Automatic Coarse Grain Task Parallel Processing Using OSCAR Multigrain Parallelizing Compiler

Motoki Obata, Kazuhisa Ishizaka, Hironori Kasahara
Waseda University
NEDO Advanced Parallelizing Compiler Project
3-4-1 Ohkubo, Shinjuku-ku, Tokyo, 169-8555, Japan
{kasahara,obata,ishizaka}@oscar.elec.waseda.ac.jp

Abstract

This paper describes a simple and efficient implementation method of automatic coarse grain task parallel processing named "One-time single level thread generation" using OSCAR multigrain parallelizing compiler for a SMP machine. The coarse grain task parallel processing is important to improve the effective performance of wide range of multiprocessor systems from a single chip multiprocessor to a high performance computer beyond the limit of the loop parallelism. The proposed method realizes hierarchical coarse grain task parallel processing using static, centralized dynamic or distributed dynamic scheduling for each nested macro-task graph with use of ordinary thread generation method like OpenMP. The performance evaluation of the proposed method, which generates parallelized program based on "One-time single level thread generation" using OpenMP from a sequential Fortran program, shows that OSCAR compiler gives us 6.3 times speed up in average against sequential processing on RS6000 SP 604e High Node 8-processor SMP machine for five application programs from SPEC 95fp and Perfect Benchmarks though IBM XL Fortran automatic loop parallelizing compiler gives us 3.3 times speed up in average.

1 Introduction

Currently, multiprocessor system, especially shared memory multiprocessor system, is widely used. However, the gap between peak performance and effective performance is getting large with the increase of processors.

The loop parallelization techniques, such as Doall and Do-across, have been used in Fortran parallelizing compilers for multiprocessor systems[1, 2]. Currently, many types of Do-loop can be parallelized with various data dependency analysis techniques[3, 4] such as GCD, Benerjee's inexact and exact tests[1, 2], OMEGA test[5], symbolic analysis[6], semantic analysis and dynamic dependence test and program restructuring techniques such as array privatization[7], loop distribution, loop fusion, strip mining and loop interchange [8, 9].

For example, Polaris compiler[10, 11, 12] exploits loop parallelism by using inline expansion of subroutine, symbolic propagation, array privatization[7, 11] and run-time data dependence analysis[12]. SUIF compiler parallelizes loops by using inter-procedure analysis[13, 14, 15], unimodular transformation and data locality optimization[16, 17].

However, these compilers cannot parallelize loops that include complex loop carrying dependences and conditional branches to the outside of a loop. Considering these facts, the coarse grain task parallelism should be exploited to improve the effective performance of multiprocessor systems further in addition to the improvement of data dependence analysis, speculative execution and so on.

NANOS compiler[18, 19] based on Parafrase2 has been trying to exploit multi-level parallelism including the coarse grain parallelism by using extended OpenMP API. PROMIS compiler[20, 21] hierarchically combines Parafrase2 compiler[22] using HTG[23] and symbolic analysis techniques[6] and EVE compiler for fine grain parallel processing. Currently, PROMIS compiler has been developing in University of Illinois for practical use.

OSCAR compiler has realized a multi-grain parallel processing [24, 25, 26] that effectively combines the coarse grain task parallel processing [24, 25, 26, 27, 28, 29], the loop parallelization and near fine grain parallel processing[30]. In OSCAR compiler, coarse grain tasks are dynamically scheduled onto processors or processor clusters to cope with the run-time uncertainties caused by conditional branches by dynamic scheduling routine generated by the compiler.

Base on research on OSCAR compiler in Japan, Advanced Parallelizing Compiler(APC) project[31] was started to improve the effective performance, ease of use and cost performance of shared memory multiprocessor systems by using multiple grain of parallelism in addition to ordinary loop level parallelism.

This paper describes the implementation scheme of a coarse grain task parallel processing on a commercially available SMP machine and its performance. Ordinary sequential Fortran programs are parallelized using by OSCAR compiler automatically and a parallelized program with OpenMP API[32, 33] is generated. In other words, OSCAR Fortran Compiler is used as a pre-processor which transforms a Fortran program into a parallelized OpenMP Fortran realizing static scheduling and centralized and distributed dynamic scheduling for coarse grain tasks depending on parallelism of the source program and performance parameters of the target machines. Parallel threads are forked only once at the beginning of the program and joined only once at the end to minimize fork/join overhead by using "One-time single level thread generation" scheme. Though OpenMP API is chosen as the thread creation method in this paper because of the portability, this implementation scheme can be used for other thread generation method as well.

The performance of the proposed coarse grain task parallel processing in OSCAR multigrain compiler is evaluated on IBM RS6000 SP 604e High Node 8 processors SMP machine. In the evaluation, OSCAR multigrain compiler automatically generates coarse grain parallel processing codes using a subset of OpenMP directives supported by IBM XL Fortran version 5.1. The codes are compiled by XL Fortran and executed on 8 processors of RS6000 SP 604e High Node.

