Performance of OSCAR Multigrain Parallelizing Compiler on SMP Servers

Kazuhisa Ishizaka[†], Takamichi Miyamoto[†], Jun Shirako[†], Motoki Obata[‡], Keiji Kimura[†], and Hironori Kasahara[†]

[†]Department of Computer Science,
Advanced Chip Multiprocessor Research Institute,
Waseda University
3-4-1 Ohkubo, Shinjuku-ku, Tokyo, 169-8555, Japan
{ishizaka,miyamoto,shirako,kimura,kasahara}@oscar.elec.waseda.ac.jp
[‡]System Development Laboratory, Hitachi Co.Ltd.
m-obata@sdl.hitachi.co.jp

Abstract. This paper describes performance of OSCAR multigrain parallelizing compiler on various SMP servers, such as IBM pSeries 690, Sun Fire V880, Sun Ultra 80, NEC TX7/i6010 and SGI Altix 3700. The OSCAR compiler hierarchically exploits the coarse grain task parallelism among loops, subroutines and basic blocks and the near fine grain parallelism among statements inside a basic block in addition to the loop parallelism. Also, it allows us global cache optimization over different loops, or coarse grain tasks, based on data localization technique with inter-array padding to reduce memory access overhead. Current performance of OSCAR compiler is evaluated on the above SMP servers. For example, the OSCAR compiler generating OpenMP parallelized programs from ordinary sequential Fortran programs gives us 5.7 times speedup, in the average of seven programs, such as SPEC CFP95 tomcatv, swim, su2cor, hydro2d, mgrid, applu and turb3d, compared with IBM XL Fortran compiler 8.1 on IBM pSeries 690 24 processors SMP server. Also, it gives us 2.6 times speedup compare with Intel Fortran Itanium Compiler 7.1 on SGI Altix 3700 Itanium 2 16 processors server, 1.7 times speedup compared with NEC Fortran Itanium Compiler 3.4 on NEC TX7/i6010 Itanium 2 8 processors server, 2.5 times speedup compared with Sun Forte 7.0 on Sun Ultra 80 UltraSPARC II 4 processors desktop workstation, and 2.1 times speedup compare with Sun Forte compiler 7.1 on Sun Fire V880 UltraSPARC III Cu 8 processors server.

1 Introduction

Currently, multiprocessor architectures are widely used for chip multiprocessors to desktop workstations, mid-range servers and high-end servers. However, the gap between peak and effective performance of a multiprocessor system is getting larger with the increase of the number of processors. Although, efficent parallel programs are important to improve effective performance, software development on a multiprocessor requires special knowledge and experience in parallel programming and the long duration. To improve effective performance, cost-performance and software productivity of multiprocessor systems , strong automatic parallelizing compilers are required.

So far, in automatic parallelizing compilers for multiprocessor systems, the loop parallelization techniques have been used. For the loop parallelization, various data dependence analysis techniques[1–3] such as GCD, Banerjee's inexact and exact tests, OMEGA test[4], symbolic analysis[5] and dynamic dependence test and program restructuring techniques have been researched and also employed in compiler products available in the market. As research compilers, Polaris compiler[3] exploits loop parallelism by using inline expansion of subroutine, symbolic propagation, array privatization[6], run-time data dependence analysis[7] and interprocedural access region test[8] and SUIF compiler uses strong inter-procedure analysis[9] unimodular transformation and data locality optimization[10] including affine partitioning[11]. However, by those research efforts, the loop parallelization techniques are reaching maturity.

In light of this fact, new generation parallelization techniques like multigrain parallelization are needed to overcome the limitation of loop parallelization. NANOS compiler[12] based on Parafrase2 has been trying to exploit the multi-level parallelism including the coarse grain parallelism by using extended OpenMP API. The OSCAR multigrain parallelizing compiler[13] exploits the coarse grain task parallelism among loops, subroutines and basic blocks[14], and the near fine grain parallelism among statements inside a basic block[15] in addition to the conventional loop parallelism among iterations.

The OSCAR compiler has been developed as a core module of Japanese Millennium Project IT21 "Advanced Parallelizing Compiler project". The advanced parallelizing compiler project is a three years project started in FY 2000 to develop an automatic parallelizing compiler to improve effective performance, cost-performance and software productivity for shared memory multiprocessor architectures used for chip multiprocessors to high-end servers.

