-	-
Fortran			
Compiler	parallel for [μs](cycles)	for + barrier [μs](cycles)	for [μs](cycles)
PC PGI	2.1 (2100)	2.3 (2300)	1.3 (1300)
PC Omni	3.0 (3000)	1.6 (1600)	1.1 (1100)
HP guide	14 (7700)	6.7 (3700)	3.7 (2000)
SGI	11 (2100)	16 (8800)	11 (2100)
SR8K	2.0 (500)	3.2 (800)	2.4 (600)
SX5	22 (5500)	10 (2500)	7.4 (1850)

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

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 section, providing hints whether threads are closed at the end of a parallel section or put into some form of waiting state. Most implementations 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.

In principle there should be no difference in performance between the two languages. Tab. 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, this fact 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.

2.2 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
```

Table 3: Manual solution for 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 Tab. 3, it is expected that the `if` clause is not much more than a convenient short hand notation for this manual solution.

Surprisingly this is not true for many compilers. 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). It should be noted, that the manual implementation requires not more than a simple text replacement. This could easily be implemented by compilers that are front ends to native back end compilers.

2.3 Orphaned Directives

An orphaned directive is a OpenMP directive that does not appear in the lexical extent of a parallel construct, but lies in the dynamic extent. If such a work-sharing 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, 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 Tab. 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 wether 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.

With the PGI and Hitachi 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. Normally the performance within a parallel region is significantly worse. 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()`. Fig. 2 shows that the overhead of orphaned directives is always huge.

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
```

Table 4: Manual solution for orphaned directives.

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. 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.

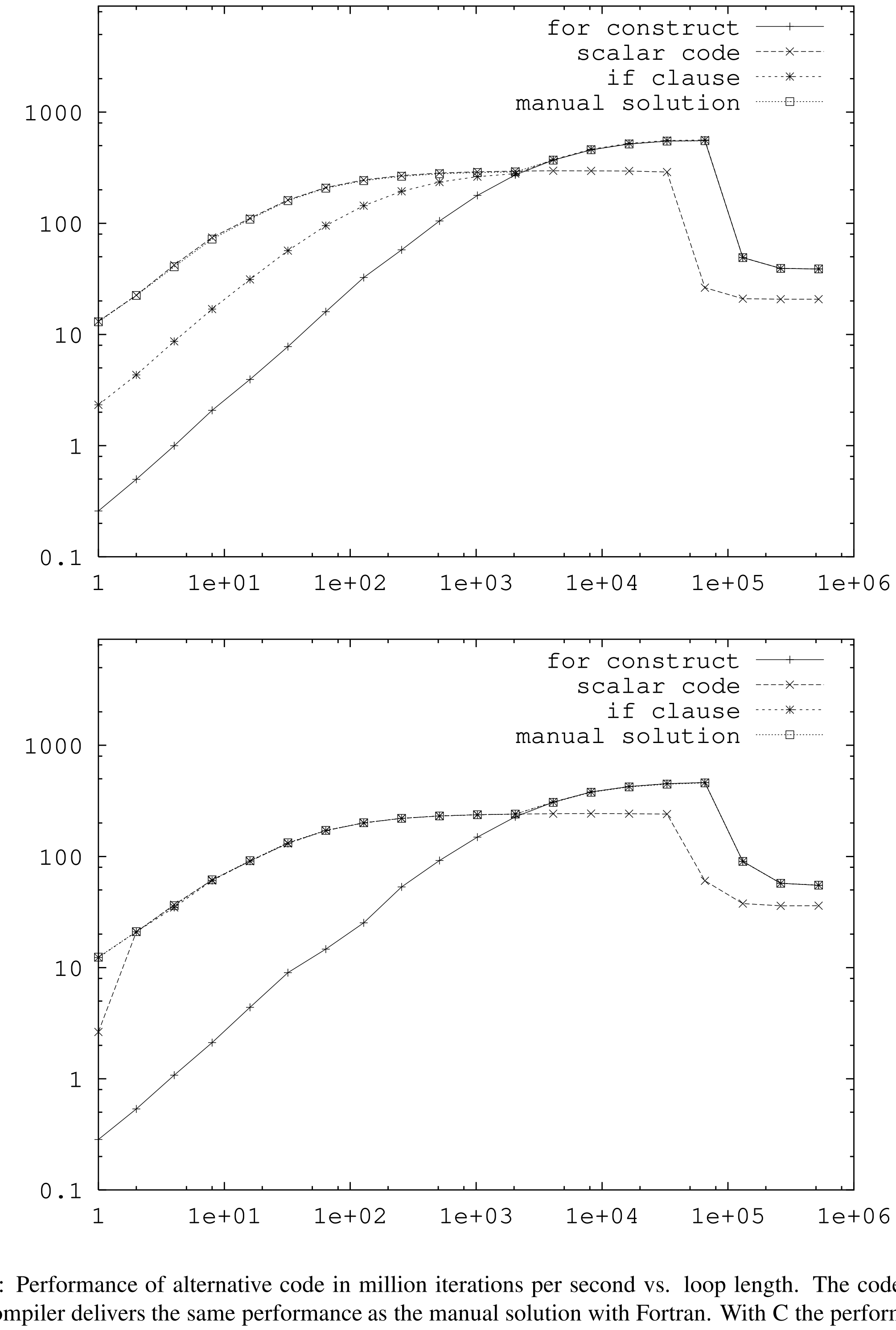

Figure 1: Performance of alternative code in million iterations per second vs. loop length. The codes of the Guide compiler delivers the same performance as the manual solution with Fortran. With C the performance of the compiler generated code is worse.

