Automatic Coarse Grain Task Parallel Processing on SMP Using OpenMP

Hironori Kasahara, Motoki Obata, and Kazuhisa Ishizaka

Waseda University
3-4-1 Ohkubo, Shinjuku-ku, Tokyo, 169-8555, Japan
{kasahara,obata,ishizaka}@oscar.elec.waseda.ac.jp

Abstract. This paper proposes a simple and efficient implementation method for a hierarchical coarse grain task parallel processing scheme on a SMP machine. OSCAR multigrain parallelizing compiler automatically generates parallelized code including OpenMP directives and its performance is evaluated on a commercial 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 scheme decomposes a Fortran program into coarse grain tasks, analyzes parallelism among tasks by "Earliest Executable Condition Analysis" considering control and data dependencies, statically schedules the coarse grain tasks to threads or generates dynamic task scheduling codes to assign the tasks to threads and generates OpenMP Fortran source code for a SMP machine. The thread parallel code using OpenMP generated by OSCAR compiler forks threads only once at the beginning of the program and joins only once at the end even though the program is processed in parallel based on hierarchical coarse grain task parallel processing concept. The performance of the scheme is evaluated on 8-processor SMP machine, IBM RS6000 SP 604e High Node, using a newly developed OpenMP backend of OSCAR multigrain compiler. The evaluation shows that OSCAR compiler with IBM XL Fortran compiler version 5.1 gives us 1.5 to 3 times larger speedup than the native XL Fortran compiler for SPEC 95fp SWIM, TOMCATV, HYDRO2D, MGRID and Perfect Benchmarks ARC2D.

1 Introduction

The loop parallelization techniques, such as Do-all and Do-across, have been widely 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].

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 189–207, 2001.

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]. Effective optimization of data localization is getting more important because of the increasing disparity between memory and processor speeds. Currently, 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 [16, 18, 19, 20].

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.

PROMIS compiler[21, 22] hierarchically combines Parafrase2 compiler[23] using HTG[24] and symbolic analysis techniques[6] and EVE compiler for fine grain parallel processing. NANOS compiler[25, 26] based on Parafrase2 has been trying to exploit multi-level parallelism including the coarse grain parallelism by using extended OpenMP API[27, 28]. OSCAR compiler has realized a multi-grain parallel processing [29, 30, 31] that effectively combines the coarse grain task parallel processing [29, 30, 31, 32, 33, 34], which can be applied for a single chip multiprocessor to HPC multiprocessor systems, the loop parallelization and near fine grain parallel processing[35]. In OSCAR compiler, coarse grain tasks are dynamically scheduled onto processors or processor clusters to cope with the runtime uncertainties caused by conditional branches by dynamic scheduling routine generated by the compiler. As the embedded dynamic task scheduler, the centralized dynamic scheduler[30, 31] in OSCAR Fortran compiler and the distributed dynamic scheduler[36] have been proposed.

A coarse grain task assigned to a processor cluster is processed in parallel by processors inside the processor cluster with the use of the loop, the coarse grain and near fine grain parallel processing hierarchically.

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 is generated. In other words, OSCAR Fortran Compiler is used as a preprocessor 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. Though OpenMP API is chosen as the thread creation method because of the

portability, the proposed 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 a result of data dependency and control flow analysis among macro-tasks.
3. Generation of Macro-Task Graph by analysis of parallelism among macro-tasks using Earliest Executable Condition analysis [29, 32, 33].
4. If a macro-task graph has only data dependency edges, macro-tasks are assigned to processor clusters or processor elements by static scheduling. If a macro-task graph has both data dependency and control dependency edges, macro-tasks are assigned to processor clusters or processor elements at runtime 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.