This paper describes the OSCAR multigrain parallelizing compiler and its performance on off-the-shelf SMP servers, such as IBM pSeries 690 24 way Power4 high-end SMP server and Sun Fire V880 8 UltraSPARC III Cus server, Sun Ultra80 4 UltraSPARC IIs desktop workstation, NEC TX7/i6010 8 way Itanium 2 server, SGI Altix 16 way Itanium 2 server using OSCAR compiler's OpenMP platform-free backend.

2 OSCAR Multigrain Parallelizing Compiler

The OSCAR compiler exploits multigrain parallelism, namely, coarse grain parallelism, loop level parallelism and near fine grain parallelism from the whole source program. As shown in Figure 1, the OSCAR compiler consists of the Fortran frontend, middle path for multigrain parallelization and several backends for different target machines such as the OSCAR chip multiprocessor[16], SMP servers supporting OpenMP and cluster systems supporting MPI. In the multigrain parallel processing for SMP servers treated in this paper, the compiler generates coarse grain tasks called "macro-tasks" such as loops, subroutines and basic blocks, analyzes parallelism among the macro-tasks by the earliest executable condition analysis based on control and data dependence analysis, decomposes macro-tasks and data for cache or distributed shared memory optimization by loop aligned decomposition, schedules macro-tasks to threads or thread groups statically or dynamically considering data locality, generates parallel code with OpenMP API using "One-time single level thread generation".

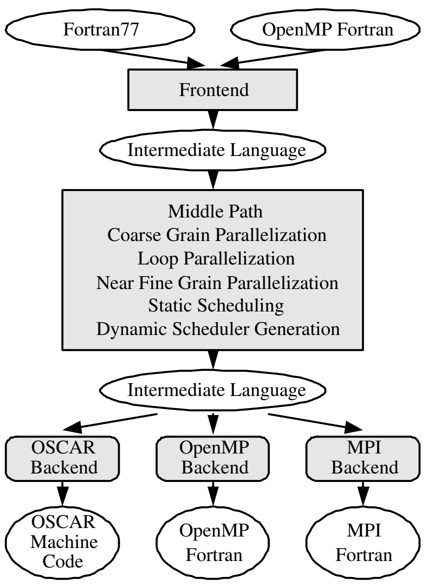

Fig. 1. Flow of OSCAR Multigrain Compiler

2.1 Macro-Task Generation

In the multigrain parallelization, a source program is decomposed into three kinds of coarse grain tasks, or macro-tasks, namely block of pseudo assignment statements(BPA) repetition block(RB), subroutine block(SB). Also, macro-tasks are generated hierarchically inside of a sequential repetition block and a subroutine block as shown in Figure 2.

2.2 Earliest Executable Condition

After the generation of macro-tasks, compiler analyzes data flow and control flow among macro-tasks in each layer or each nested level. Next, to extract

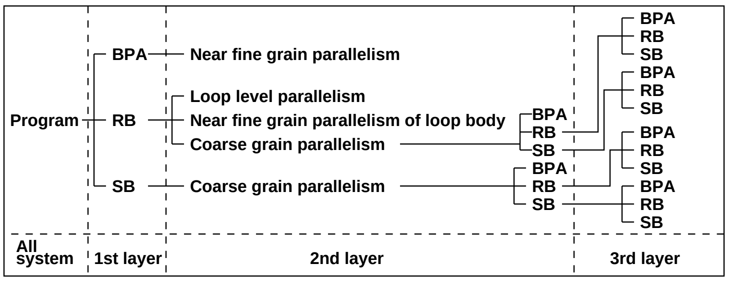

Fig. 2. Hierarchical Macro Task Definition

parallelism among macro-tasks, the compiler analyzes Earliest Executable Condition(EEC)[13] of each macro-task. EEC represents the conditions on which macro-task may begin its execution earliest.

EEC of macro-task is represented in macro-task Graph (MTG) as shown in Figure 3. In macro-task graph, nodes represent macro-tasks. A small circle inside nodes represents conditional branches. Solid edges represent data dependencies. Dotted edges represent extended control dependencies. Extended control dependency means ordinary control dependency and the condition on which a data dependent predecessor macro-task is not executed. A solid arc represents that edges connected by the arc are in AND relationship. A dotted arc represents that edges connected by the arc are in OR relation ship.

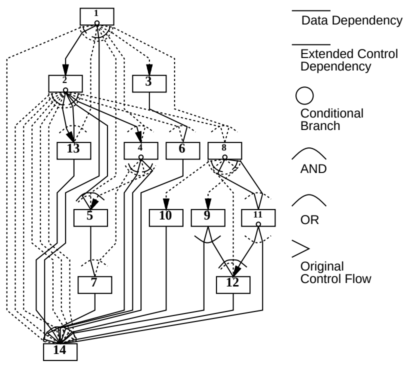

Fig. 3. An Example of Macro-Task Graph

2.3 Macro-Task Scheduling

In the coarse grain task parallel processing, static scheduling and dynamic scheduling are used for an assignment of macro-tasks to threads.

