Coarse-grain Task Parallel Processing Using the OpenMP Backend of the OSCAR Multigrain Parallelizing Compiler

Kazuhisa Ishizaka, Motoki Obata, Hironori Kasahara
{ishizaka,obata,kasahara}@oscar.elec.waseda.ac.jp

Waseda University
3-4-1 Ohkubo, Shinjuku-ku, Tokyo, 169-8555, Japan

Abstract. This paper describes automatic coarse grain parallel processing on a shared memory multiprocessor system using a newly developed OpenMP backend of OSCAR multigrain parallelizing compiler for from single chip multiprocessor to a high performance multiprocessor and a heterogeneous supercomputer cluster. OSCAR multigrain parallelizing compiler exploits coarse grain task parallelism and near fine grain parallelism in addition to traditional loop parallelism. The OpenMP backend generates parallelized Fortran code with OpenMP directives based on analyzed multigrain parallelism by middle path of OSCAR compiler from an ordinary Fortran source program. The performance of multigrain parallel processing function by OpenMP backend is evaluated on an off the shelf eight processor SMP machine, IBM RS6000. The evaluation shows that the multigrain parallel processing gives us more than 2 times speed up compared with a commercial loop parallelizing compiler, IBM XL Fortran compiler, on the SMP machine.

1 Introduction

Automatic parallelizing compilers have been getting more important with the increase of parallel processing in a high performance multiprocessor system and use of multiprocessor architecture inside a single chip and for an upcoming home server for improving effective performance, cost-performance and ease of use. Current parallelizing compilers exploit loop parallelism, such as Do-all and Do-across[33, 3]. In these compilers, Do-loops are parallelized using various data dependency analysis techniques [4, 25] such as GCD, Banerjee's inexact and exact tests [33, 3], OMEGA test[28], symbolic analysis[9], semantic analysis and dynamic dependence test and program restructuring techniques such as array privatization[31], loop distribution, loop fusion, strip mining and loop interchange[32, 23].

For example, Polaris compiler[26, 6, 29] exploits loop parallelism by using inline expansion of subroutine, symbolic propagation, array privatization [31, 6] and run-time data dependence analysis[29]. PROMIS compiler[27, 5] combines Parafrace2 compiler[24] using HTG[8] and symbolic analysis techniques[9], and

EVE compiler for fine grain parallel processing. SUIF compiler parallelizes loop by using inter-procedure analysis [10, 11, 1], unimodular transformation and data locality optimization[20, 2].

Effective optimization of data localization is more and more important because of the increasing a speed gap between memories and processors. Also, many researches for data locality optimization using program restructuring techniques such as blocking, tiling, padding and data localization, are proceeding for high performance computing and single chip multiprocessor systems [20, 12, 30, 35].

OSCAR compiler has realized a multigrain parallel processing [7, 22, 19] that effectively combines the coarse grain task parallel processing [7, 22, 19, 16, 13, 15], which can be applied from a single chip multiprocessor to HPC multiprocessor systems, the loop parallelization and near fine grain parallel processing[17]. In the conventional OSCAR compiler with the backend for OSCAR architecture, coarse grain tasks are dynamically scheduled onto processors or processor clusters to cope with the runtime uncertainties by the compiler. As the task scheduler, the dynamic scheduler in OSCAR Fortran compiler, and distributed dynamic scheduler[21] have been proposed.

This paper describes the implementation scheme of a thread level coarse grain parallel processing on a commercially available SMP machine and its performance. Ordinary sequential Fortran programs are parallelized using by OSCAR compiler with newly developed OpenMP backend automatically and a parallelized program with OpenMP directive is generated. In other words, OSCAR Fortran compiler is used as a preprocessor which transforms a Fortran program into a parallelized OpenMP Fortran. Parallel threads are forked only once at the beginning of the program and joined only once at the end in this scheme to minimize fork/join overhead. Also, this OSCAR OpenMP backend realizes hierarchical coarse grain parallel processing only using ordinary OpenMP directives though NANOS Compiler uses customly made n-thread library[34].