The rest of this paper is composed as follows. Section 2 introduces the coarse grain task parallel processing scheme. Section 3 shows the implementation method of the coarse grain task parallelization on a SMP. Section 4 evaluates the performance of this method on IBM RS6000 SP 604e High Node for several programs like Perfect Benchmarks and SPEC 95fp Benchmarks.

2 Coarse Grain Task Parallel Processing

Coarse grain task parallel processing uses parallelism among three kinds of macro-tasks, namely, Basic Block(BB), Repetition Block(RB or loop) and Subroutine Block(SB). Macro-tasks are generated by decomposition of a source program and assigned to processor clusters or processor elements and executed in parallel inter and/or intra processor clusters.

The coarse grain task parallel processing scheme in OSCAR multigrain automatic parallelizing compiler consists of the following steps.

1. Generation of macro-tasks from a source code of an ordinary sequential program
2. Generation of Macro-Flow Graph which represents data dependency and control flow among macro-tasks.
3. Generation of Macro-Task Graph representing parallelism among macro-tasks by Earliest Executable Condition analysis [24, 27, 28] unifying control and data dependencies.
4. Scheduling of macro-tasks in each nest level processors or processor clusters. If a macro-task graph in a nest level has only data dependency edges, the macro-tasks are assigned to processor clusters or processor elements by static scheduling at compile-time. If a macro-task graph has both data dependency and control dependency edges, macro-tasks are assigned to processor clusters or processor elements at run-time by dynamic scheduling routine generated and embedded into the parallelized user code by the compiler.

In the following, these steps are briefly explained.

2.1 Generation of Macro-tasks

In the coarse grain task parallel processing, a source program is decomposed into three kinds of macro-tasks, namely, Basic Block(BB), Repetition Block(RB) and Subroutine Block(SB) as mentioned above.

If there is a parallelizable loop, it is decomposed into smaller loops in the iteration direction and the decomposed partial loops are defined as different macro-tasks. The number of decomposed loops is decided considering the number of processor clusters and processor elements, cache size or memory size.

RBs composed of a sequential loops having large processing cost and SBs to which inline expansion can not apply effectively are decomposed into macro-tasks hierarchically and the hierarchical coarse grain task parallel processing is applied as shown in Figure 2 explained later.

2.2 Generation of Macro-flow Graph

Next, the data dependency and control flow among macro-tasks for each nest level are analyzed hierarchically. The control flow and data dependency

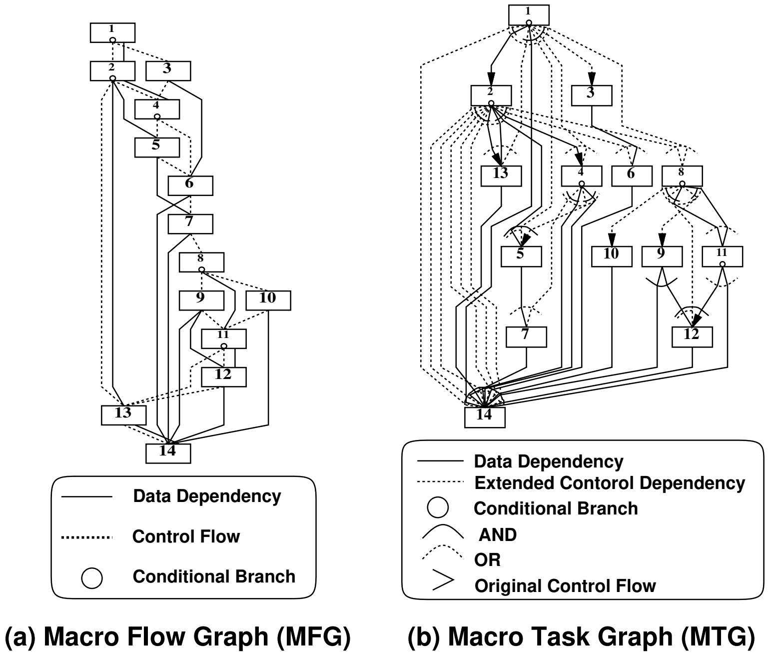

Figure 1: Macro flow graph and macro-task graph

among macro-tasks are represented by macro-flow graph as shown in Figure 1(a).

In the figure, nodes represent macro-tasks, solid edges represent data dependencies among macro-tasks and dotted edges represent control flow. A small circle inside a node represents a conditional branch inside the macro-task. Though arrows of edges are omitted in the macro-flow graph, it is assumed that the directions are downward.

2.3 Generation of Macro-task Graph

Though the generated macro-flow graph represents data dependencies and control flow, it does not represent parallelism among macro-tasks. To extract parallelism among macro-tasks from macro-flow graph, Earliest Executable Condition analysis considering data dependencies and control dependencies is used. Earliest Executable Condition represents the conditions on which macro-task may begin its execution earliest. It is obtained assuming the following conditions.