If a macro-task graph has only data dependencies and is deterministic, static scheduling is selected. In the static scheduling, an assignment of macro-tasks to threads is determined at compile time by the scheduler in the compiler. Static scheduling is useful since it allows us to minimize data transfer and synchronization overhead without runtime scheduling overhead.

If a macro-task graph has control dependencies, the dynamic scheduling is selected to handle runtime uncertainties like conditional branches. The scheduling routines for the dynamic scheduling are generated by the compiler and inserted into a parallelized program with macro-task code.

2.4 Global Cache Optimization

In the coarse grain task parallel processing, macro-tasks can begin its execution when Earliest Executable Condition is satisfied without regard for the program order in the original source code. Therefore, the compiler decides the execution order of macro-tasks so that macro-tasks accessing the same data can be executed on the same processor consecutively to optimize cache usage among the tasks.

Loop Aligned Decomposition
To avoid cache misses among the macro-tasks, Loop Aligned Decomposition (LAD) [17] is applied to the loops that access data larger than cache size. LAD divides the loops, or macro-tasks, into partial loops with the smaller number of iterations so that data size accessed by the divided loops is smaller than cache size.

The partial loops are defined as coarse grain tasks and the Earliest Executable Condition analysis is applied again. The partial loops connected by data dependence edge on the macro task graph are grouped into "Data Localizable Group(DLG)". The macro-tasks inside a DLG are assigned to the same processor as consecutively as possible statically or dynamically.

In the macro-task graph of Figure 4(a), it is assumed that macro-tasks 2, 3 and 7 are parallel loops accessing the same shared array variables exceeding cache size. In this example, the loops are divided into four partial loops by the LAD technique. For example, the macro-task 2 in Figure 4(a) is divided into the macro-tasks 2_A through 2_D shown in Figure 4(b). Also, the DLGs like DLG_A composed of macro-task 2_A, 3_A, 7_A are defined.

Consecutive Execution of Data Localizable Group
In the original program, macro-tasks are executed in the increasing order of the node number on the macro-task graph. For example, the execution order of macro-tasks 2_A to 3_D is 2_A, 2_B, 2_C, 2_D, 3_A 3_B, 3_C, 3_D. In this order, macro-tasks in the same DLG are not executed consecutively.

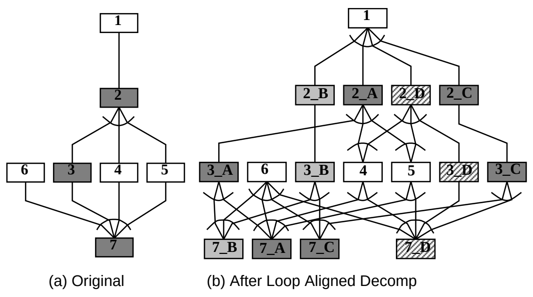

Fig. 4. Example of Loop Align Decomposition

However, the earliest executable condition shown in Figure 4(b) means that macro-task 3_B, for example, can be executed immediately after macro-task 2_B because macro-task 3_B depends on macro-task 2_B only.

In the proposed cache optimization scheme, the task scheduler assigns macro-tasks inside a DLG to the same processor as consecutively as possible[18] in addition to the "critical path" priority used by the both static and dynamic scheduling. Figure 5 shows a schedule when the proposed cache optimization is applied to macro-task graph in Figure 4(b) for a single processor. As shown in Figure 5, the macro-task 2_B, 3_B, 8_B in DLG_B and the macro-task 2_C, 3_C, 7_C in DLG_C are executed consecutively to use the data on cache optimally.

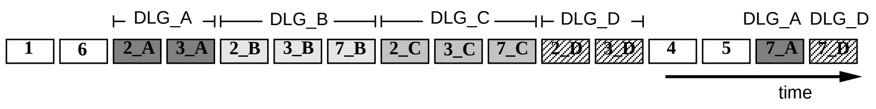

Fig. 5. Consecutive Execution of Data Localizable Group

Reduction of Cache Conflict Misses

The Loop Aligned Decomposition and The Consecutive Execution of a DLG enable the shared data to be reused before cache line replacement. However, if the data accessed by the macro-tasks in a DLG share the same cache line on a cache as shown in Figure 6(a), the data may be removed from the cache before reuse because of line conflict misses even though the amount of data accessed in a DLG is smaller than the cache size. To reduce conflict misses, the OSCAR compiler analyzes data layout on the cache and applies inter-array padding to remove line conflicts among data in a DLG on the cache as shown in Figure 6(b)[19].