The rest of this paper is composed as follows. Section 2 introduces the execution model of the thread level coarse grain task parallel processing. Section 3 shows the coarse grain parallelization in OSCAR compiler. Section 4 shows the implementation method of the multigrain parallel processing in OpenMP backend. Section 5 evaluates the performance of this method on IBM RS6000 SP 604e High Node for several programs like Perfect Benchmarks and SPEC 95fp Benchmarks.

2 Execution Model of Coarse Grain Task Parallel Processing in OSCAR OpenMP Backend

This section describes the coarse grain task parallel processing using OpenMP directives. Coarse grain task parallel processing uses parallelism among three kinds of macro-tasks(MTs), namely, Basic Block(BB), and Repetition Block(RB), Subroutine Block(SB) described in Section 3. Macro-tasks are generated by decomposition of a source program and assigned to threads or thread groups and executed in parallel.

In the coarse grain task parallel processing using OSCAR OpenMP backend, threads are generated only once at the beginning of the program, and joined only once at the end. In other words, OSCAR OpenMP backend realizes hierarchical coarse grain parallel processing without hierarchical child thread generation. For example, in Fig.1, four threads are generated at the beginning of the program, and all generated threads are grouped to one thread group(group0). Thread group0 executes MT1, MT2 and MT3.

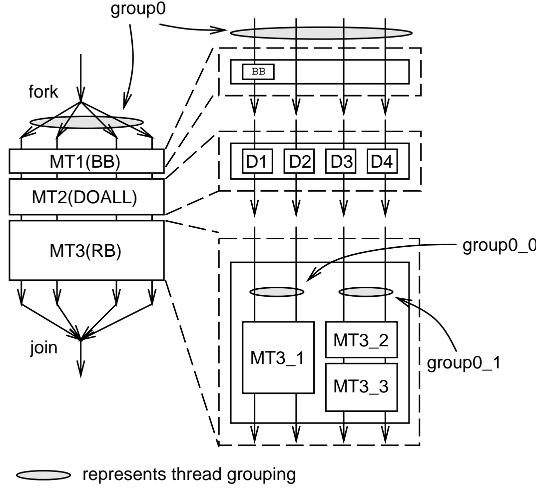

Fig. 1. execution image

When thread group executes a MT, threads in the group use parallelism inside a MT. For example, if MT is a parallelizable loop, threads in group use parallelism among loop iteration. In Fig.1, a parallelizable loop MT2 is distributed to four threads in the group. Also, nested parallelism among sub-MTs, which are generated by decomposition of body of a MT, is used. Sub MTs are assigned to nested(lower level) thread groups, that are hierarchically defined inside a upper level thread group. For example, MT3 in the Fig.1 is decomposed into sub-MTs(MT3_1,MT3_2 and MT3_3), and sub-MTs are executed by two nested thread groups, namely group0_0 and group0_1, each of which have two threads respectively. These groups are defined inside thread group0 which execute MT3.

3 Coarse Grain Parallelization in OSCAR Compiler

This section describes the analysis of OSCAR compiler for coarse grain task parallel processing. First, OSCAR compiler defines coarse grain macro-tasks from source program, and analyzes parallelism among macro-tasks. Next, the