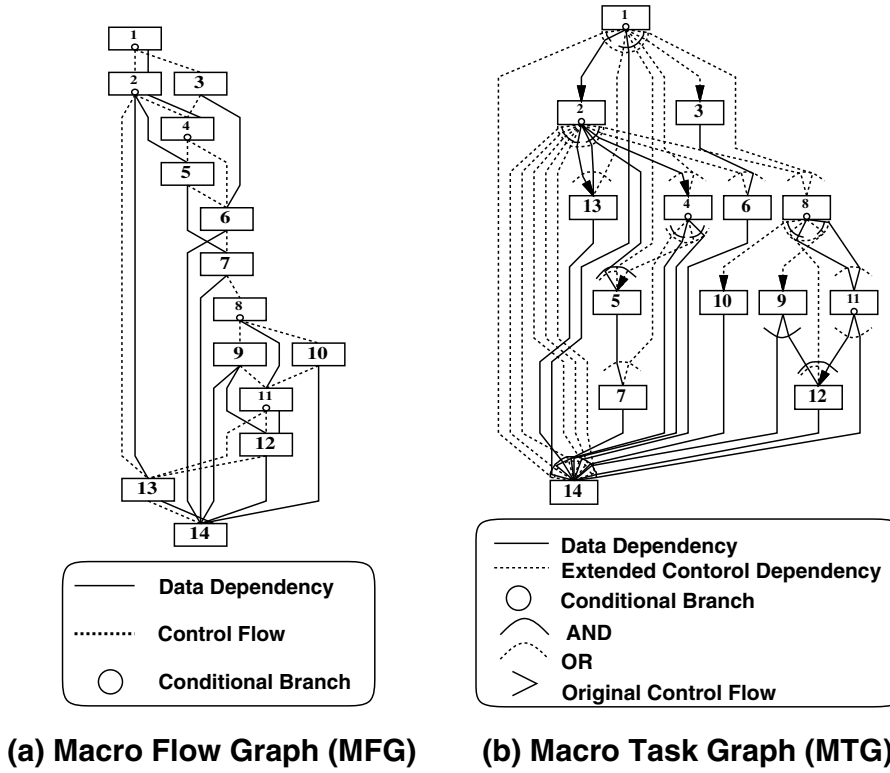

Fig. 1. Macro flow graph and macro-task graph

If there is a prallelizable 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 or processor elements and cache or memory size.

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

2.2 Generation of Macro-flow Graph

Next, the date dependency and control flow among macro-tasks for each nest level are analyzed hierarchically. The control flow and data dependency among macro-tasks are represented by macro-flow graph as shown in Fig.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 applied. 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 data-depends on MTj, MTi cannot 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 Executable Condition is represented as follows;

(Macro-Task(MT)j, on which MTi is control dependent,
takes a branch that guarantees MTi will execute)
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 Executable Condition of MT6 on Fig.1(b) is

(MT1 takes a branch that guarantees MT3 will be execute
OR MT2 takes a branch that guarantees MT4 will be execute)
AND
(MT3 completes execution
OR MT1 takes a branch that guarantees MT4 will be execute).

However, the completion of MT3 means MT1 already took the branch to MT3. Also, "MT2 takes a branch that guarantees MT4 will execute" means that MT1 already branched to MT2. Therefore, this condition is redundant and its simplest form is

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

Earliest Executable Condition of macro-task is represented in a macro-task graph as shown in Fig.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 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 macro-flow graph.

2.4 Generation of Scheduling Routine

In the coarse grain task parallel processing, the dynamic scheduling and the 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 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 assignment. There are two kinds of schemes for dynamic scheduling, namely, a centralized dynamic scheduling, in which the scheduling routine is executed by a processor element, and a distributed scheduling, in which the scheduling routine is distributed to all processors.

Also, in static scheduling, assignment of macro-tasks to processor clusters or processor elements is determined at compile-time inside auto parallelize compiler if macro-task graph 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.

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 only 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.

Though OpenMP is used as a method of the thread generation in this implementation because of its high portability, the proposed scheme can be used with other thread creation methods as well.

3.1 Generation of Threads

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

Generally, upper level threads fork nested threads to realize nested or hierarchical parallel processing.

However, the proposed scheme realizes this hierarchical parallel processing with single level thread generation by writing all hierarchical behavior, or by embedding hierarchical scheduling routines, in each 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 any language extension.

3.2 Macro-task Scheduling

This section describes code generation scheme using static and dynamic scheduling to assign macro-tasks to threads or thread groups hierarchically.