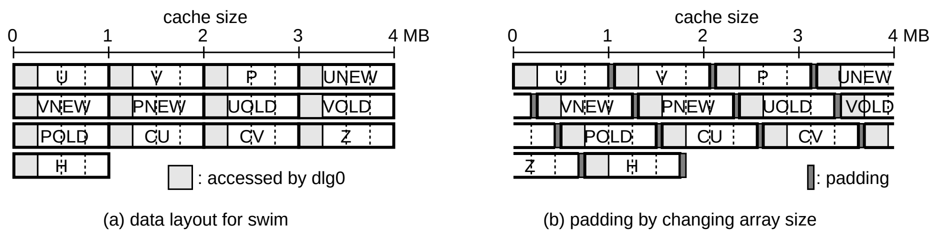

Fig. 6. Padding to Reduce Conflict Messes

2.5 OpenMP Code Generation

The OSCAR compiler generates a parallelized Fortran program with OpenMP directives. A generated code image for eight threads is shown in Figure 7 for the macro-task graph in 8. In this figure, eight threads are generated only once by the OpenMP PARALLEL SECTIONS directives and the generated threads join at the end of program by using the "One-time single level thread generation scheme"[13]. In this example, the static scheduling is applied to the 1st layer of MTG in Figure 8. In this case, the eight threads are grouped into two thread groups each of which has four threads as shown in Figure 7 to process MTG1 having parallelism of "2" estimated by the compiler in Figure 7. MT1_1 and MT1_3 are assigned to thread group0 composed of four threads and MT1_2 is assigned to thread group1. When static scheduling is applied like this program layer, the compiler generates different program codes for each OpenMP SECTION according to the static schedule as shown in Figure 8. The macro-tasks assigned to each thread groups are processed in parallel by threads inside each thread group by using static scheduling or dynamic scheduling hierarchically. In this example, MT1_2 in Figure 7 assigned to thread-group1 in Figure 8 is processed by four threads in parallel using the centralized dynamic scheduling scheme. In Figure 8, threads 5, 6 and 7 execute some of sub macro-tasks like MT1_2_1, MT1_2_2 and so on, which are generated inside MT1_2 in Figure 7, assigned by thread 4 working as the centralized dynamic scheduler. Also, MT1_3 in Figure 7 shows a code image for distributed dynamic scheduling in which scheduling codes are inserted before and after task codes. In this case, MT1_3 is decomposed into sub macro-tasks 1_3_1 through 1_3_4 as shown in Figure 7 and assigned to thread group0_0 and 0_1 defined inside thread group0. In this example, the thread group0_0 and 0_1 consists of two threads respectively.

3 Performance of OSCAR Compiler

This section describes the performance evaluation of the OSCAR multigrain parallelizing compiler on different multiprocessor servers available on the market using popular benchmark programs such as, tomcatv, swim, su2cor hydro2d, mgrid, applu, turb3d from SPEC CFP95 Benchmarks.

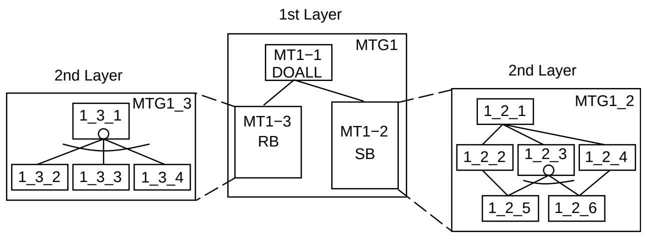

Fig. 7. Sample Macro Task Graph having 3 Layers

3.1 Performance on IBM pSeries 690 24 way SMP Server

Figure 9 shows the performance of OSCAR compiler on the IBM pSeries 690 high-end UNIX server with 24 processors, or 12 Power4 chips. Left bars show speedups by automatic parallelization of the IBM XL Fortran compiler version 8.1 against sequential executions of the XLF compiler. Right bars show speedups by OSCAR compiler against the same sequential executions of the XLF compiler. The speedup by OSCAR compiler was mesured using the OpenMP backend of the OSCAR compiler. The generated OpenMP parallelized programs from sequential programs were compiled by the XL Fortran compiler and executed on pSeries 690 server. The numbers above bars show parallel execution times and the numbers below the benchmark program names are sequential execution times by the XLF compiler.