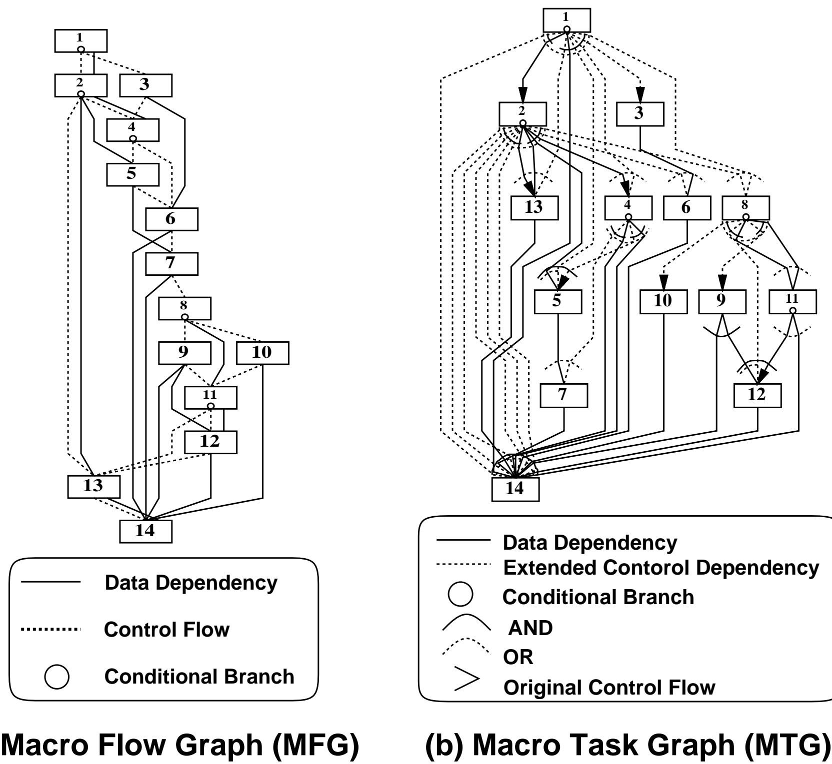

Fig. 2. Macro Flow Graph and Macro Task Graph

generated MTs are scheduled to thread groups statically at compile time or dynamically by embedded scheduling code generated by compiler.

3.1 Definition of Coarse Grain Task

In the coarse grain task parallel processing, a source program is decomposed into three kinds of MTs, namely, BB, RB and SB as mentioned above. Generated MTs are assigned to thread groups, and executed in parallel by threads in the thread group.

If a generated RB is a parallelizable loop, parallel iterations are distributed onto threads inside thread group considering cache size.

If a RB is a sequential loop having large processing cost or SB, it is decomposed into sub-macro-tasks and hierarchically processed by coarse grain task parallel processing scheme like MT3 in Fig.1.

3.2 Generation of Macro-Flow Graph

After generation of macro-tasks, the data dependency and control flow among MTs for each layer are analyzed hierarchically, and represented by Macro-Flow Graph(MFG) as shown in Fig.2(a).

In the Fig.2, nodes represent MTs, solid edges represent data dependencies among MTs and dotted edges represent control flow. A small circle inside a node represents a conditional branch inside the MT. Though arrows of edges are omitted in the MFG, it is assumed that the directions are downward.

3.3 Generation of Macro-Task Graph

To extract parallelism among MTs from MFG, Earliest Executable Condition analysis considering data dependencies and control dependencies is applied. Earliest Executable Condition represents the conditions on which MT may begin its execution earliest. It is obtained assuming the following conditions.

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

Then, the original form of Earliest Execution Condition is represented as follows;

(MTj, on which MTi is control dependent, branches to MTi) AND
(MTk($0 \leq k \leq |N|$), on which MTi is data dependent, completes execution OR
it is determined that MTk is not be executed), where N is the number of
predecessors of MTi

For example, the original form of Earliest Execution Condition of MT6 on Fig.2(b) is

(MT1 branches to MT3 OR MT2 branches to MT4) AND
(MT3 completes execution OR MT1 branches to MT4).

However, the completion of MT3 means that MT1 already branched to MT3. Also, "MT2 branches to MT4" means that MT1 already branched to MT2. Therefore, this condition is redundant and its simplest form is

(MT3 completes execution OR MT2 branches to MT4).

Earliest Execution Condition of MT is represented in Macro-Task Graph(MTG) as shown in Fig.2(b).

In MTG, nodes represent MTs. A small circle inside nodes represents conditional branches. Solid edges represent data dependencies. Dotted edges represent extended control dependencies. Extended control dependency means ordinary

normal 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 MFG.