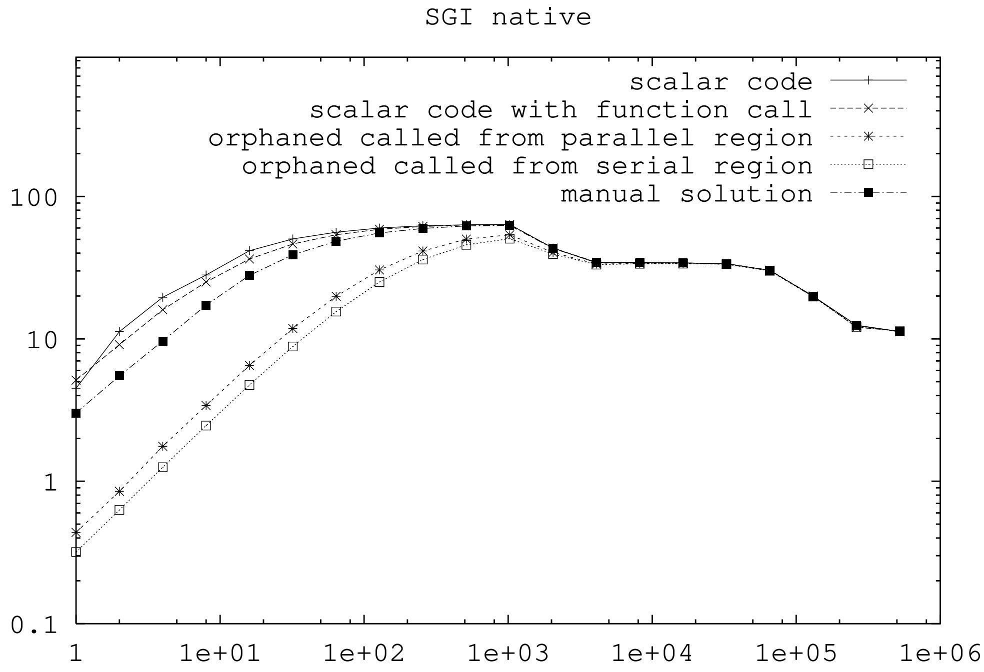

Figure 2: Performance of orphaned directives in million iterations per second vs. loop length.

2.4.2 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. The Guide and NEC Compiler show the desired behavior.

2.4.3 Implicit `nowait` at 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:

```
      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
C  Same loop with additonal
C  work on second half
```

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 detailled 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,

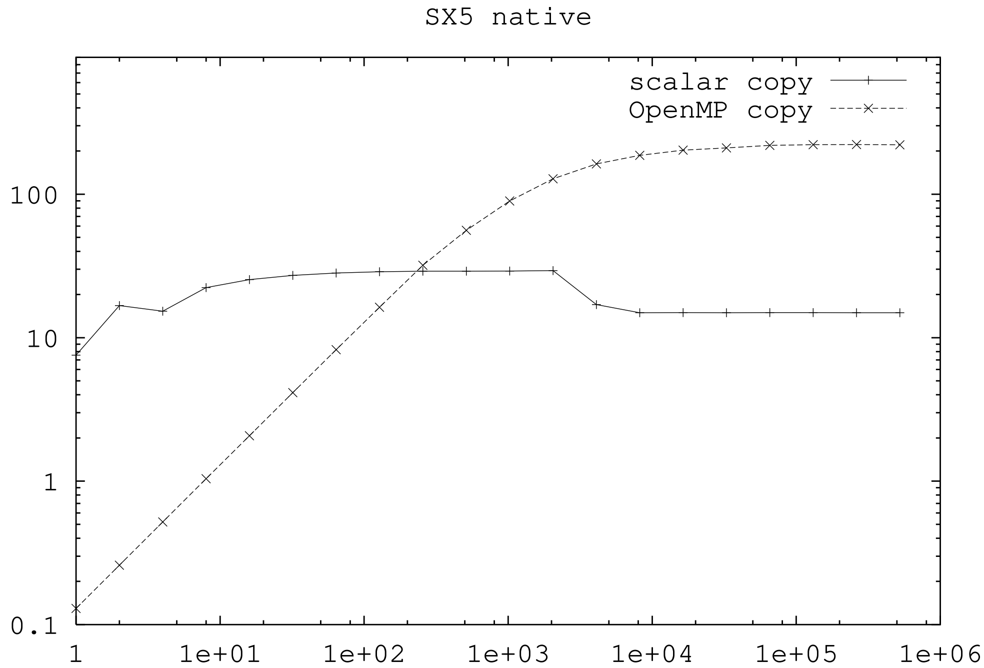

Figure 3: Performance of copy operations in million iterations per second vs. loop length. OpenMP directives may help to optimize serial code. Both versions run with one thread. The OpenMP directive allows to vectorize the code.

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. During the tests there was 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

From the six tested compilers only one shows different optimization techniques for C and Fortran. There seems to be no general advantage for either of the languages. Both Fortran and C compilers lack many possible optimization techniques. The extension of this benchmark to Fortran has shown that all proposed solutions exists in different compilers. The only feature that is not currently visible in a compiler is an optimal solution for orphaned directives. The area of orphaned directives is a field where huge improvements are necessary to increase the performance of applications and parallel libraries.

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] Matthias Müller. Some simple OpenMP optimization techniques. In R. Eigenmann and M.J. Voss, editors, *OpenMP Shared Memory Parallel Programming*, volume 2104 of *LNCS*, WOMPAT 2001, West Lafayette, IN, USA, 2001. Springer.

[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] RWCP. Openmp version of NAS parallel benchmarks. http://pdplab.trc.rwcp.or.jp/Omni/benchmarks/NPB/index.html.

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

Table 5: Summary of optimization techniques of native compilers.

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

Table 6: Summary of optimization techniques of front end compilers.

[8] SPEC. SPEComp 2001. http://www.spec.org.