1. If Macro-Task(MT)i is data-dependent on MTj, MTi cannot begin execution before MTj finishes execution.

2. If the branch direction of MTj is determined, MTi that is control-dependent but not data-dependent on MTj can begin execution even though MTj has not completed its execution.

For example, the simplest form of Earliest Executable Condition of MT6 is the following:

(MT3 completes execution
OR
MT2 takes a branch that
guarantees execution of MT4).

Earliest Executable Condition of macro-task is represented in a macro-task graph as shown in Figure 1(b).

In the 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 represents ordinary control dependency and the condition on which a data dependence predecessor of MTi is not executed.

Solid and dotted arcs connecting solid and dotted edges have two different meanings. 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.

In MTG, though arrows of edges are omitted assuming downward, an edge having arrow represents original control flow edges, or branch direction in macro-flow graph.

2.4 Generation of Scheduling Routine

In the coarse grain task parallel processing, dynamic scheduling and static scheduling are used for assignment of macro-tasks to processor clusters or processor elements. In the dynamic scheduling, MTs are assigned to processor clusters or processor elements at run-time to cope with run-time uncertainties like conditional branches. The dynamic scheduling routine is generated and embedded into user program by OSCAR compiler to eliminate the overhead of OS call for thread scheduling.

Though generally dynamic scheduling overhead is large, overhead of the compiler generated dynamic scheduling routine is relatively small since it is optimized by the compiler and also used for the coarse grain tasks assignment.

Furthermore, two kinds dynamic scheduling scheme are adopted, namely, centralized dynamic scheduling in which the scheduling routine is executed by a processor element, and distributed scheduling in which the scheduling routine is distributed to all processors.

In static scheduling, assignment of macro-tasks to processor clusters or processor elements is determined at compile-time if macro-task graph has only data dependency edges and task processing times can be accurately estimated. Static scheduling is effective to minimize data transfer and synchronization overheard without run-time scheduling overhead.

3 Implementation of Coarse Grain Task Parallel Processing Using OpenMP

This section describes an implementation method of the coarse grain task parallel processing using OpenMP for SMP machines.

Though macro-tasks are assigned to processor clusters or processor elements in the coarse grain task parallel processing in OSCAR compiler, OpenMP supports the thread level parallel processing. Therefore, the coarse grain parallel processing is realized by corresponding a thread to a processor element, and a thread group to a processor cluster.

This scheme can be realized with other thread creation methods as well, though OpenMP is used in this paper as an example of thread generation methods with high portability.

3.1 Generation of Threads

In the proposed "One-time single level thread generation" coarse grain task parallel processing scheme using OpenMP, threads are generated by PARALLEL SECTIONS directive only once at the beginning of the execution of program.

Generally, upper level master threads fork nested children threads to realize nested or hierarchical parallel processing. However, this scheme realizes the hierarchical parallel processing by writing all hierarchical behavior, or by embedding hierarchical scheduling routines, in a section between PARALLEL SECTIONS and END PARALLEL SECTIONS. This scheme allows us to minimize thread fork and join overhead and to implement hierarchical coarse grain task parallel processing without any language extension.

3.2 Macro-task scheduling

This section describes code generation schemes using static and dynamic scheduling for hierarchical coarse grain task parallel processing.

OSCAR compiler can choose the centralized dynamic scheduling and/or the distributed dynamic scheduling scheme in addition to static scheduling to assign macro-tasks to threads or thread groups. These scheduling methods are used considering parallelism of the source program, a number of processors, data transfer and synchronization overhead of a target multiprocessor system with their any hierarchical combinations. In the centralized dynamic scheduling, scheduling code is assigned to a single thread. In the distributed dynamic scheduling, scheduling code is distributed to before and after each task assuming exclusive access to the scheduling tables. Those scheduling methods can be hierarchically combined freely depending on program parallelism, the number of processors available for the program layer, synchronization overhead and so on.

3.2.1 Centralized dynamic scheduling

In the centralized scheduling scheme, one thread in a parallel processing layer choosing centralized scheduling serves as a centralized scheduler, which assigns macro-tasks to threads or thread groups.

The behavior of the centralized scheduler written in OpenMP "SECTION" is shown in the following.

step1 Receive a completion or branch signal from each macro-task.

step2 Check Earliest Executable Condition(EEC), and enqueue ready macro-tasks which satisfy EEC to a ready task queue.

