Impact of OpenMP Optimizations for the MGCG Method

Osamu Tatebe[1], Mitsuhisa Sato[2], and Satoshi Sekiguchi[1]

[1] Electrotechnical Laboratory, 1-1-4 Umezono, Tsukuba, Ibaraki 305-8568 JAPAN
{tatebe, sekiguchi}@etl.go.jp
[2] Real World Computing Partnership, 1-6-1 Takezono, Tsukuba, Ibaraki 305-0032 JAPAN
msato@trc.rwcp.or.jp

Abstract. This paper shows several optimization techniques in OpenMP and investigates their impact using the MGCG method. MGCG is important for not only an efficient solver but also benchmarking since it includes several essential operations for high-performance computing. We evaluate several optimizing techniques on an SGI Origin 2000 using the SGI MIPSpro compiler and the RWCP Omni OpenMP compiler. In the case of the RWCP Omni OpenMP compiler, the optimization greatly improves performance, whereas for the SGI MIPSpro compiler, it does not affect very much though the optimized version scales well up to 16 processors with a larger problem. This impact is examined by a light-weight profiling tool bundled with the Omni compiler. We propose several new directives for further performance and portability of OpenMP.

1 Introduction

OpenMP is a model for parallel programming that is portable across shared memory architectures from different vendors[4, 5]. Shared memory architectures have become a popular platform and are commonly used in nodes of clusters for high-performance computing. OpenMP is accepted by vendors or research institutes for portable specification across shared memory architectures. Supercomputer vendors, compiler vendors and research institutes now provide OpenMP compilers.

OpenMP can parallelize a sequential program incrementally. To parallelize a **do** loop in a serial program, it is enough to insert a parallel region construct as an OpenMP directive just before the loop. Sequential execution is always possible unless a compiler interprets OpenMP directives.

This paper investigates how to optimize OpenMP programs and its impact. Each directive of OpenMP may cause an overhead. The overhead of each OpenMP directive can be measured by, for example EPCC OpenMP microbenchmarks[2], however, the important thing is how these overhead affect the performance of applications. In this paper, we focus the impacts of the OpenMP source-level optimization to reduce overhead by creating and joining a parallel region and eliminating unnecessary barrier for the MGCG method.

2 MGCG Method

The MGCG method is a conjugate gradient (CG) method with a multigrid (MG) preconditioner and is quite efficient for the Poisson equation with severe coefficient jumps[3, 7, 8]. This method is important for not only several applications but also benchmarking because MGCG includes several important operations for high-performance computing.

MGCG consists of MG and CG. MG exploits several sizes of meshes from a fine-grain mesh to a coarse-grain mesh. It is necessary to efficiently execute parallel loops with various loop lengths. CG needs an inner product besides a matrix-vector multiply and a daxpy. It is also necessary to efficiently execute a reduction in parallel.

The major difficulties for the parallelization of MGCG are the matrix-vector multiply, the smoothing method, restriction and prolongation. In this paper, we use only regular rectangular meshes and a standard interpolation. That is for not only avoiding complexity but also its wide applicability for applications such as computational fluid dynamics, plasma simulation and so on. Note that MG with standard interpolation results in a divergence when a coefficient has severe jumps, while MGCG converges efficiently.

3 Optimization Techniques in OpenMP

Parallelization using OpenMP is basically incremental and straightforward. All we should do is to determine the time-consuming and parallelizable **do** (or **for**) loops and to annotate them by an OpenMP directive for parallel execution. For example, the following code is parallelized using the `PARALLEL DO` directive and `REDUCTION` clause.

```
      IP = 0.0
!$OMP PARALLEL DO REDUCTION(+:IP)
      DO I = 1, N
          IP = IP + A(I) * B(I)
      END DO
```

This easy parallelization may cover a large number of programs including dusty deck codes and achieve moderate performance, while the performance strongly depends on the program, compiler and platform. Generally, it is necessary that a parallelized loop spends most of the execution time and it has enough loop length or plenty of computation to hide an overhead of OpenMP parallelization overhead, thread management, synchronization and so on.

