Coarse Grain Task Parallel Processing with Cache Optimization on Shared Memory Multiprocessor

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

Dept.EECE, Waseda University
3-4-1 Ohkubo, Shinjuku-ku, Tokyo, 169-8555, Japan
Advanced Parallelizing Compiler Project
http://www.apc.waseda.ac.jp/

Abstract. In multiprocessor systems, the gap between peak and effective performance has getting larger. To cope with this performance gap, it is important to use multigrain parallelism in addition to ordinary loop level parallelism. Also, effective use of memory hierarchy is important for the performance improvement of multiprocessor systems because the speed gap between processors and memories is getting larger.
This paper describes coarse grain task parallel processing that uses parallelism among macro-tasks like loops and subroutines considering cache optimization using data localization scheme. The proposed scheme is implemented on OSCAR automatic multigrain parallelizing compiler. OSCAR compiler generates OpenMP FORTRAN program realizing the proposed scheme from an ordinary FORTRAN77 program. Its performance is evaluated on IBM RS6000 SP 604e High Node 8 processors SMP machine. In the evaluation, OSCAR compiler gives us up to 1.3 times speedup on 1PE, 4.7 times speedup on 4PE and 8.8 times speedup on 8PE compared with a sequential processing time.

1 Introduction

Shared memory multiprocessor architecture is widely used from a single chip multiprocessor to a high performance computer. The difference between peak performance and effective performance has been getting larger with the increase of the number of processors. Moreover, the speed gap between processors and memories is getting significant. Therefore, optimal use of hierarchical memories and task parallelism are very important for the improvement of effective performance of multiprocessor systems. However, the optimization requires high expertise for the parallel processing and the data distribution to the hierarchical memories, scheduling and so on. Considering the above facts, to improve effective performance and ease of use, an automatic parallelizing compiler realizing coarse grain parallel processing in addition to loop parallel processing with memory hierarchy optimization is required.

As automatic parallelizing compilers, loop parallelizing compilers are very popular for SMPs currently available on the market.

Also, advanced research compilers, such as Polaris[1] exploiting loop parallelism by using inline expansion of subroutine, symbolic propagation, array privatization, range test and run-time data dependence analysis [2, 3] and SUIF[4] which parallelizes loop by using inter-procedure analysis unimodular transformation and data locality optimization have been developed. [5–7].

The data locality optimization is essential to cope with increasing speed gap between processors and memories. Many researches for data locality optimization using program restructuring techniques such as blocking, tiling, padding and data localization have been performed for high performance computers and multiprocessor systems [8–10].

In spite of those efforts, the gap between peak and effective performance has been getting larger with increase of the number of processors. Therefore, the exploitation of multigrain parallelism in addition to the loop parallelism is required. Multigrain parallel processing, which has been realized in OSCAR compiler, uses not only loop level parallelism but also coarse grain task parallelism among basic blocks, loops and subroutines and near fine grain task parallelism among statements. Based on the OSCAR multigrain parallelizing techniques, "Advanced Parallelizing Compiler(APC)" project[11] has been started since the autumn in 2000 as a part of Japanese Government Millennium project IT21. A target of this project is to develop a practical multigrain parallelizing compiler in cooperation with Government, Industry and Academia.

Also, PROMIS compiler[12] aims at integration of loop and instruction level parallelism using a common intermediate representation. NANOS compiler[13] exploits the multi level parallelism using extended OpenMP API.

This paper describes coarse grain task parallel processing considering cache optimization using data localization[10] in order to enhance the performance of coarse grain task parallel processing. The proposed scheme is implemented on OSCAR automatic multigrain parallelizing compiler. OSCAR compiler generates a parallelized FORTRAN program using OpenMP API[14, 15], which is a standard API for shared memory multiprocessor. OSCAR compiler realizes hierarchical coarse grain task parallel processing with cache optimization without special extension of OpenMP [16].