3.4 Scheduling of MTs to Thread Groups

In the coarse grain task parallel processing, the static scheduling and the dynamic scheduling are used for assignment of MTs to thread groups.

In the dynamic scheduling, MTs are assigned to thread groups at runtime to cope with runtime uncertainties like conditional branches. The dynamic scheduling routine is generated and embedded into user program by compiler to eliminate the overhead of OS call for thread scheduling. Though generally dynamic scheduling overhead is large, in OSCAR compiler the dynamic scheduling overhead is relatively small since it is used for the coarse grain tasks with relatively large processing time.

In static scheduling, assignment of MTs to thread groups is determined at compile-time if MTG has only data dependency edges. Static scheduling is useful since it allows us to minimize data transfer and synchronization overheard without run-time scheduling overhead.

In the proposed coarse grain task parallel processing, both scheduling schemes are selectable for each hierarchy.

4 Code Generation in OpenMP Backend

This section describes a code generation scheme for the coarse grain task parallel processing using threads in OpenMP backend of OSCAR multigrain automatic parallelization compiler.

The code generation scheme is different for each scheduling scheme. Therefore, after a thread generation method is explained, the code generation scheme for each scheduling scheme is described.

4.1 Generation of Threads

In the proposed coarse grain task parallel processing using OpenMP, the same number of threads as the number of processors are generated by PARALLEL SECTIONS directive only once at the beginning of the execution of program.

Generally, to realize nested or hierarchical parallel processing, nested threads are forked by an upper level thread. However, in the proposed scheme, it is assumed that the number of generated thread, thread grouping and the scheduling scheme applied to each hierarchy are determined at compile-time. In other words,

the proposed scheme realizes this hierarchical parallel processing with single level thread generation by writing all MT code or embedding hierarchical scheduling routines in each OpenMP 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 parallel processing without special extension of OpenMP.

4.2 Static Scheduling

If a Macro Task Graph in a target layer has only data dependencies, the static scheduling is applied to reduce data transfer, synchronization and scheduling overheads.

In the static scheduling, the assignment of MTs to thread groups is determined at compile-time. Therefore, each OpenMP SECTION needs only the MTs that should be executed in the predetermined order.

At runtime, each thread group should synchronize and transfer shared data to other thread groups in the same hierarchy to satisfy the data dependency among MTs. Therefore, the compiler generates synchronization codes using shared memory.

A code image for eight threads generated by OpenMP backend of OSCAR compiler is shown in Fig.3. In this example, static scheduling is applied to the first layer. In Fig.3, eight threads are generated by OpenMP PARALLEL SECTIONS directives. The eight threads are grouped into two thread groups, each of which has four threads. MT1 and 3 are statically assigned to thread group0 and MT2 is assigned to thread group1.

When static scheduling is applied, compiler generates different codes for the thread groups which include only task codes assigned to the thread group.

The assigned MTs to thread groups are processed in parallel by threads inside the thread group by using static scheduling or dynamic scheduling hierarchically.

4.3 Dynamic Scheduling

Dynamic scheduling is applied for a Macro Task Graph with runtime uncertainty caused by a conditional branch. In the dynamic scheduling, since each thread group has possibility to execute any MTs, the all MT codes are copied to every OpenMP SECTION. Each thread group executes MTs selectively according to the scheduling result.

For the dynamic scheduling, OpenMP backend can generate centralized scheduler codes or distributed scheduler codes to be embedded into user code for any parallel processing layer, or nested level. In Fig3, MT2 assigned onto thread group1 is processed by four threads in parallel using the centralized scheduler. In the centralized scheduler method, a master thread, assigned to a thread, assigns macro-tasks to the other three slave threads.

The master thread repeats the following steps.