To get much higher performance in OpenMP, reducing these parallelization overheads are necessary. In order to reduce these overheads, the following optimizations are necessary.

1. Join several parallel regions. Each parallel region construct starts parallel execution that may contain an overhead of thread creation and join. At the extreme, it is desirable for an entire program to be one big parallel region.

2. Eliminate unnecessary barriers. Each work-sharing construct implies a barrier (and a flush) at the end of the construct unless NOWAIT is specified. Note that the BARRIER directive is necessary to ensure that each thread accesses updated data.
3. Privatize variables if possible by specifying PRIVATE, FIRSTPRIVATE and LASTPRIVATE.
4. Determine a trade-off of parallel execution including parallelizing overhead and sequential execution. Parallel execution of a loop without enough loop length may be slower than serial execution, since parallel execution needs an overhead of thread management and synchronization. For several independent loops with short loop length, each loop may be parallelized using the SECTIONS construct.

Fig. 1 is an excerpt of the MGCG method written in OpenMP. In this case, a parallel region includes three work-sharing constructs and one subroutine call. The subroutine MG called in the parallel region, includes several work-sharing constructs as an orphaned directive. Orphaned directives are interpreted as a work-sharing construct when the subroutine is called within a parallel region.

The second work-sharing construct computes an inner product, and it includes a REDUCTION clause. The shared variable RR2 is summed up at the end of this work-sharing construct. In most cases, this kind of computation is a subroutine call. In this case, the subroutine includes an orphaned directive as follows:

```
      SUBROUTINE INNER_PRODUCT(N, R1, R, RR2)
!$OMP DO REDUCTION(+:RR2)
      DO I = 1, N
          RR2 = RR2 + R1(I) * R(I)
      END DO
      RETURN
      END
```

Since it is necessary to perform a reduction, RR2 must be shared within a parallel region.

The MGCG method is parallelized in two ways. One is a version that is parallelized naively. Basically, almost all DO loops are parallelized using a PARALLEL DO directive. We call this a *naive* version. The other is an *optimized* version that exploits the above optimization techniques. In the optimized version, the entire MGCG program is a parallel region. In Fig. 1, the first and the last statements are in a serial region. The optimized version uses a MASTER directive followed by a BARRIER directive to execute serially within the parallel region. This version also eliminates all unnecessary barriers by specifying NOWAIT.

```
      RR2 = 0

!$OMP PARALLEL PRIVATE(BETA)

*     /*****  MG Preconditioning  *****/

!$OMP DO
      DO I = 1, N
          R1(I) = 0.0
      END DO
      CALL MG(...)

*     /*****  beta = (new_r1, new_r) / (r1, r)  *****/

!$OMP DO REDUCTION(+:RR2)
      DO I = 1, N
          RR2 = RR2 + R1(I) * R(I)
      END DO
      BETA = RR2 / RR1

*     /*****  p = r1 + beta p  *****/

!$OMP DO
      DO I = 1, N
          P(I) = R1(I) + BETA * P(I)
      END DO

!$OMP END PARALLEL

      RR1 = RR2
```

Fig. 1. An excerpt of the MGCG method in OpenMP

4 Evaluation of Optimization Impact

4.1 Overhead of Creating a Parallel Region

To investigate an overhead to create and join a parallel region, we consider two kinds of programs, one has several parallel regions (Fig. 2) and the other has one large parallel region with several work-sharing constructs (Fig. 3). For small N and large M, the program of Fig. 2 shows an overhead to create and join a parallel region including an overhead of a work-sharing construct and that of Fig. 3 shows an overhead of a work-sharing construct. We evaluate these programs on an SGI Origin 2000 with 16 R10000 195MHz processors with a

```
      DO J = 1, M
!$OMP PARALLEL DO
          DO I = 1, N
              A(I) = A(I) + 2 * B(I)
          END DO
      END DO
```

Fig. 2. A naive program with several parallel regions

```
!$OMP PARALLEL
      DO J = 1, M
!$OMP DO
          DO I = 1, N
              A(I) = A(I) + 2 * B(I)
          END DO
      END DO
!$OMP END PARALLEL
```