The rest of this paper is organized as follows. In section 2, coarse grain task parallel processing is described. Section 3 proposes cache optimization scheme for coarse grain task parallel processing. Section 4 describes the overview of OSCAR compiler. The effectiveness of the proposed schemes is evaluated on IBM RS6000 604e High Node using several benchmarks in SPEC95fp in section 5. Finally, conclusions are shown in section 6.

2 Coarse Grain Task Parallel Processing

This section describes coarse grain task parallel processing, which is a part of multigrain parallel processing. Coarse grain task parallel processing uses paral-

lelism among three kinds of macro-tasks or coarse grain tasks, namely block of pseudo assignment statements(BPA) repetition block(RB), subroutine block(SB). The compiler decomposes a source program into the macro-tasks. Also, it hierarchically generates macro-tasks inside inside of a sequential repetition block and a subroutine block.

Coarse grain task parallel processing in OSCAR compiler is performed in the following steps.

1. Decomposition of a source program into macro-tasks.
2. Analysis of data dependencies and control flows among macro-tasks and generation of Macro Flow Graph (MFG) that represents them.
3. Analysis of Earliest Execution Condition(EEC) that represents the condition on which macro-task may start its execution earliest and generation of Macro Task Graph (MTG).
4. Scheduling macro-tasks to processors or processor groups. When a macro-task graph has no conditional dependencies, macro-tasks are scheduled to processors or processor clusters at a compiler time and parallelized code is generated for each processor according to the scheduling results. When macro-task graph contains control dependencies, compiler generates dynamic scheduling routine to assign macro-tasks to processors or processor clusters at a run time and embeds the dynamic scheduling routine to the generated parallelized code with macro-task code in order to cope with runtime uncertainties.

2.1 Generation of macro-tasks

The compiler first generates macro-tasks namely block of pseudo assignment statements(similar to basic blocks), repetition blocks and subroutine blocks from a source program. Furthermore, compiler hierarchically decomposes the body of sequential repetition block and a subroutine block.

If a repetition block(RB) is a parallelizable loop, it is divided into partial loops by loop iteration direction taking into consideration the number of processors, cache size and so on. These partial loops are defined as different macro-tasks that are executed in parallel.

2.2 Generation of macro flow graph

After generation of macro-tasks, the data dependency and control flow among a macro-tasks for each layer are analyzed hierarchically, and represented by macro flow graph(MFG) as shown in Fig.1(a).

In the Fig. 1(a), 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 a macro-task. Though arrows of edges are omitted in the macro flow graph, it is assumed that the directions are downward.