In the coarse grain task parallel processing by OSCAR compiler, the macro-tasks are assigned to threads or thread groups at run-time and/or at compilation-time. OSCAR compiler can choose the centralized dynamic scheduling and/or the distributed dynamic scheduling scheme in addtion to static scheduling. These scheduling methods are suitably used considering parallelism of the source program, a number of processors, data transfer and syncronization overhead of a target multiprocessor system with their any 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. More concretely, the compiler chooses static scheduling for a macro-task graph with data dependence edges and dynamic scheduling for a macro-task graph with control dependence edges in each layer, or nest level. Also, centralized scheduler is usually chosen for a processor, or a thread group, and distributed scheduler is chosen for a processor cluster with low mutual exclusion overhead to shared scheduling information.

Those scheduling methods can be hierarchically combined freely depending on program parallelism, the number of processors available for the program layer, syncronization overhead and so on.

Centralized Dynamic Scheduling. In centralized scheduling scheme, one thread in a parallel processing layer choosing centralized scheduling serves as centralized scheduler, which assigns macro-tasks to thread groups, namely, processor clusters or processor elements. This thread is called scheduler or Centralized Scheduler.

The behavior of *CS* 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, and enqueue ready macro-tasks, which satisfy this condition, to a ready task queue.

step3 Find a processor cluster, or a thread group, to which a ready macro-task should be assigned according to the priority of Dynamic CP method.[29]
step4 Assign macro-task to the processor cluster or the processor element. If the assigned macro-task is "End MT"(EMT), the centralized scheduler finishes scheduling routine in the layer.
step5 Jump to step1.

In the centralized scheduling scheme, ready macro-tasks are initially assigned to thread groups. Each thread group executes these macro-tasks at the beginning. When macro-task finishes execution or determines branch direction, it sends signal to the centralized scheduler. Therefore the centralized scheduler busy waits for these signals at the start.

If the centralized scheduler receives these signals, it searches new executable, or ready, macro-tasks by checking Earliest Executable Condition for each macro-task.

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-task and goes back to the signal waiting routine.

On the other hand, slave threads execute macro-tasks assigned by the centralized scheduler. Their behavior is summarized in following.

step1 Wait macro-task assignment information from the centralized scheduler
step2 Execute the assigned macro-task
step3 Go back to step1

So, at the beginning of OpenMP "SECTION" or a thread, slave thread executes the busy/wait code for waiting assignment information from the centralized scheduler if a macro-task isn't assigned initially.

In the dynamic scheduling mode, since every thread or thread group has possibility to execute all macro-tasks, the whole code including all macro-tasks is copied into each OpenMP "SECTION" for each slave thread.

Figure 2 shows an image of generated OpenMP code for every thread. Sub macro-tasks generated inside of Macro-Task(MT)3, a macro-task in the 1st layer, are represented as macro-tasks in the 2nd layer, MT3_1, MT3_2, and so on. The 2nd layer macro-tasks are executed on "k" threads for program execution and one thread to serve as a centralized dynamic scheduler. Moreover, MTs like MT3_2_1 and MT3_2_2 in the 3rd layer are generated inside MT3 in the 2nd layer. Figure 2 exemplifies a code image in a case where the 3rd layer MTs like MT3_2_1 and MT3_2_2 are dynamically scheduled to threads by the distributed scheduler. The details of distributed scheduling are described later.

After the completion of the execution of a macro-task, the slave threads go back to the routine to wait for the assignment of a macro-task by a centralized scheduler.

Also, the compiler generates a special macro-task called "End MT"(EMT) in each layer. As shown in the 2nd layer in Fig.2, the EndMT is written at the end of all OpenMP "SECTION", and *CS* assigns EMT to the all thread groups when