Fig. 3. An optimized program with one parallel region and several work-sharing constructs

Table 1. Overhead of parallel region construct and work-sharing construct when $N = 8$ and $M = 100000$ using eight threads

	naive	optimized
MIPSpro	2.90	2.07
Omni	46.76	2.86

[sec.]

developing version of the RWCP Omni OpenMP compiler[1, 6] and the SGI MIPSpro compiler version 7.3. The current version 1.1 of the Omni compiler only supports pthreads for the IRIX operating system; however, this developing version exploits **sproc** for further efficient execution on IRIX. The sproc system calls create a new process whose virtual address space can be shared with the parent process.

Table 1 shows the overhead of parallel and work-sharing constructs when $N = 8$ and $M = 100000$ using eight threads. Overhead for a parallel region construct is quite small using the MIPSpro compiler as also reported by EPCC OpenMP microbenchmarks, while it is quite large using the Omni OpenMP compiler. The Omni compiler basically translates an OpenMP program to a C program using the Omni runtime libraries. To reduce the overhead for the parallel

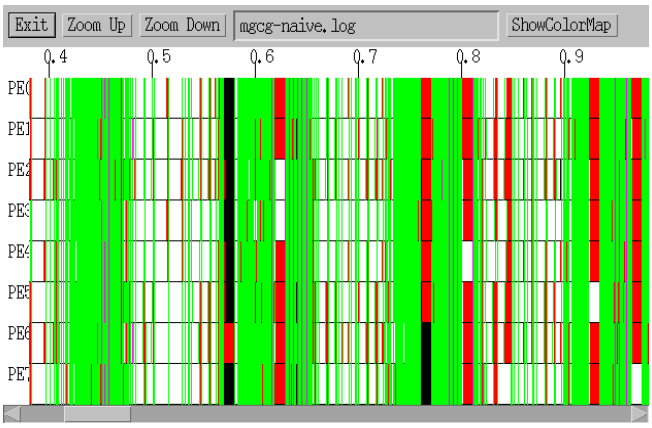

Fig. 4. Snapshot of `tlogview` for the naive version

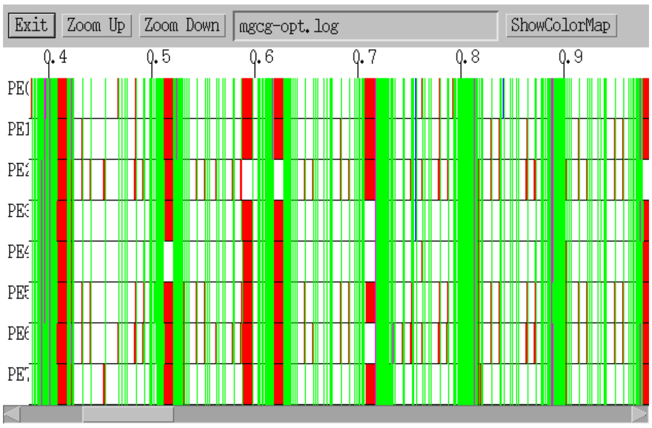

Fig. 5. Snapshot of `tlogview` for the optimized version

region construct, it is necessary to improve the performance of the runtime library for Origin 2000.

4.2 Performance Evaluation of MGCG Method

Profiling The RWCP Omni OpenMP compiler provides a very light-weight profiling tool and a viewer to investigate a behavior of parallel execution. All events related to OpenMP directives are time-stamped at runtime. Each event consists of only two double words, and the profiled log can be viewed by `tlogview`.

Naive and optimized versions of MGCG are profiled on an SGI Origin 2000 with the developing version of the Omni compiler. Figures 4 and 5 show profiled data of the MGCG program by `tlogview`. The white region denotes a parallel execution, the red (dark gray) region shows a barrier, the green (light gray) region shows a loop initialization, and the black region shows a serial part. The MGCG method roughly consists of two parts; CG and MG. The CG part includes a daxpy operation, an inner product and a matrix-vector multiply on the finest grid. Since each loop length is long, the CG part is white. The MG part includes a smoothing method (Red-Black Gauss-Seidel), restriction and prolongation in each grid level. Since MG needs operations on coarse grids that do not have long loop length, the MG part is almost green and red.