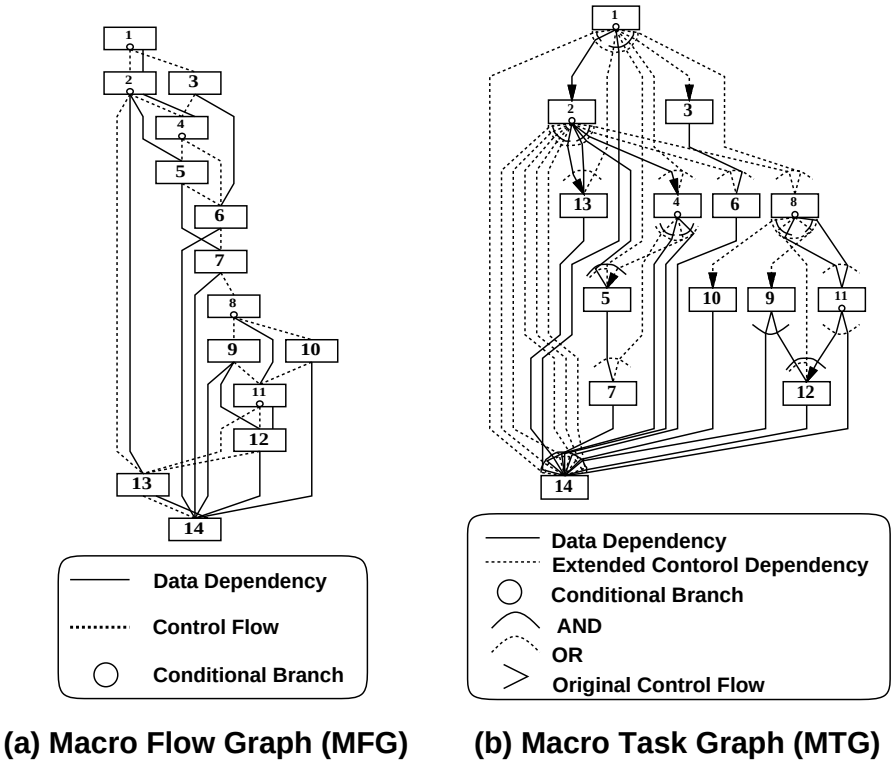


Fig. 1. Macro Flow Graph and Macro Task Graph

2.3 Generation of macro task graph

To extract parallelism among macro-tasks from macro flow graph, compiler analyses Earliest Executable Condition of each macro-task. Earliest Executable Condition represents the conditions on which macro-task may begin its execution earliest.

Earliest execution condition of macro-task is represented in macro task Graph(MTG) as shown in Fig. 1(b).

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 normal control dependency and the condition on which a data dependence predecessor macro-task 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 macro task graph, 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 Macro-Task Scheduling

In the coarse grain task parallel processing, static scheduling and dynamic scheduling are used for assignment of macro-tasks to processors or processor clusters. A suitable scheduling scheme is selected considering the shape of macro task graph and target machine parameters such as the synchronization overhead, data transfer overhead and so on.

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

Dynamic scheduling If a macro task graph has control dependencies, dynamic scheduling is selected to cope with runtime uncertainties like conditional branches. Scheduling routine for dynamic scheduling are generated and embedded into a parallelized program with macro-task code by compiler to eliminate the overhead for runtime thread scheduling.

Though dynamic scheduling overhead is generally large, the dynamic scheduling overhead in OSCAR compiler is relatively small since it is used for the coarse grain tasks with relatively large processing time.

There are two types of dynamic scheduling; Centralized dynamic scheduling and Distributed dynamic scheduling. The centralized dynamic scheduling routine is executed by one processor specified as the scheduler and other processors execute only macro-task code according to scheduling result. In the distributed dynamic scheduling, scheduling routines are distributed to the all processors with exclusive accesses to scheduling information such as ready task queues, earliest exclusive conditions and so on.

3 Cache Optimization For Coarse Grain Task Parallel Processing

This section describes the scheme to use cache effectively in order to enhance the performance of coarse grain task parallel processing.

If macro-tasks that access the same data are executed on the same processor as consecutively as possible, data can be transfered among these macro-tasks using fast memory near a processor such as cache, distributed shared memory or local memory.

To realize such task assignments, the data localization scheme[10] has been proposed.

In this paper, this data localization scheme is extended to use cache effectively on the shared memory machine. A task scheduler for coarse grain task parallel

processing is extended to assign macro-tasks that access the same data to be executed as consecutively as possible on the same processor considering task parallelism.

A simple example of the proposed scheme is shown in Fig. 2. In the macro task graph in Fig. 2(b), macro-task 1 and 6 access the same data. However, execution order in the original program is the increasing order of the task number as shown in Macro Flow Graph in Fig.2(a). Therefore, macro-task 2, 3, 4 and 5 are executed after macro-task 1 prior to the macro-task 6.In this case, shared data accessed by macro-task 1 may be forced out of cache by macro-task 2 through 5 before macro-task 6 is executed. However, because macro-task 6 depends on only macro-task 1 and 5, macro-task 6 can be executed immediately after macro-task 1 and macro-task 6 can access data in the cache.