For tomcatv, the sequential execution time was 23 seconds and the fastest parallel execution time up to 24 processors by XLF compiler was 19 seconds and OSCAR compiler was 2.9 seconds. In other words, OSCAR compiler gave us 7.9 times speedup against sequential execution, 6.6 times speedup compared with the XLF compiler. Also, OSCAR gave us 17.8 times speedup compared with sequential execution (7.2 times compared with XLF compiler automatic parallelization) for swim, 3.0 times (3.0 times) for su2cor, 9.2 times (6.4 times) for hydro2d, 8.3 times (3.9 times) for mgrid, 2.0 times (1.7 times) for applu, 12.2 times (11.5 times) for turb3d.

In the average of the above 7 programs, the XLF compiler gave us 1.5 times speedup against sequential execution and OSCAR gave us 8.6 times speedup against sequential execution, namely 5.7 times compared with the parallel execution of the XLF.

3.2 Performance on SGI Altix 3700 16 Itanium 2 SMP Server

This section shows the performance of OSCAR compiler on SGI Altix 3700 server with 16 Itanium 2 processors. Figure 10 shows the performance of Intel Fortran Itanium compiler revision 7.1 and OSCAR compiler using up to 16 processors.

OSCAR compiler gave us 5.7 times speedup against sequetial execution (5.7 times speedup against Intel compiler automatic parallelization) for tomc atv, 8.0

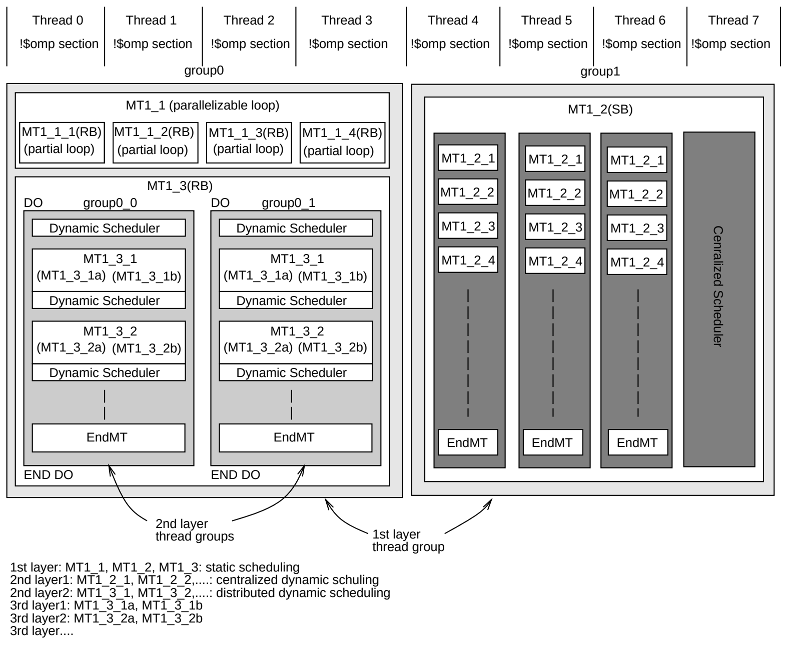
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
!$omp section
group0
group1
MT1_1 (parallelizable loop)
MT1_1_1(RB) (partial loop)
MT1_1_2(RB) (partial loop)
MT1_1_3(RB) (partial loop)
MT1_1_4(RB) (partial loop)
MT1_3(RB)
DO
group0_0
group0_1
Dynamic Scheduler
MT1_3_1
(MT1_3_1a) (MT1_3_1b)
MT1_3_2
(MT1_3_2a) (MT1_3_2b)
EndMT
END DO
MT1_2(SB)
MT1_2_1
MT1_2_2
MT1_2_3
MT1_2_4
Cenralized Scheduler
2nd layer thread groups
1st layer thread group
1st layer: MT1_1, MT1_2, MT1_3: static scheduling
2nd layer1: MT1_2_1, MT1_2_2,.....: centralized dynamic schuling
2nd layer2: MT1_3_1, MT1_3_2,.....: distributed dynamic scheduling
3rd layer1: MT1_3_1a, MT1_3_1b
3rd layer2: MT1_3_2a, MT1_3_2b
3rd layer....

Fig. 8. Generated Code Image using OpenMP (8 threads)

times (2.1 times) for swim, 1.4 times (1.4 times) for su2cor, 3.2 times (3.2 times) for hydro2d, 2.6 times (1.7 times) for mgrid, 1.2 times (1.1 times) for applu, 5.5 times (5.4 times) for turb3d and 3.9 times (2.6 times) in the average for the above 7 programs.