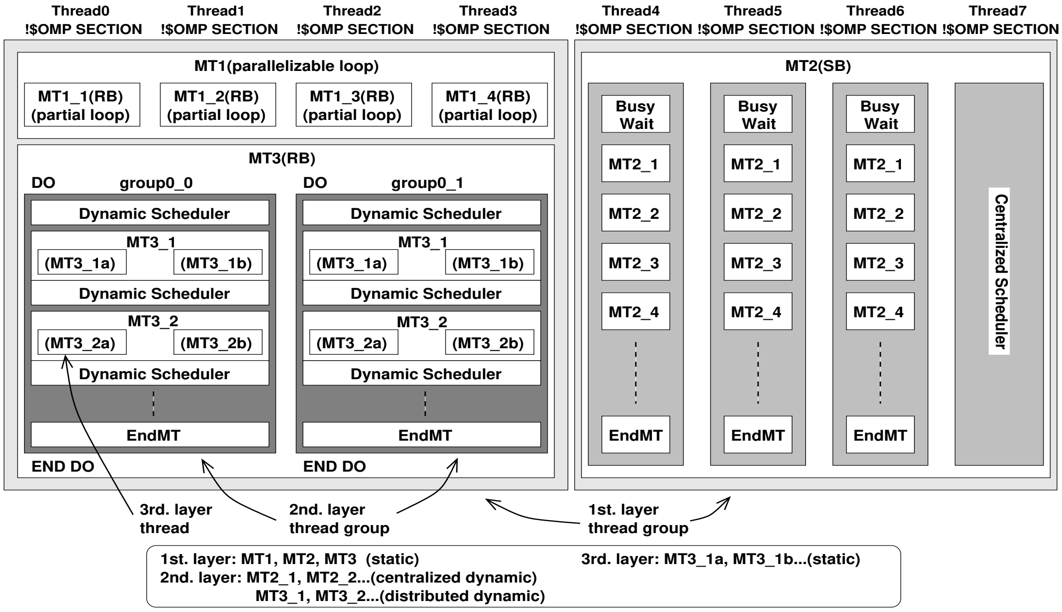

Figure 2: Code image (8 threads)

step3 Find a thread group or thread, to which a ready macro-task should be assigned according to the priority of Dynamic CP method.[24]

step4 Assign a macro-task to the thread group or a thread. If the assigned macro-task is "End MT"(EMT), the centralized scheduler finishes scheduling routine in the layer.

step5 Jump to step1.

During dynamic scheduling at run-time, when a macro-task finishes execution or determines branch direction, it sends the signals to the centralized scheduler. The centralized scheduler busy waits for these signals. If the centralized scheduler receives these signals, it quickly searches new executable, or ready, macro-tasks by checking Earliest Executable Condition with optimized scheduling code generated by the compiler.

If a ready macro-task is found, the centralized scheduler finds a thread group or a thread to which a macro-task should be assigned. The centralized scheduler assigns macro-tasks to the slave threads and goes back to the signal waiting routine. Slave threads to which no task is assigned execute the busy wait code until a task is assigned by the scheduler. After the completion of the assigned macro-task, the slave threads go back to the busy wait routine.

Figure 2 shows an image of generated OpenMP code for each thread. Sub macro-tasks generated inside of Macro-Task(MT)2 and MT3 in the 1st layer are represented as macro-tasks in the 2nd layer like MT2_1, MT2_2, MT3_1, MT3_2 and so on. In this example, a centralized scheduling scheme is applied to the 2nd layer inside MT2 in the 1st layer and a distributed scheduling scheme is applied to inside MT3 in the 1st layer. The 2nd layer macro-tasks inside MT2 are executed on Thread 4∼6 and Thread 7 as a centralized dynamic scheduler. In the dynamic scheduling mode, since every thread or thread group has a possibility to execute all macro-tasks, so the same code including all macro-tasks is copied into each OpenMP "SECTION" for each slave thread as shown in Thread 4∼6 of Figure 2.

Inside MT3_1 and MT3_2 in the 2nd layer, macro-tasks like MT3_1a, MT3_1b and so on in the 3rd layer are generated. Figure 2 also exemplifies a code image in a case where the 2nd layer macro-tasks inside MT3 in the 1st layer are dynamically scheduled to threads by the distributed scheduler. The details of distributed scheduling are described later.

Also, the compiler generates a special macro-task called "End MT"(EMT) in each layer. As shown in the 2nd layer inside MT2 in Figure 2, the EndMT is written at the end of all OpenMP "SECTION", and centralized scheduler assigns EMT to the all thread groups when all thread executing the same hierarchy finish. After assigning EMT to the all thread, centralized scheduler finishes scheduling routine in the layer. Each thread group jumps to outside of its hierarchy or nest level. If the hierarchy is a top layer or main routine, the program finishes the execution. If there exists an upper layer, threads continue to execute the upper layer macro-tasks.

3.2.2 Distributed Dynamic Scheduling.

In the distributed dynamic scheduling mode, each thread group schedules a macro-task to itself and executes the assigned macro-task.

In this scheduling scheme, all shared data for scheduling are shared and accessed exclusively as shown in the following.

step1 Search ready macro-tasks that satisfy Earliest Executable Condition by the completion or a branch of the macro-task and enqueue the ready macro-tasks to the ready task queue with exclusive access to shared data for dynamic scheduling.

step2 Choose a macro-task, which the thread should execute next, considering Dynamic CP[24] algorithm's priority.

step3 Execute the macro-task

step4 Update the Earliest Executable Condition table exclusively.

step5 Go back to step1

For example, this distributed dynamic scheduling scheme is applied to the 2nd layer inside MT3 in the 1st layer in Figure 2 and two thread groups that consist of two threads are realized in this layer by only executing the thread code generated by OSCAR compiler.