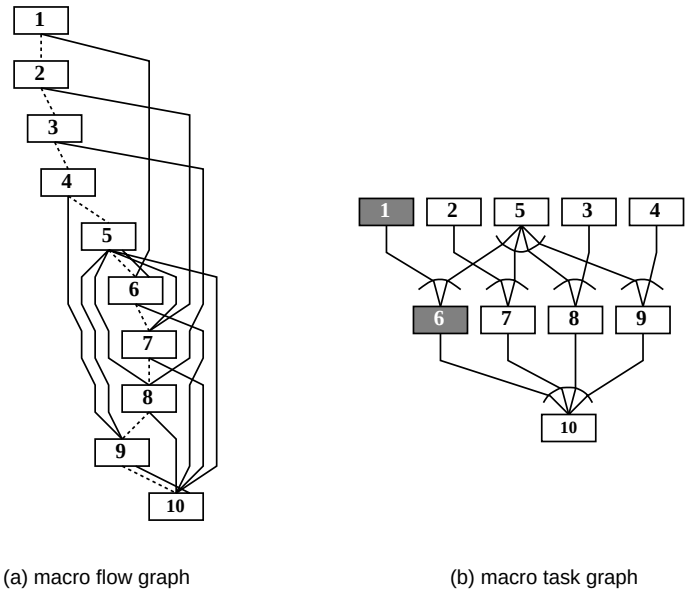


Fig. 2. An example of cache optimization for a macro task graph

The proposed cache optimization scheme using data localization mainly consists of two techniques; loop aligned decomposition and partial static task assignment.

3.1 Loop Aligned Decomposition

To avoid cache misses, Loop Aligned Decomposition(LAD)[17] is applied to loops that use large size of data. LAD divides a loop into partial loops with the smaller

number of iterations so that data size used in the divided loops is smaller than cache size.

The partial loops are treated as coarse grain tasks. Next, the partial loops connected by data dependence edge on the macro task graph are grouped into "Data Localization Group(DLG)" [10]. The partial loops, or tasks, inside a DLG are assigned to the same processor by static or dynamic scheduler.

In macro-task graph in Fig. 3(a), it is assumed that macro-tasks 2, 3 and 7 are parallel loops and they access the same data and its size exceeds cache size. In this example, these loops are divided into four partial loops by LAD. For example, macro-task 2 in Fig. 3(a) is divided into macro-task 2 through 5 in Fig. 3(b). In this case, the Data Localization Groups of macro-tasks with large share data are respectively (2, 6, 13), (3, 7, 14), (4, 8, 15), (5, 9, 16) in Fig. 3(b). In Fig. 3(b), the light gray band shows DLG. For example, DLG0 contains macro-tasks 2, 6 and 13 and DLG1 contains macro-tasks 3, 7 and 14 and so on.

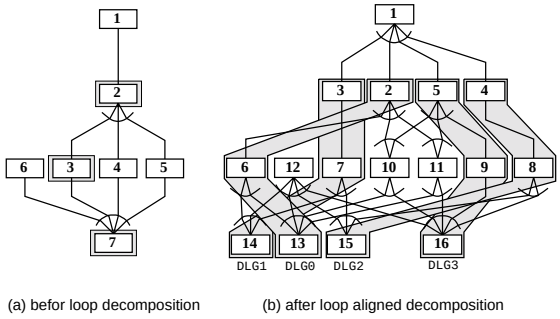


Fig. 3. Example of Loop Align Decomposition

3.2 Partial Static Assignment

As mentioned above, in the proposed cache optimization scheme, a task scheduler for coarse grain task parallel processing is extended to assign macro-tasks inside DLG to be executed on same processor consecutively.

This extension is called as partial static assignment[17] and it can be applied to both centralized and distributed dynamic scheduling. It is implemented in distributed dynamic scheduling routine at present in OSCAR compiler. This extended distributed dynamic scheduling routine with partial static assignment for cache is summarized as follows.