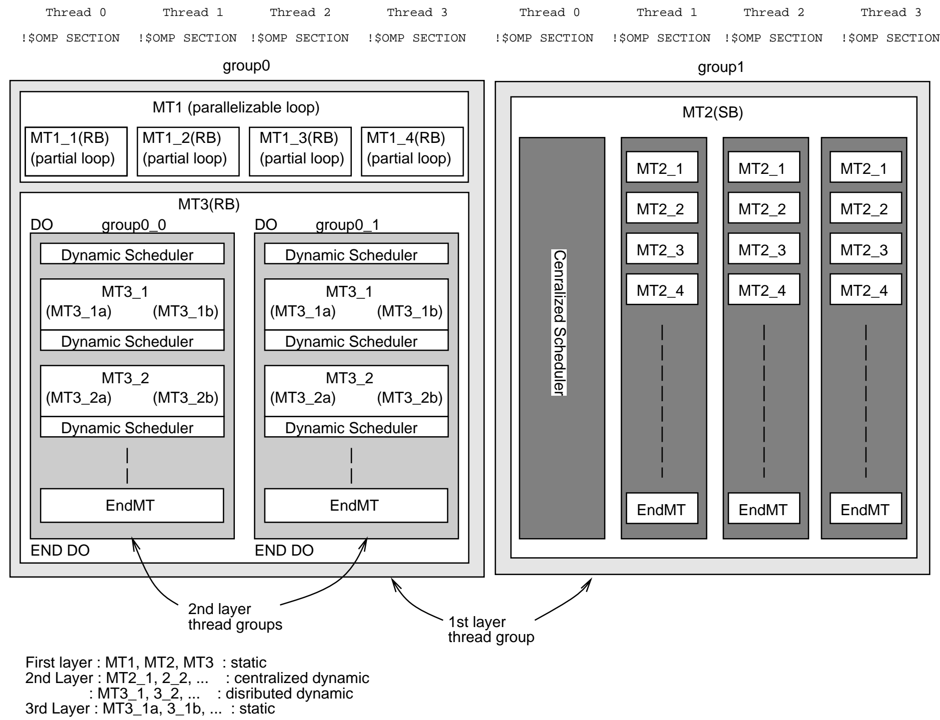

Fig. 3. Code image (four threads)

Behavior of Master thread

step1 Search executable, or ready, MTs of which Earliest Executable Condition(EEC) are satisfied by the completion or a branch of the preceding MT and enqueue the ready MTs to the ready queue.
step2 Choose a MT with highest priority and assigned it to a idle slave thread.
step3 Go back to step1

The behavior of slave threads is summarized in following.

Behavior of Slave thread

step1 Wait for the macro-task assignment by master thread.
step2 Execute assigned macro-task.
step3 Send signals to report to the master thread a branch direction and/or completion of the task execution.
step4 Go back to step1.

Also, the compiler generates a special MT called EndMT(EMT) in all OpenMP SECTIONS in each hierarchy. The assignment of EndMT shows the end of its

hierarchy. In other words, if a EndMT is scheduled to thread groups, the groups finish execution of a hierarchy. As shown in the second layer in Fig.3, the EndMT is written at the end of layer.

In Fig.3, it is assumed that MT2 is executed by master thread(thread 4) and three slave threads.

Next, MT3 shows an example of distributed dynamic scheduling. In this case, MT3 is decomposed into sub-macro-tasks and assigned thread group0_0 and 0_1 defined inside thread group0. In this example, the thread group0_0 and 0_1 has two threads. Each thread group works as scheduler, which behave same as master thread described before, though distributed dynamic schedulers need mutual exclusion to access the shared scheduling data like EEC and ready queue.

The distributed dynamic scheduling routines are embedded into before each macro-task code as shown in Fig.3. Furthermore, Fig.3 shows MT3_1, 3_2 and so on are processed by two threads inside thread group0_0, or 0_1.