3.2.3 Static Scheduling Scheme

If a macro-task graph has only data dependencies, the static scheduling scheme is applied to the macro-task graph to reduce data transfer, synchronization and scheduling overheads.

In the static scheduling, the assignment of macro-tasks to thread groups or threads is determined at compile-time. Therefore, each OpenMP "SECTION" needs only the macro-tasks that should be executed in the order pre-determined by static scheduling algorithms CP/DT/MISF, DT/CP and ETF/CP. In other words, the compiler generates different program to each threads as shown in the 1st layer of Figure 2. When this static scheduling is used, it is assumed that each thread is bound to a processor.

If each thread group needs to synchronize and transfer data among other thread groups in the same hierarchy to satisfy the data dependency among macro-tasks as static scheduling result, OSCAR compiler prepares shared variables and program codes for the data transfer and generates busy wait code by using the shared variables for synchronization.

For example, static scheduling scheme is applied to the 1st layer shown in Figure 2. Each OpenMP "SECTION" for Thread 0∼3 have only codes of MT1 and MT3 and parts of Thread 4∼7 have only codes of MT2 in the 1st layer.

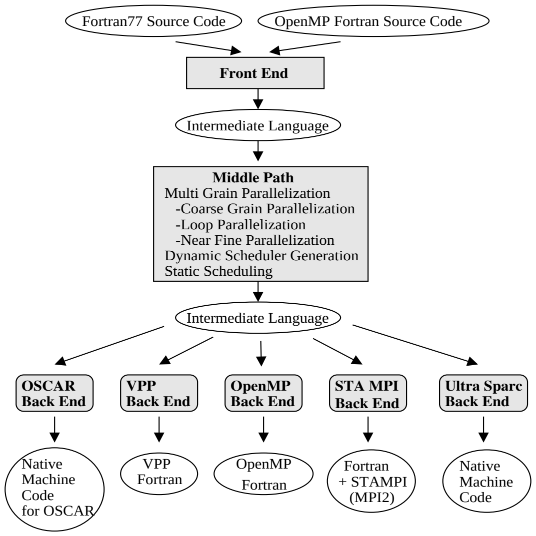

Figure 3: OSCAR Fortran Compiler

4 Performance Evaluation

This section describes the optimization for exploiting coarse grain task parallelization by OSCAR Fortran Compiler and its performance for several programs in Perfect benchmarks and SPEC 95fp benchmarks on IBM RS6000 SP 604e High Node 8 processor SMP.

4.1 OSCAR Fortran Compiler

OSCAR Fortran Compiler consists of Front End, Middle Path and Back Ends as shown in Figure 3. OSCAR Fortran Compiler has various Back Ends for different target multiprocessor systems like OSCAR distributed/shared memory multiprocessor system[34], UltraSparc, MPI-2 and OpenMP. OpenMP Back End used in this paper, which generates the parallelized Fortran source code with OpenMP directives. In other words, OSCAR Fortran Compiler is used as a pre-processor that transforms from an ordinary sequential Fortran program to OpenMP Fortran program for SMP machines.

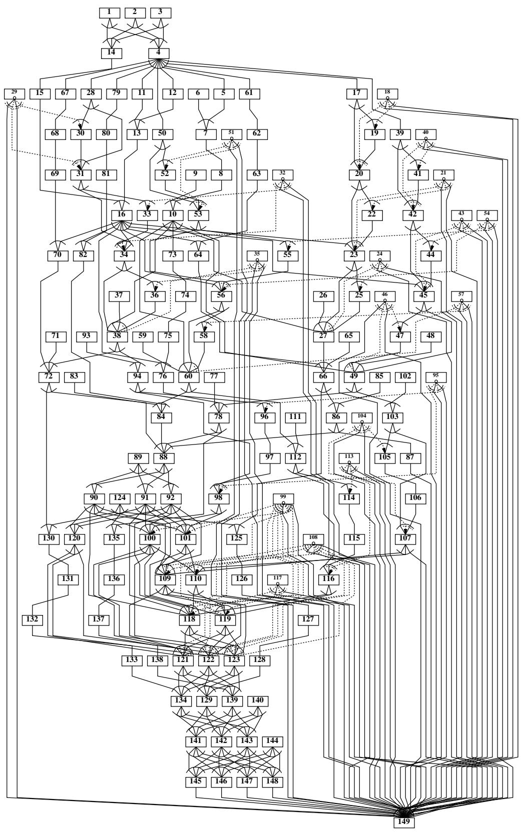

Figure 4: Optimized macro-task graph of subroutine INTEGR in ARC2D

4.2 Evaluation environment

In this evaluation, a coarse grain parallelized program automatically generated by OSCAR compiler is compiled by IBM XL Fortran compiler version 5.1[35] and executed on 1 through 8 processors of RS6000 SP 604e High Node. RS6000 SP 604e High Node is SMP having 8 PowerPC 604e processors and each processor has 32KB L1 instruction and data caches and 1MB L2 unified cache. The performance of OSCAR compiler with XL Fortran compiler is compared with native IBM XL automatic parallelizing Fortran compiler[36] and shown as the difference of speed-up ratio. Speed-up ratio is calculated by (sequential execution time) / (parallel processing time). As the sequential execution time, the shortest execution time by XL Fortran Compiler using maximum optimization options is used. In the compilation for parallel processing by a XL Fortran, maximum optimization option "-qsmp=auto -O3 -qmaxmem=-1 -qhot" is used.