1 Execute its distributed scheduling routine to determine the macro-task to be executed next. This scheduler assigns, or acquires, macro-tasks outside DLG to own processor from the ready macro-task queue. At this time, if there is

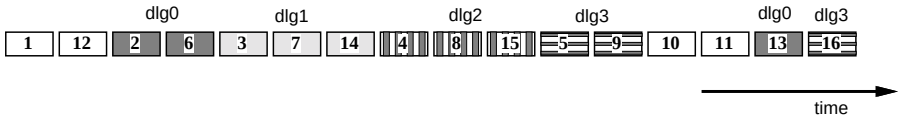


Fig. 4. An example of schedule for a single processor

no macro-task outside DLG in the ready macro-task queue, scheduler assigns macro-tasks inside DLG.

2 Execute the assigned macro-task. If an executed macro-task is the special macro-task which represents the end of the macro task graph, the execution of this hierarchy of the macro task graph is finished. Otherwise goto step3.

3 If the last executed macro-task is inside the DLG, goto 3.1, otherwise goto 3.2.

3.1 Assign macro-task in the same DLG from ready queue. If there is no macro-task inside the same DLG, assign macro-task outside DLG. If there is no ready macro-task outside DLG, assign macro-task in another DLG.

3.2 Assign macro-task outside DLG. If there is no such macro-task, assign macro-task inside the DLG.

4 Goto 2.

Fig. 4 shows a schedule when the proposed partial static task assignment for cache optimization is applied to macro task graph in Fig. 3(b) for a single processor. As described above, macro-tasks are executed in the increasing order of the node number on the macro task graph in original program. Fig. 4 shows that macro-tasks in the same DLG are executed consecutively to use cache effectively by using partial static assignment. As shown in Fig. 4, macro-task 3, 7 and 14 in DLG1 and macro-task 4, 8 and 15 in DLG2 are executed consecutively.

4 OSCAR Multigrain Parallelizing Compiler

Fig. 5 shows the overview of OSCAR compiler. It consists of frontend, middle path and backends. OSCAR compiler has various backends for different target multiprocessor systems like OSCAR type distributed/shared memory single chip multiprocessor system[18], UltraSparc, MPI-2 and OpenMP. OpenMP backend used in this paper generates the parallelized FORTRAN source code with OpenMP directives.

In OpenMP backend, OSCAR compiler is used as a preprocessor that transforms an ordinary sequential FORTRAN program to OpenMP FORTRAN program for shared memory multiprocessor system.

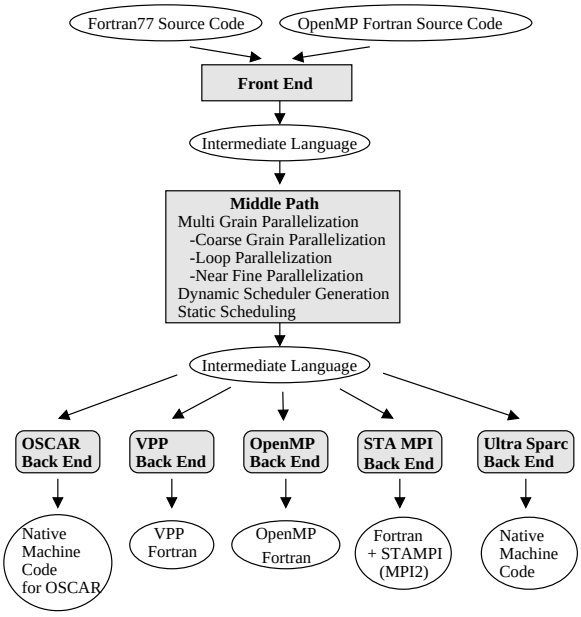


Fig. 5. Configuration of OSCAR compiler

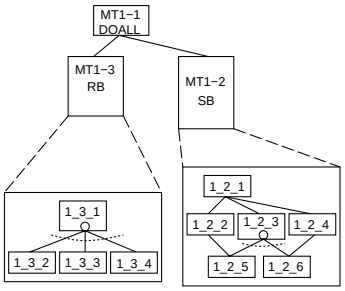
4.1 Realization of the Proposed Scheme Using OpenMP

This section describes the program generated by OSCAR compiler which realizes the proposed scheme using OpenMP API. A code image for eight threads generated by OpenMP backend for a macro task graph in Fig. 6(a) is shown in Fig. 6(b).