5 Performance Evaluation

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

5.1 OSCAR Fortran Compiler

Fig.4 shows the overview of OSCAR Fortran Compiler. It consists of Front End(FE), Middle Path(MP) and Back Ends(BE). OSCAR Fortran Compiler has various Back Ends for different target multiprocessor systems like OSCAR distributed/shared memory multiprocessor system[18], Fujitsu's VPP supercomputer, UltraSparc, PowerPC, MPI-2 and OpenMP. The newly developed OpenMP Backend used in this paper, generates the parallelized Fortran source code with OpenMP directives. In other words, OSCAR Fortran Compiler is used as a preprocessor that transforms from an ordinary sequential Fortran program to OpenMP Fortran program for SMP machines.

5.2 Evaluated Programs

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 arithmetics. MGRID is the Multi-grid solver in 3D potential field.

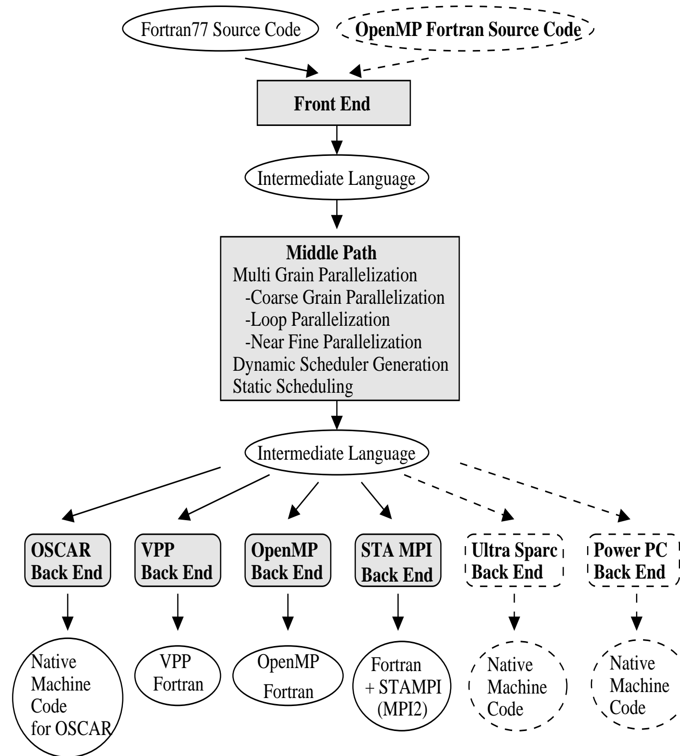

Fig. 4. Overview of OSCAR Fortran Compiler

5.3 Architecture of IBM RS6000 SP

RS6000 SP 604e High Node used for the evaluation is a SMP server having eight PowerPC 604e (200 MHz). Each processor has 32 KB L1 instruction and data caches and 1 MB L2 unified cache. The shared main memory is 1 GB.

5.4 Performance on RS6000 SP 604e High Node

In this evaluation, a coarse grain parallelized program automatically generated by OSCAR compiler is compiled by IBM XL Fortran compiler version 5.1[14] and executed on 1 through 8 processors of RS6000 SP 604e High Node. The performance of OSCAR compiler is compared with IBM XL automatic parallelizing Fortran compiler. In the compilation by a XL Fortran, maximum optimization option "-qsmp=auto -O3 -qmaxmem=-1 -qhot" is used.

Fig.5(a) shows speed-up ratio for ARC2D by the proposed coarse grain task parallelization scheme by OSCAR compiler and the automatic loop parallelization by XL Fortran compiler. The sequential processing time for ARC2D was 77.5s and parallel processing time by XL Fortran version 5.1 compiler using