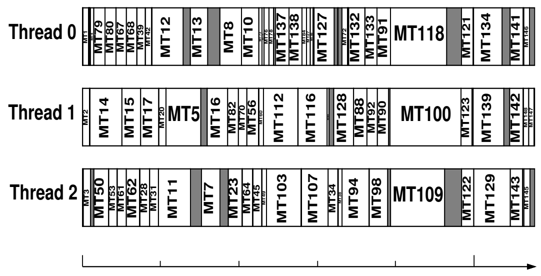

Figure 5: Execution trace data of INTEGR

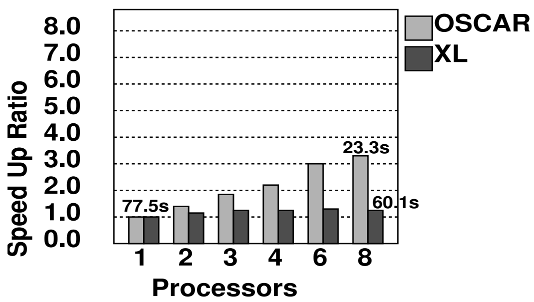

Figure 6: Speed-up ratio of ARC2D

4.3 Evaluation result

The programs used for performance evaluation are ARC2D in Perfect Benchmarks, SWIM, TOMCATV, HYDRO2D, MGRID in SPEC 95fp Benchmarks. ARC2D is an implicit finite difference code for analyzing fluid flow problems and solves Euler equations. SWIM solves the system of shallow water equations using finite difference approximations. TOMCATV is a vectorized mesh generation program. HYDRO2D is a vectorizable Fortran program with double precision floating-point arithmetic. MGRID is the Multi-grid solver in 3D potential field.

First, as an example of coarse grain task parallelism exploitation, compiler optimizations for ARC2D are described. ARC2D has about 4500 statements including 40 subroutines. More than 90% of the execution time is spent in subroutine INTEGR. Since subroutine INTEGR has several subroutines including loops having the small number of iterations, only use of the loop level parallelism cannot attain large performance improvement even if more than 4 or 5 processors are used. Therefore, OSCAR compiler applies loop unrolling to the loops inside the subroutines. In addition, the inline expansion is applied to the subroutine calls in the subroutine INTEGR for extracting more coarse grain parallelism. As the result, macro-task graph