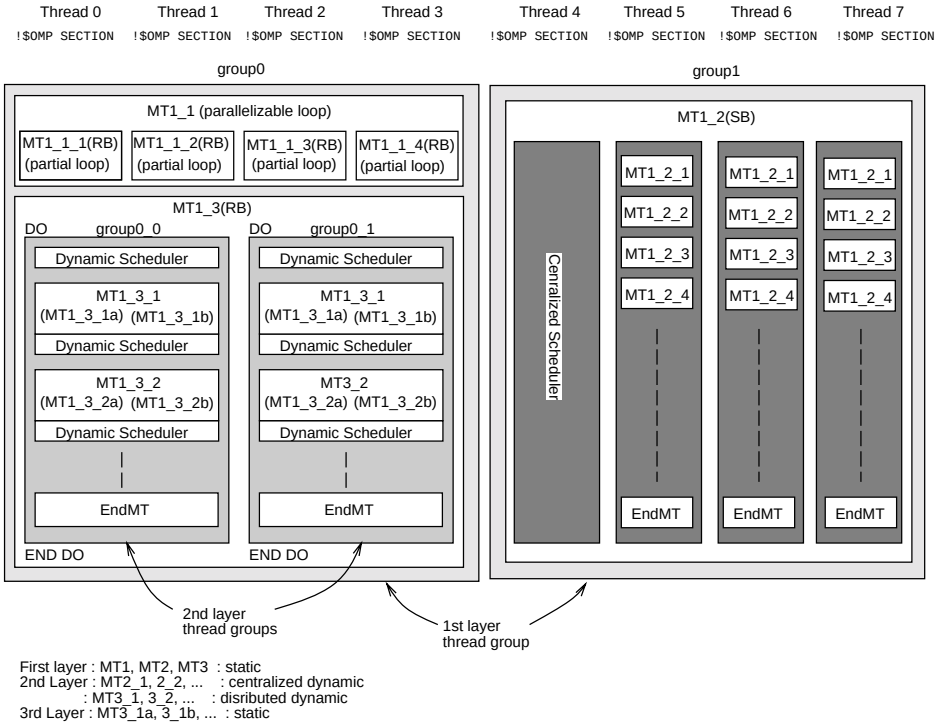
In this figure, eight threads are generated by OpenMP PARALLEL SECTIONS directive and these generated threads join only once at the end of program based on "one time single level thread generation"[16].

In this example, static scheduling is applied to the first layer. In this case, the eight threads are grouped into two thread groups, each of which has four threads. Macro-task 1_1 and 1_3 are statically assigned to thread group0 and macro-task 1_2 is assigned to thread group1. When static scheduling is applied, compiler generates different codes into each OpenMP SECTION for each thread according to the static scheduling result. The assigned macro-tasks to thread groups are processed in parallel by threads inside the thread group by using static scheduling or dynamic scheduling hierarchically.

Macro-task 1_2 in Fig. 6 assigned onto thread group1 is processed by four threads in parallel using the centralized dynamic scheduling. In this example, thread 4 works as the centralized scheduler and thread 5 to 7 execute sub macro-tasks 1_2_1, 1_2_2 and so on, which generated by decomposition of the inside of



(a) An example of macro task graph



(b) An image of generated parallel code

Fig. 6. Generated code image using OpenMP (eight threads)

macro-task 1_2, according to the dynamic scheduling result of the centralized scheduler.

Macro-task 1_3 shows an example of distributed dynamic scheduling. In this case, macro-task 1_3 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. The distributed dynamic scheduling routines that perform partial static task assignment for cache optimization are embedded into before each macro-task code as shown in Fig. 6. Furthermore, Fig. 6 shows macro-task 1_3_1, 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 result of the performance evaluation of the proposed scheme on a commercial SMP machine, IBM RS6000 SP 604e High Node. The generated OpenMP FORTRAN programs by OSCAR compiler are compiled by IBM XL FORTRAN compiler version 6.1 for RS6000. RS6000 used in this evaluation has eight 200MHz PowerPCs each of which has 32KB L1 instruction and data cache respectively, 2MB unified L2 cache per two processors and 512MB main memory. Programs used for this evaluation are tomcatv, swim and mgrid in SPEC95fp benchmarks.