Figures 6 and 7 are snapshots of a finer scale. The upper picture of Fig. 6 shows the CG part and the lower picture shows the MG part. These two pictures show almost one iteration of MGCG. In this scale, a serial part stands out. Particularly, an overhead of MG part is conspicuous, since the MG part has loops with various loop length from long length to short length. On the other hand, the optimized version successfully reduces these overheads and serial part.

Performance Evaluation We evaluate naive and optimized MGCG programs using the Omni OpenMP compiler and the SGI MIPSpro compiler. We also compare these OpenMP programs with a program written in MPI. The MPI library is the SGI Message-passing toolkit version 1.3. Every program is compiled with the `-Ofast` optimization option.

Each program solves a two-dimensional Poisson equation with severe coefficient jumps. The two-dimensional unit domain is discretized by 512×512 meshes and 1024×1024 meshes. This kind of problem with severe coeffient jumps is difficult by MG, while it is quite efficiently solved by a combination of MG and CG.

Fig. 8 shows floating-point performance for the problem size 512×512. Using the Omni compiler, the floating-point performance of the optimized version is clearly better than that of the naive version, and it is almost double the performance for large numbers of processors. On the other hand, the difference of the two programs is quite small and both programs achieve good performance with the SGI MIPSpro compiler, since the overhead of a parallel region construct of the MIPSpro compiler is small. In both compiler cases, the performance does not scale with more than 8 processors because the problem size is not so large. The elapsed execution time of the optimized version is only 1.4 seconds with 8 processors.

The MPI program achieves quite good performance with 8 processors. In fact, each processor achieves 44 MFLOPS with 8 processors, while only 32 MFLOPS with 1 processor. The MGCG program parallelized by MPI needs approximately 32 MB of data when the problem size is 512×512. With 8 processors, each processor processes only 4 MB of data that fits into secondary cache. Unfortunately, OpenMP programs cannot take this advantage so much.

Fig. 9 is a result of 1024×1024. In this case, approximately 128 MB of data is necessary. Up to 8 processors, all programs but the naive version with the Omni compiler achieve almost the same good performance. With more processors, the