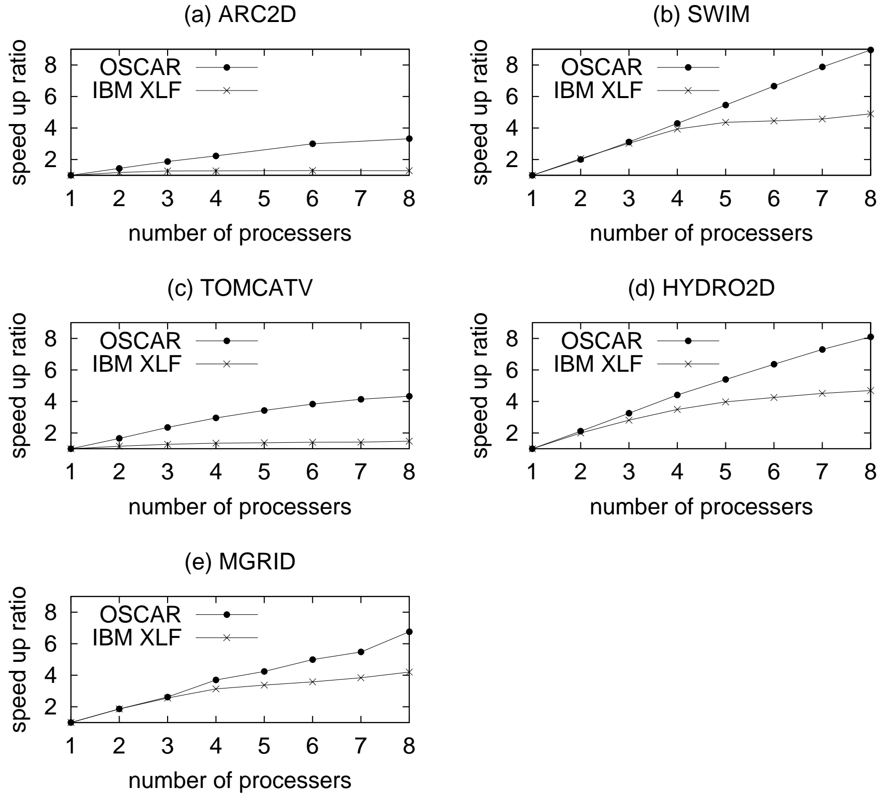

Fig. 5. Speed-up of several benchmarks on RS6000

8PEs was 60.1s. On the other hand, the execution time of coarse grain parallel processing using 8 PEs by OSCAR Fortran compiler 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 XL Fortran compiler for 8 processors.

Next, Fig.5(b) shows speed-up ratio for SWIM. The sequential execution time of SWIM was 551s. While the automatic loop parallel processing time using 8 PEs by XL Fortran needed 112.7s ,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.

Fig.5(c) shows speed-up ratio for TOMCATV. The sequential execution time of TOMCATV was 691s. The parallel processing time using 8 PEs by XL Fortran was 484s and 1.4 times speed-up. On the other hand, the coarse grain parallel processing using 8 PEs 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 PEs.

Fig.5(d) 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 PEs compared with the sequential execution time, OSCAR Fortran compiler gave us 8.1 times speed-up (128s).

Finally, Fig.5(e) 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 PEs. Also, OSCAR compiler achieved 6.8 times speed up, or 97.4s.

OSCAR Fortran Compiler gives us scalable speed-up and more than 2 times speed up for the evaluated benchmark programs compared with XL Fortran compiler

6 Conclusions

This paper has described performance of coarse grain task parallel processing using OpenMP backend of OSCAR multigrain parallelizing compiler. The OSCAR compiler generates a parallelized Fortran program using the OpenMP backend from a sequential Fortarn program. Though OSCAR compiler can exploit hierarchical multigrain parallelism, such as coarse grain task level, loop iteration level and statement level near fine grain task level, two kinds of parallelism, namely, the coarse grain task level and loop iteration level parallelism are examined in this paper considering machine performance parameters for the used eight processors SMP machine IBM RS6000 604e High Node.

The evaluation shows that OSCAR compiler gives us more then 2 times speedup compared with IBM XL Fortran compiler version 5.1 for several benchmark programs, such as Perfect Benchmarks ARC2D, spec95fp TOMCATV, SWIM, HYDRO2D, and MGRID.