5.1 Tomcatv

Tomcatv is a vectorized mesh generation program. The convergence loop in main routine spends nearly 99% of execution time. There are several loops inside the body of convergence loop and these loops access shared data which are larger than cache size. Therefore cache optimization is applied these loops. The obtained speedups against sequential processing are shown in Fig. 7.

The sequential execution time of tomcatv was measured with XL FORTRAN compiler option "-O3 -qhot -qmaxmem=-1 -qarch=auto -qtune=auto -qcach=auto"and it was 693 seconds. The execution times of automatic loop parallelization by XL FORTRAN compiler were 370 seconds for 2PEs, 233 seconds for 4PEs and 177 seconds for 8PEs. When OSCAR compiler was used as a preprocessor of XL FORTRAN compiler, the execution times were reduced to 523 seconds for 1PE, 275 seconds for 2PEs, 147 seconds for 4PEs and 86.8 seconds for 8PEs.

5.2 Swim

Swim solves the system of shallow water equation using difference approximations and has large loop level parallelism. The most execution time is spent in three subroutines, namely "calc1", "calc2", "calc3" called from main loop. These subroutines contain several loops which access larger data than cache size. Therefore, coarse grain task parallel processing with cache optimization is applied to these loops inside subroutines. The evaluation results are shown in Fig. 8

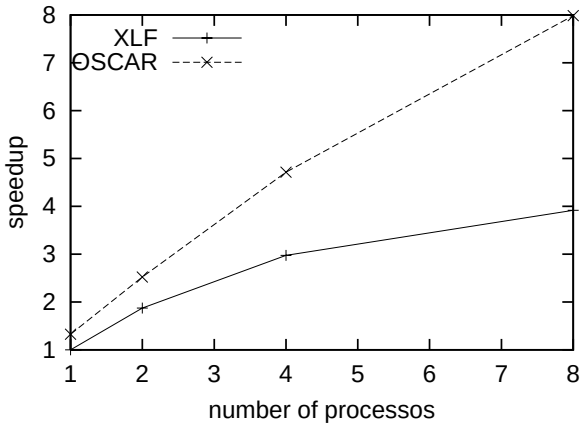


Fig. 7. Speedup of tomcatv

When swim was compiled by only XL FORTRAN compiler for single processor, the sequential execution time was 521 seconds and the execution times were 277 seconds for 2PE, 183 seconds for 4PE, 169 seconds for 8PE by using XLF FORTRAN compiler automatic parallelization. The execution times of OSCAR compiler were 443 seconds for 1PE, 221 seconds for 2PEs, 113 seconds for 4PEs and 60.2 seconds for 8PEs.

5.3 Mgrid

Mgrid is the 3 dimensional multi-grid solver. About 70% of execution time is spent in subroutine "resid" and "psinv". These subroutines contain loops which access larger data than the cache size. Therefore, the proposed coarse grain task parallel processing with cache optimization is applied inside these subroutines. Fig. 9 shows the speedups against sequential processing by XL FORTRAN compiler.

Mgrid by XL FORTRAN compiler needed 676 seconds for a single processor. The execution times of OSCAR compiler were 637 seconds for 1PE,352 seconds for 2PEs, 174 seconds on 4PE and 94.5 seconds on 8PEs in Mgrid. Scalable speedups was obtained and it was shown that OSCAR compiler gave us lager speedup than XL FORTRAN compiler which spent 552 seconds for 2PEs, 594 seconds for 4PEs and 722 seconds for 8PEs.