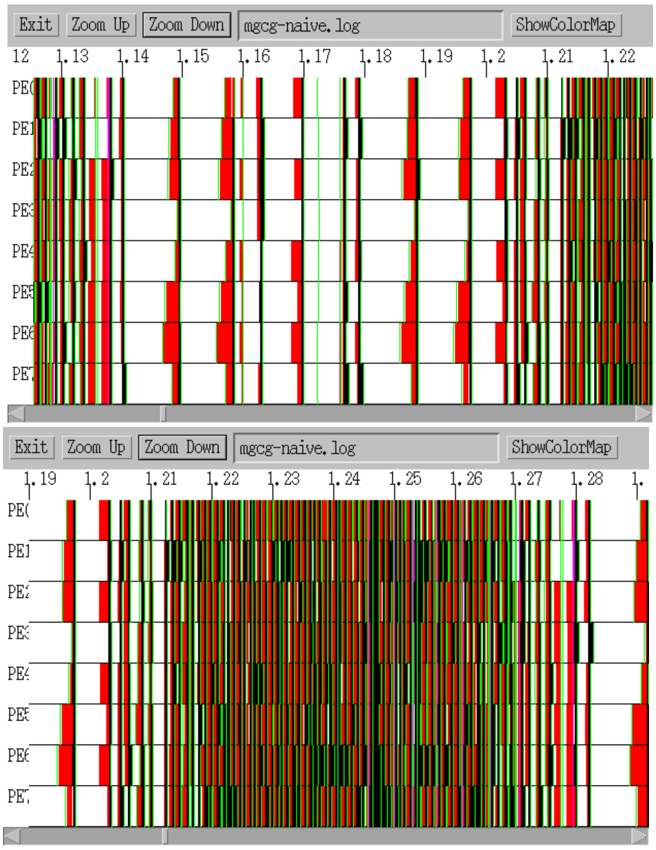

Fig. 6. Magnified snapshot of the naive version

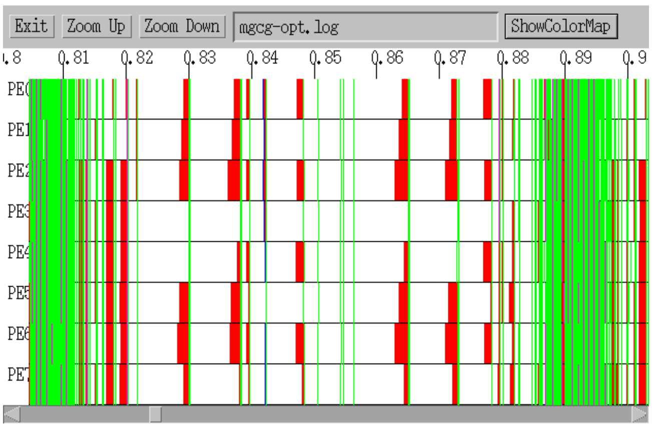

Fig. 7. Magnified snapshot of the optimized version

naive version with the MIPSpro compiler and the optimized version with the Omni compiler slightly degrade the performance, while the MPI program and the optimized version with MIPSpro scale well. That is because the ratio of computation on coarse grid, i.e., a loop with short loop length that tends to become the overhead of parallelization increases, when the number of processes increases. Comparing with the performance of the OpenMP programs of two problem sizes, 512×512 case is better than 1024×1024 case in the case of eight threads. This shows that a program parallelized by OpenMP can utilize memory locality slightly however it is no match for a program written in MPI.

5 Proposals for OpenMP Specification

When constant variables are privatized in a parallel region, both `FIRSTPRIVATE` and `LASTPRIVATE` need to be specified, since these constant variables become undefined at the end of the parallel region without `LASTPRIVATE`; however, this also may cause an overhead, since copying is necessary from a private variable to a shared variable at the end of the parallel region. This case is quite common. We propose a new data scope attribute clause; `READONLY`. This clause only copies from the shared variable to a private variable and ensures to maintain the shared variable after the parallel region.

We employed an iterative method for performance evaluation. The iterative method is quite sensitive to rounding error and the order of floating-point operations of reduction. Since the OpenMP specification allows any order of floating-point operations for reduction, an OpenMP program may compute a different inner product on successive execution even with the static loop scheduling, the same loop length and the same number of threads. Actually, even the number of iterations until convergence differs every time using the MIPSpro compiler. This may also happen with MPI since the MPI specification also allows any order of floating-point operations for reduction. The problem is that the difference with OpenMP is observed to be much larger than that with MPI. The situation is much worse using dynamic loop scheduling. Since reproducible results are quite important for debugging and testing, this should be controlled by a new directive or a new environmental variable.

6 Conclusions and Future Research

This paper showed several techniques to optimize OpenMP programs and their impact. Impact is quite dependent on compilers. The Omni compiler showed attractive and almost double the performance compared with the naive version. The SGI MIPSpro compiler showed little difference of performance with small number of processors, however, the optimized version scales well with more processors.

Even though the SGI Origin 2000 supports distributed shared memory, the program written in MPI performs better than the OpenMP program especially with 8 processors and the problem size 512×512. That is because the MPI program exploits locality of memory explicitly. How to exploit the locality is also a big future research of the OpenMP compiler and the OpenMP specification.

Acknowledgment

We are grateful to every member of TEA group of Electrotechnical Laboratory (ETL), Real World Computing Partnership, University of Tsukuba and Tokyo Institute of Technology for discussion about semantics and performance of OpenMP programs. We give great thanks for Dr. L. S. Blackford at University of Tennessee, Knoxville for precious comments and proofreading. We also give great thanks for Dr. K. Ohmaki, director of the computer science division at ETL for supporting this research.