The authors are planning to evaluate the performance of coarse grain parallel processing on various shared memory multiprocessor systems including SGI 2100, Sun Enterprise 3000 and so on using the developed OpenMP backend.

References

1. 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.
2. 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.
3. U. Banerjee. Loop parallelization. *Kluwer Academic Pub.*, 1994.
4. U. Barnerjee. Dependence analysis for supercomputing. *Kluwer Pub.*, 1989.
5. Carrie J. Brownhill, Alexandru Nicolau, Steve Novack, and Constantine D. Polychronopoulos. Achieving multi-level parallelization. *Proc. of ISHPC'97*, Nov. 1997.

6. Rudolf Eigenmann, Jay Hoeflinger, and David Padua. On the automatic parallelization of the perfect benchmarks. *IEEE Trans. on parallel and distributed systems*, 9(1), Jan. 1998.
7. H. Kasahara et al. A multi-grain parallelizing compilation scheme on oscar. *Proc. 4th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1991.
8. M. Girkar and C.D. Polychronopoulos. Optimization of data/control conditions in task graphs. *Proc. 4th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1991.
9. Mohammad R. Haghighat and Constantine D. Polychronopoulos. *Symbolic Analysis for Parallelizing Compliers*. Kluwer Academic Publishers, 1995.
10. 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.
11. Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, 1996.
12. Hwansoo Han, Gabriel Rivera, and Chau-Wen Tseng. Software support for improving locality in scientific codes. *8th Workshop on Compilers for Parallel Computers (CPC'2000)*, Jan. 2000.
13. 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.
14. IBM. *XL Fortran for AIX Language Reference.*
15. H. Kasahara. *Parallel Processing Technology.* Corona Publishing, Tokyo (in Japanese), Jun. 1991.
16. 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.
17. 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.
18. H. Kasahara, S. Narita, and S. Hashimoto. Oscar's architecture. *Trans. IEICE (in Japanese)*, J71-D-I(8), Aug. 1988.
19. 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.
20. Monica S. Lam. Locallity optimizations for parallel machines. *Third Joint International Conference on Vector and Parallel Processing*, Nov. 1994.
21. Jose E. Moreira and Constantine D. Polychronopoulos. Autoscheduling in a shared memory multiprocessor. *CSRD Report No.1337*, 1994.
22. 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.
23. D.A. Padua and M.J. Wolfe. Advanced compiler optimizations for supercomputers. *C.ACM*, 29(12):1184–1201, Dec. 1986.

24. Parafrase2. http://www.csrd.uiuc.edu/parafrase2/.
25. P.M. Petersen and D.A. Padua. Static and dynamic evaluation of data dependence analysis. *Proc. Int'l conf. on supemputing*, Jun. 1993.
26. Polaris. http://polaris.cs.uiuc.edu/polaris/.
27. PROMIS. http://www.csrd.uiuc.edu/promis/.
28. W. Pugh. The omega test: A fast and practical integer programming algorithm for dependence alysis. *Proc. Supercomputing'91*, 1991.
29. Lawrence Rauchwerger, Nancy M. Amato, and David 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.
30. Gabriel Rivera and Chau-Wen Tseng. Locality optimizations for multi-level caches. *Super Computing '99*, Nov. 1999.
31. P. Tu and D. Padua. Automatic array privatization. *Proc. 6th Annual Workshop on Languages and Compilers for Parallel Computing*, 1993.
32. M. Wolfe. Optimizing supercompilers for supercomputers. *MIT Press*, 1989.
33. M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
34. Nacho Navaro Xavier Martorell, Jesus Labarta and Eduard Ayguade. A library implementation of the nano-threads programing model. *Proc. of the Second International Euro-Par Conference, vol. 2*, 1996.
35. A. Yoshida, K. Koshizuka, M. Okamoto, and H. Kasahara. A data-localization scheme among loops for each layer in hierarchical coarse grain parallel processing. *Trans. of IPSJ*, 40(5), May. 1999.