6 Conclusions

This paper has proposed coarse grain task parallel processing with cache optimization to enhance the effective performance of shared memory multiprocessor

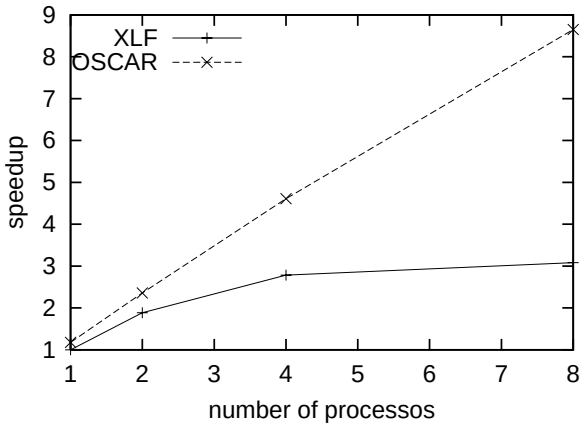


Fig. 8. Speedup of swim

systems. The proposed schemes are implemented in OSCAR automatic multigrain parallelizing compiler. OSCAR compiler generates parallelizing code using ordinary OpenMP API for portability.

In the performance evaluation, several programs in SPEC95fp were parallelized by OSCAR compiler and were run on the commercial shared memory multiprocessor IBM RS 6000 SP 604e High Node. Evaluation results show that coarse grain task parallel processing with cache optimization gave us speedups for tomcatv, swim and mgrid on 8 processors against 8.0, 8.6 and 7.2 respectively. Furthermore, compared with XL FORTRAN loop parallelizing compiler 2.0, 2.8 and 7.6 times speedups were obtained.

A part of this research has been supported by METI/NEDO Millennium project IT21 "Advanced Parallelizing Compiler".

References

1. 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.
2. P. Tu and D. Padua. Automatic array privatization. *Proc. 6th Annual Workshop on Languages and Compilers for Parallel Computing*, 1993.
3. 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.
4. M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, 1996.
5. 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.

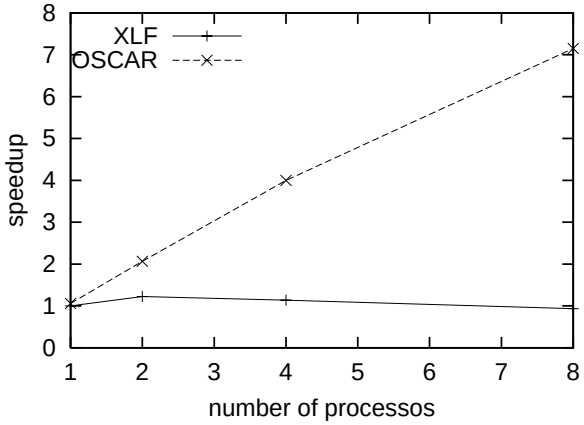


Fig. 9. Speedup of mgrid

6. 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.
7. 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.
8. H. Han, G. Rivera, and C-W. Tseng. Software support for improving locality in scientific codes. *8th Workshop on Compilers for Parallel Computers*, Jan. 2000.
9. G. Rivera and C-W. Tseng. Locality optimizations for multi-level caches. *Super Computing '99*, Nov. 1999.
10. A. Yoshida, Y. Ujigawa, M. Obata, K. Kimura, and H. Kasahara. Data-localization among doall and sequential loops in coarse grain parallel processing. *Seventh Workshop on Compilers for Parallel Computers*, Jul. 1998.
11. Advanced Parallelizing Compiler Project. http://www.apc.waseda.ac.jp/.
12. C. J. Brownhill, A. Nicolau, S Novack, and C. D. Polychronopoulos. Achieving multi-level parallelization. *Proc. of the International Symposium on High Performance Computing*, 1997.
13. 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.
14. Portable Scalable SMP Programing OpenMP: Simple. http://www.openmp.org/.
15. L. Dagum and R. Menon. Openmp: An industry standard api for shared memory programming. *IEEE Computational Science & Engineering*, 1998.
16. 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.
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. 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.