References

1. *http://pdplab.trc.rwcp.or.jp/pdperf/Omni/home.html.*
2. BULL, J. M. Measuring synchronisation and scheduling overheads in OpenMP. In *Proceedings of First European Workshop on OpenMP (EWOMP'99)* (1999).
3. KETTLER, R. Analysis and comparison of relaxation schemes in robust multigrid and preconditioned conjugate gradient methods. In *Multigrid Methods*, W. Hackbusch and U. Trottenberg, Eds., vol. 960 of *Lecture Notes in Mathematics*. Springer-Verlag, 1982, pp. 502–534.
4. OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP C and C++ Application Program Interface*, October 1998. version 1.0.
5. OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP Fortran Application Program Interface*, November 1999. version 1.1.
6. SATO, M., SATOH, S., KUSANO, K., AND TANAKA, Y. Design of OpenMP compiler for an SMP cluster. In *Proceedings of 1st European Workshop on OpenMP (EWOMP'99)* (1999), pp. 32–39.
7. TATEBE, O. The multigrid preconditioned conjugate gradient method. In *Proceedings of Sixth Copper Mountain Conference on Multigrid Methods* (April 1993), NASA Conference Publication 3224, pp. 621–634.
8. TATEBE, O., AND OYANAGI, Y. Efficient implementation of the multigrid preconditioned conjugate gradient method on distributed memory machines. In *Proceedings of Supercomputing '94* (November 1994), IEEE Computer Society, pp. 194–203.

This article was processed using the LaTeX macro package with LLNCS style

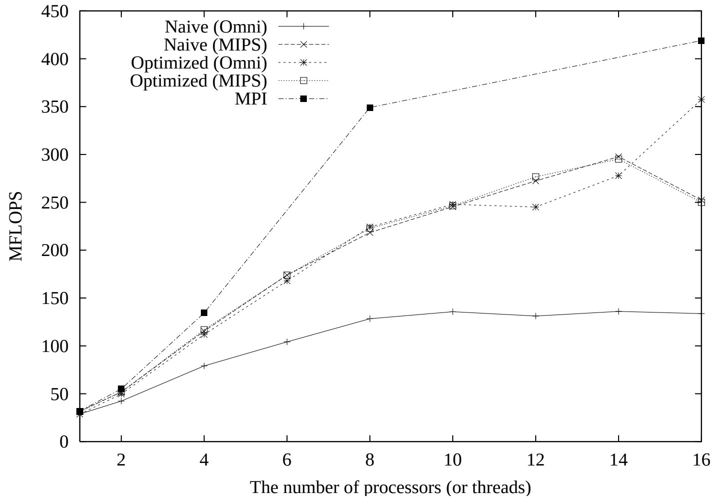

Fig. 8. Floating-point performance of MGCG (512 × 512)

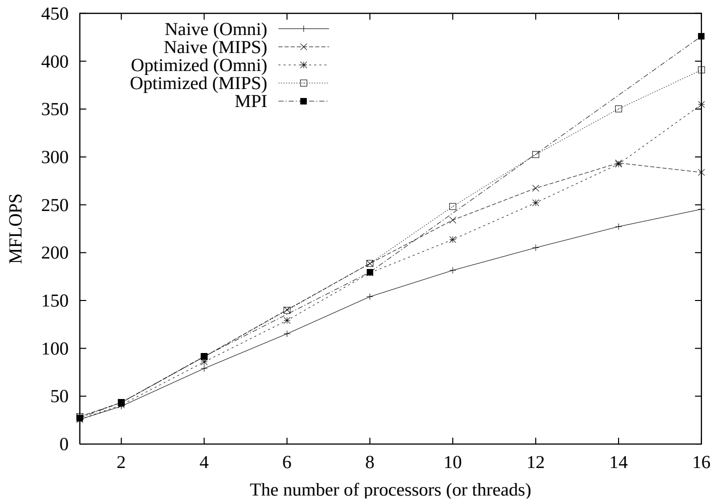

Fig. 9. Floating-point performance of MGCG (1024 × 1024)