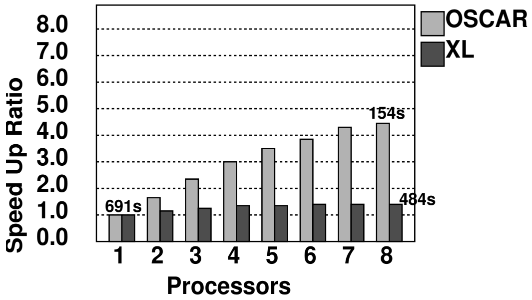

Figure 7: Speed-up ratio of TOMCATV

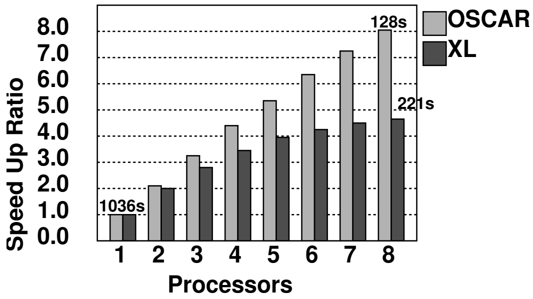

Figure 9: Speed-up ratio of HYDRO2D

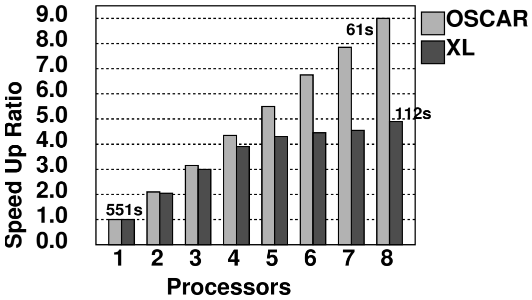

Figure 8: Speed-up ratio of SWIM

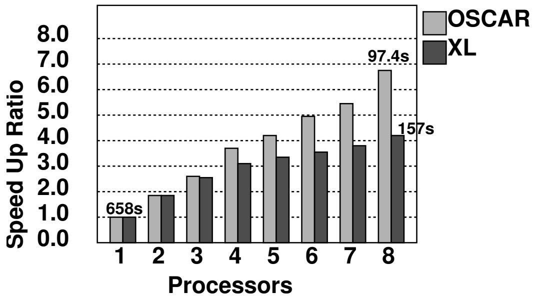

Figure 10: Speed-up ratio of MGRID

of subroutine INTEGR shown in Figure 4.3 is generated. The performance difference between OSCAR compiler and XL Fortran compiler in Figure 4.3 come from the coarse grain parallelism detected by OSCAR compiler. Figure 4.3 shows the execution trace image of subroutine INTEGR by distributed dynamic scheduling among 3 threads. Scheduling overhead is very small as shown in the edges between macro-tasks in Figure 4.3.

Figure 6 shows speed-up ratios for ARC2D by the proposed coarse grain task parallelization scheme in OSCAR compiler and the automatic loop parallelization by XL Fortran compiler. In this figure, the numbers with bar charts for 1 and 8PE show each execution time. The sequential processing time for ARC2D was 77.5s and parallel processing time by XL Fortran version 5.1 compiler using 8 processors was 60.1s. On the other hand, the execution time of coarse grain parallel processing using 8 processors by OSCAR Fortran compiler combined with XL Fortran compiler was 23.3s. In other words, OSCAR compiler gave us 3.3 times speed up against sequential processing time and 2.6 times speed up against native XL Fortran compiler for 8 processors. The performance difference between OSCAR compiler and XL Fortran compiler come from the exploitation of the coarse grain parallelism by OSCAR compiler.

Next, Figure 7 shows speed-up ratio for TOMCATV. The sequential execution time of TOMCATV was 691s. The parallel processing time using 8 processors by XL Fortran was 484s and 1.4 times speed-up against sequential execution time. On the other hand, the coarse grain parallel processing using 8 processors by OSCAR Fortran compiler was 154s and gave us 4.5 times speed-up against sequential execution time. OSCAR Fortran compiler also gave us 3.1 times speed up compared with XL Fortran compiler using 8 processors.

Figure 8 shows speed-up ratio for SWIM. The sequential execution time of SWIM was 551s. While the automatic loop parallel processing time using 8 processors by XL Fortran needed 112.7s and 4.9 times speed-up was attained, coarse grain task parallel processing by OSCAR Fortran compiler required only 61.1s and gave us 9.0 times speed-up by the effective use of distributed caches against the sequential execution time and 1.8 times speed-up compared with XL Fortran compiler.

Figure 9 shows speed-up in HYDRO2D. The sequential execution time of Hydro2d was 1036s. While XL Fortran gave us 4.7 times speed-up (221s) using 8 processors compared with the sequential execution time, OSCAR Fortran compiler gave us 8.1

times speed-up (128s) compared with sequential execution time.

Finally, Figure 10 shows speed-up ratio for MGRID. The sequential execution time of MGRID was 658s. For this application, XL Fortran compiler attains 4.2 times speed-up, or processing time of 157s, using 8 processors. Also, OSCAR compiler achieved 6.8 times speed up, or 97.4s. Namely, OSCAR Fortran compiler gave us 1.6 times speed-up compared with XL Fortran compiler for 8 processors.

5 Conclusions

This paper has described the realization scheme of the automatic coarse grain task parallel processing and its performance on an off the shelf SMP machine.

OSCAR compiler generates coarse grain parallelized code using OpenMP API which forks threads only once at the beginning of a program and joins only once at the end, namely "One-time single level thread generation", to minimize the overhead though hierarchical coarse grain task parallelism are automatically exploited. In this paper, though OpenMP API is selected as the thread generation method because of the portability, this realization scheme can be used for other thread generation method.

In the performance evaluation, OSCAR compiler with XL Fortran compiler gave us scalable speed up for five application programs in Perfect and SPEC 95fp benchmarks and significant speed-up compared with native XL Fortran compiler, such as 2.6 times for ARC2D, 1.8 times for SWIM, 3.1 times for TOMCATV, 1.7 times for HYDRO2D and 1.6 times for MGRID when the 8 processors are used. In other words, OSCAR Fortran compiler can boost the performance of XL Fortran compiler, which is one of the best commercially available loop parallelizing compilers for IBM RS6000 SP 604e High Node, easily using coarse grain parallelism with low overhead.

Currently, the authors are planning to evaluate the proposed coarse grain task parallel processing scheme on other SMP machines using OpenMP and data transfer overhead using data localization scheme to use distributed cache efficiently.

Acknowledgment

A part of this research has been supported by Japan Government Millennium Project METI/NEDO Advanced Parallelizing Compiler Project (http://www.apc.waseda.ac.jp) and Waseda University Grant for Special Research Projects No.2000A-154.

References

[1] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[2] U. Banerjee. Loop Parallelization. *Kluwer Academic Pub.*, 1994.