3.3 Performance on NEC TX7/i6010 8 Itanium 2 SMP Server

This section shows the performance of OSCAR compiler on NEC TX7/i6010 server with 8 Itanium 2 processors. Figure 11 shows the performance of NEC Fortran Itanium Compiler revision 3.4 and OSCAR compiler.

OSCAR compiler gave us 4.4 times speedup against sequetial execution (3.3 times speedup against NEC Compiler automatic parallelization) for tomcat v, 7.6 times (1.1 times) for swim, 1.1 times (1.1 times) for su2cor, 3.9 times (2.5 times) for hydro2d, 2.9 times (0.9 times) for mgrid, 1.4 times (1.2 times) for applu, 5.8 times (5.8 times) for turb3d and 3.9 times (1.7 times) in the average for the above 7 programs.

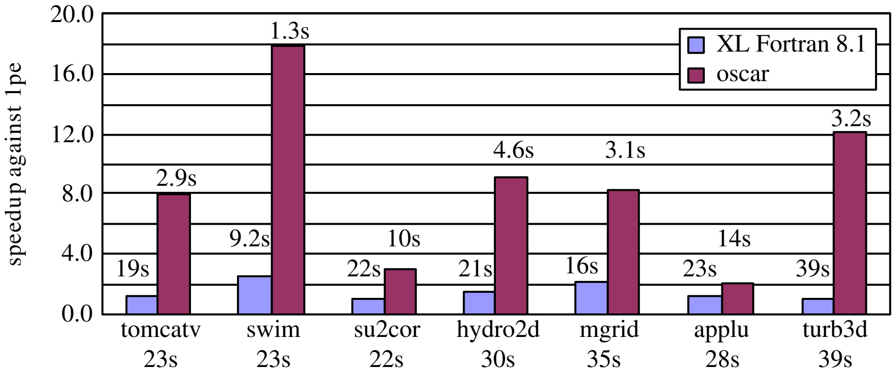

Fig. 9. Performance on IBM pSeries 690 24 Processors Server

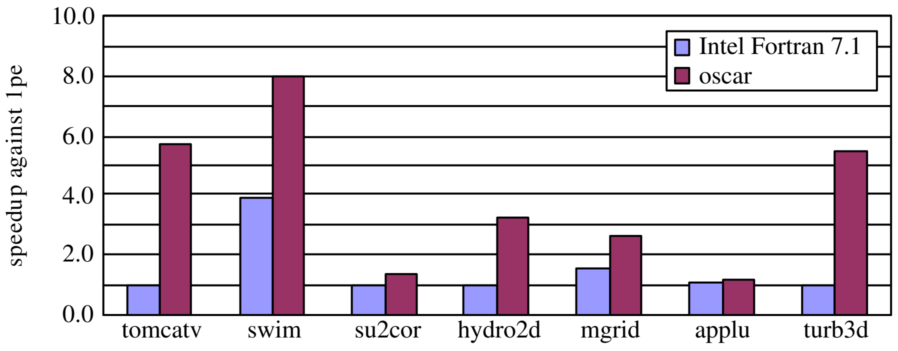

Fig. 10. Performance on Altix 3700 16 Processors Server

3.4 Performance on Sun Ultra 80 4 UltraSPARC II SMP Workstation

This section shows the performance of OSCAR compiler on Sun Ultra 80 desktop workstation with 4 UltraSPARC II. Figure 12 shows the performance of Sun Forte 7.1 compiler and OSCAR compiler on Ultra 80.

OSCAR compiler gave us 6.3 times speedup against sequetial execution (5.2 times speedup against Forte compiler automatic parallelization) for tomcatv, 9.2 times (5.5 times) for swim, 2.5 times (1.6 times) for su2cor, 4.3 times (2.6 times) for hydro2d, 4.7 times (1.0 times) for mgrid, 1.2 times (0.9 times) for applu, 4.1 times (3.8 times) for turb3d and 4.6 times (2.5 times) in the average for the above 7 programs.