[3] U. Barnerjee. Dependence Analysis for Supercomputing. *Kluwer Pub.*, 1989.

[4] P. Petersen and D. Padua. Static and Dynamic Evaluation of Data Dependence Analysis. *Proc. Int'l conf. on supemputing*, Jun. 1993.

[5] W. Pugh. The OMEGA Test: A Fast and Practical Integer Programming Algorithm for Dependence Alysis. *Proc. Supercomputing'91*, 1991.

[6] M. R. Haghighat and C. D. Polychronopoulos. *Symbolic Analysis for Parallelizing Compliers.* Kluwer Academic Publishers, 1995.

[7] P. Tu and D. Padua. Automatic Array Privatization. *Proc. 6th Annual Workshop on Languages and Compilers for Parallel Computing*, 1993.

[8] M. Wolfe. Optimizing Supercompilers for Supercomputers. *MIT Press*, 1989.

[9] D. Padua and M. Wolfe. Advanced Compiler Optimizations for Supercomputers. *C.ACM*, 29(12):1184–1201, Dec. 1986.

[10] Polaris. http://polaris.cs.uiuc.edu/polaris/.

[11] 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.

[12] 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.

[13] 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 (LCPC95)*, Aug. 1995.

[14] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 1996.

[15] S. Amarasinghe, J. Anderson, M. Lam, and C. Tseng. The SUIF Compiler for Scalable Parallel Machines. *Proc. of the 7th SIAM conference on parallel processing for scientific computing*, 1995.

[16] M. S. Lam. Locallity Optimizations for Parallel Machines. *Third Joint International Conference on Vector and Parallel Processing*, Nov. 1994.

[17] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and Computation Transformations for Multiprocessors. *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing*, Jul. 1995.

[18] X. Martorell, E. Ayguade, N. Navarro, J. Corbalan, M. Gozalez, and J. Labarta. Thread Fork/Join Techniques for Multi-level Parllelism Exploitation in NUMA Multiprocessors. *ICS'99 Rhodes Greece*, 1999.

[19] E. Ayguade, X. Martorell, J. Labarta, M. Gonzalez, and N. Navarro. Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study. *ICPP'99*, Sep. 1999.

[20] PROMIS. http://www.csrd.uiuc.edu/promis/.

[21] C. J. Brownhill, A. Nicolau, S. Novack, and C. D. Polychronopoulos. Achieving Multi-level Parallelization. *Proc. of ISHPC'97*, Nov. 1997.

[22] Parafrase2. http://www.csrd.uiuc.edu/parafrase2/.

[23] M. Girkar and C. Polychronopoulos. Optimization of Data/Control Conditions in Task Graphs. *Proc. 4th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1991.

[24] H. K. et al. A Multi-grain Parallelizing Compilation Scheme on OSCAR. *Proc. 4th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1991.

[25] M. Okamoto, K. Aida, M. Miyazawa, H. Honda, and H. Kasahara. A Hierarchical Macro-dataflow Computation Scheme of OSCAR Multi-grain Compiler. *Trans. IPSJ*, 35(4):513–521, Apr. 1994.

[26] H. Kasahara, M. Okamoto, A. Yoshida, W. Ogata, K. Kimura, G. Matsui, H. Matsuzaki, and H.Honda. OSCAR Multi-grain Architecture and Its Evaluation. *Proc. International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, Oct. 1997.

[27] H. Kasahara, H. Honda, M. Iwata, and M. Hirota. A Macro-dataflow Compilation Scheme for Hierarchical Multiprocessor Systems. *Proc. Int'l. Conf. on Parallel Processing*, Aug. 1990.

[28] H. Honda, M. Iwata, and H. Kasahara. Coarse Grain Parallelism Detection Scheme of Fortran programs. *Trans. IEICE (in Japanese)*, J73-D-I(12), Dec. 1990.

[29] H. Kasahara. *Parallel Processing Technology*. Corona Publishing, Tokyo (in Japanese), Jun. 1991.

[30] H. Kasahara, H. Honda, and S. Narita. Parallel Processing of Near Fine Grain Tasks Using Static Scheduling on OSCAR. *Proc. IEEE ACM Supercomputing'90*, Nov. 1990.

[31] Advanced Parallelizing Compiler project http://www.apc.waseda.ac.jp/.

[32] OpenMP: Simple, Portable, Scalable SMP Programming http://www.openmp.org/.

[33] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared Memory Programming. *IEEE Computational Science & Engineering*, 1998.

[34] H. Kasahara, S. Narita, and S. Hashimoto. OSCAR's Architecture. *Trans. IEICE (in Japanese)*, J71-D-I(8), Aug. 1988.

[35] IBM. *XL Fortran for AIX Language Reference*.

[36] D. H. Kulkarni, S. Tandri, L. Martin, N. Copty, R. Silvera, X.-M. Tian, X. Xue, and J. Wang. XL Fortran Compiler for IBM SMP Systems. *AIXpert Magazine*, Dec. 1997.