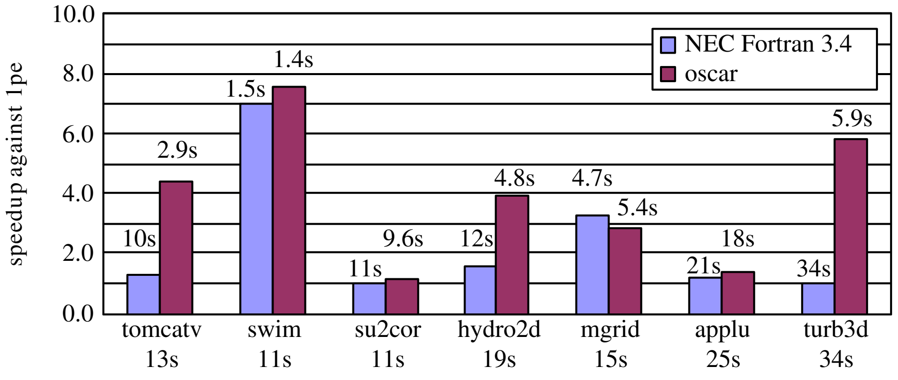

Fig. 11. Performance on NEC TX7/i6010 8 Processors Server

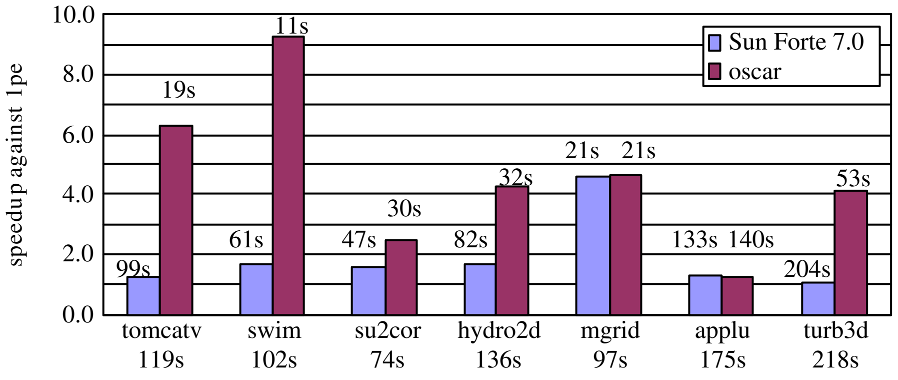

Fig. 12. Performance on Sun Ultra 80 4 Processors Workstation

3.5 Performance on Sun Fire V880 8 UltraSPARC III Cu SMP Server

This section shows the performance of OSCAR compiler on Sun Fire V880 server with 8 UltraSPARC III Cu processors. Figure 13 shows speedups and execution times of benchmark programs compiled by Sun Forte 7.1 compiler and OSCAR compiler on the V880.

OSCAR compiler gave us 4.9 times speedup against sequetial execution (2.1 times speedup against Forte compiler automatic parallelization) for tomcatv, 18.6 times (2.5 times) for swim, 3.6 times (1.3 times) for su2cor, 4.7 times (1.6 times) for hydro2d, 4.1 times (1.1 times) for mgrid, 1.2 times (1.0 times) for applu, 7.1 times (7.1 times) for turb3d and 6.3 times (2.1 times) in the average for the above 7 programs.

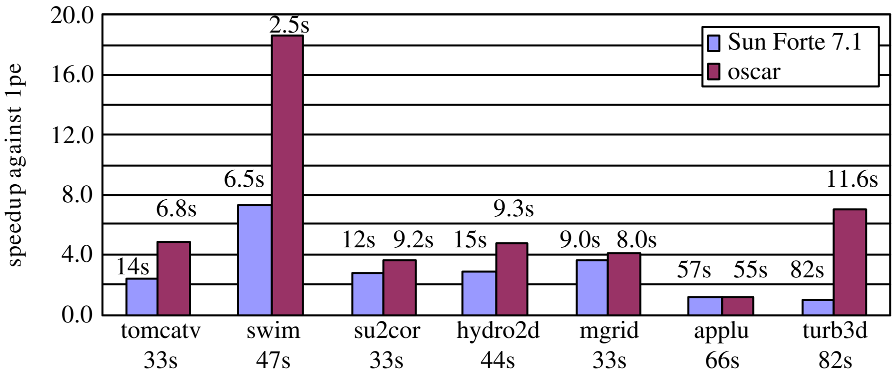

Fig. 13. Performance on Sun Fire V880 8 Processors Server

4 Conclusions

This paper has described the current performance of OSCAR multigrain parallelizing compiler that has been developed as a Japanese Government Millennium Project IT21 Advanced Parallelizing Compiler (APC). The OSCAR compiler exploits the coarse grain task parallelism, the loop parallelism and the near fine grain parallelism hierarchically with "one-time single level thread generation technique" and global cache optimization with padding data layout transformation for various SMP servers such as IBM pSeries 690 24 way Power4 SMP server, Sun Fire V880 8 UltraSPARC III server, Sun Ultra 80 4 UltraSPARC II desktop workstation, NEC TX7/i6010 8 Itanium 2 server, SGI Altix 3700 16 Itanium 2 server.

It currently gives us 5.7 times speedup compared with IBM XL Fortran compiler 8.1 on the IBM pSeries 690 in the average of 7 programs, such as SPEC CFP95 tomcatv, swim, su2cor, hydro2d, mgrid, applu and turb3d. Also it gives us 2.6 times speedup compared with Intel Fortran Itanium Compiler 7.1 on SGI Altix 3700, 1.7 times speedup compared with NEC Fortran Itanium Compiler 3.4 on NEC TX7/i6010, 2.5 times compared with Sun Forte 7.0 on Sun Ultra 80, and 2.1 times speedup compare with Sun Forte compiler 7.1 on Sun Fire V880.

Acknowledgments

A part of this research was supported by METI/NEDO millennium project IT21 "advanced Parallelizing Compiler", NEDO Advanced Heterogeneous Multiprocessor Project and STARC (Semiconductor Technology Academic Research Center). Also, the auhours thank to NEC soft, Ltd. and SGI Japan, Ltd. for the kind offer of the use of the NEC TX7/i6010 and SGI Altix 3700 System for this research.

References

1. Randy Allen and Ken Kennedy. *Optimizaing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, 2001.
2. Michel Wolfe. *High performance compilers for parallel computing.* Addison-Wesley, 1996.
3. R. Eigenmann, J. Hoeflinger, and D. Padua. On the automatic parallelization of the perfect benchmarks. *IEEE Trans. on parallel and distributed systems*, 9(1), Jan. 1998.
4. W. Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proc. of Super Computing '91*, 1991.
5. M. R. Haghighat and C. D. Polychronopoulos. *Symbolic analysis for parallelizing compilers.* Kluwer Academic Publishers, 1995.
6. P. Tu and D. Padua. Automatic array privatization. *Proc. 6th Annual Workshop on Languages and Compilers for Parallel Computing*, 1993.
7. L. Rauchwerger, N. M. Amato, and D. A. Padua. Run-time methods for parallelizing partially parallel loops. *Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona, Spain*, pages 137–146, Jul. 1995.
8. Jay Hoeflinger and Yunheung Paek. Unified interprocedural parallelism detection. *International Journal of Parallel Processing*, 2000.
9. M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S. Liao, , and M. S. Lam. Interprocedural parallelization analysis: A case study. *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, Aug. 1995.
10. J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. *Proc. of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing*, Jul. 1995.
11. A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitoning algorithm to maximize parallelism and minimize communication. *Proc. of the 13th ACM SIGARCH International Conference on Supercomputing*, Jun. 1999.
12. X. Martorell, E. Ayguade, N. Navarro, J. Corbalan, M. Gonzalez, and J. Labarta. Thread fork/join techniques for multi-level parallelism exploitatio in numa multiprocessors. *Proc. of the 1999 International Conference on Supercomputing*, June 1999.
13. H. Kasahara, M. Obata, and K. Ishizaka. Automatic coarse grain task parallel processing on smp using openmp. *Proc. of 13 th International Workshop on Languages and Compilers for Parallel Computing 2000*, Aug. 2000.
14. H. Kasahara, H. Honda, M. Iwata, and M. Hirota. A macro-dataflow compilation scheme for hierarchical multiprocessor systems. In *Proc. Int'l. Conf. on Parallel Processing*, Aug. 1990.
15. H. Kasahara, H. Honda, and S. Narita. Parallel processing of near fine grain tasks using static scheduling on oscar.
16. K. Kimura and H. Kasahara. Near fine grain parallel processing using static scheduling on single chip multiprocessors. *Proc. of International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, Nov. 1999.
17. H. Kasahara A. Yhoshida, K. Koshizuka. Data-localization using loop aligned decomposition for macro-dataflow processing. *Proc. of 9th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1996.

18. K. Ishizaka, M. Obata, and H. Kasahara. Coarse grain task parallel processing with cache optimization on shared memory multiprocessor. In *Proc. of 14th International Workshop on Languages and Compilers for Parallel Computing*, Aug. 2001.
19. K. Ishizaka, M. Obata, and H. Kasahara. Cache optimization for coarse grain task parallel processing using inter-array padding. In *Proc. of 16th International Workshop on Languages and Compilers for Parallel Computing*, Oct. 2003.