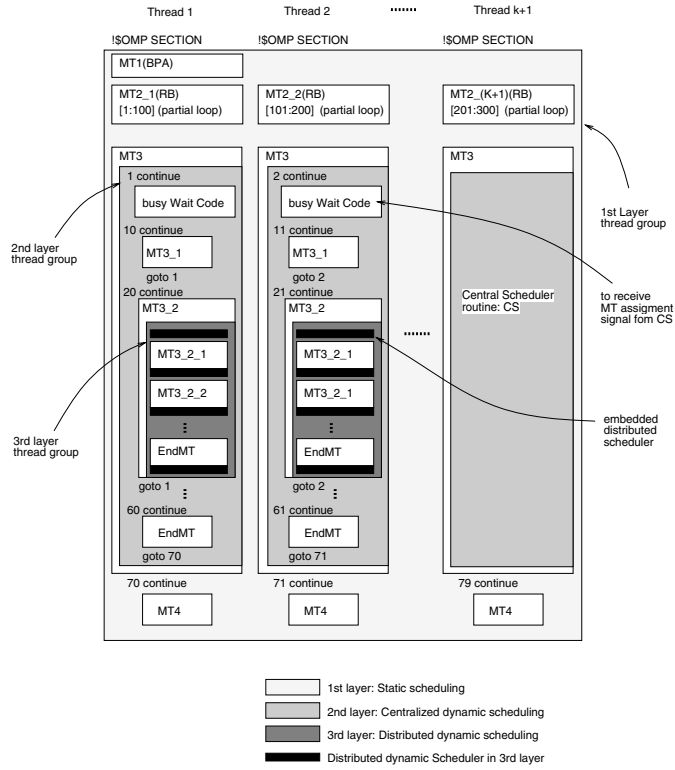

Fig. 2. Code image ($k$threads)

all of the threads executing the same hierarchy are finished. After assigning EMT to the all threads, *CS* finishes scheduling routine in the layer. Each thread group jumps to outside of its hierarchy. If the hierarchy is a top layer, the program finishes the execution. If there exists an upper layer, threads continue to execute the upper layer macro-tasks.

In the example shown in Fig.2, thread1 finishes the execution in 2nd layer by "goto 70" after "End MT" and jumps out to a statement"70 continue" in the upper layer.

Distributed Dynamic Scheduling. Each thread group or processor cluster schedules a macro-task to itself and executes the macro-task in the layer where the distributed dynamic scheduler is chosen.

In the distributed scheduling scheme, all shared data for scheduling are assigned onto shared memory and accessed exclusively.

step1 Search executable, or 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 queue with exclusive access to data on shared memory required for dynamic scheduling.

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

step3 Execute the macro-task

step4 Update the Earliest Executable Condition table exclusively.

step5 Go back to step1

For example, the 3rd layer shown in Fig.2 uses the distributed dynamic scheduling scheme. In this example, the distributed scheduling is applied to inside of MT3_2 in second layer, and two thread groups that consist of one thread are realized in this layer by only executing the thread code generated by compiler.

Static Scheduling Scheme. If a macro-task graph in a target hierarchy has only data dependencies, the static scheduling is applied to reduce data transfer, synchronization and scheduling overheads.

In the static scheduling, the assignment of macro-tasks to processor clusters or processor elements is determined at compile-time. Therefore, each OpenMP "SECTION" needs only the macro-tasks that should be executed in the order predetermined 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 first layer of Fig.2. When this static scheduling is used, it is assumed that each thread is binded to a processor.

In the OSCAR compiler, all of those algorithms are applied to the same macro-task graph, and the best schedule is automatically chosen.

At runtime, each thread group needs to synchronize and transfer shared data among other thread groups in the same hierarchy to satisfy the data dependency among macro-tasks.

To realize the data transfer and synchronization, the following code generation scheme is used.

If a macro-task assigned to the thread group is Basic Block, only one thread in the thread group executes the Basic Block(BB) or Basic Pseudo Assignment(BPA). Therefore, the code for MT1(BPA) is written in an OpenMP "SECTION" for one thread as shown in the first layer in Fig.2. OpenMP "SECTIONS" for the other threads don't have the code for MT1(BPA).

A parallel loop, for example RB2, is decomposed into smaller loops like MT2_1 to MT2_(K+1) at compile-time and defined as independent MTs as shown in Fig.2. Each thread has a code for assigned RBs, or decomposed partial parallel loops. In this case, since the partial loops are different macro-tasks, barrier synchronization at the end of the original parallel loops is not required.

If a macro-task is a sequential loop to which data localization cannot be applied, only one thread or thread group executes it. The sequential loop, or a RB, assigned to a processor cluster has large execution cost, its body hierarchically decomposed into coarse grain tasks. In Fig.2, MT3 is a sequential loop and decomposed into MT3_1, MT3_2, and so on.

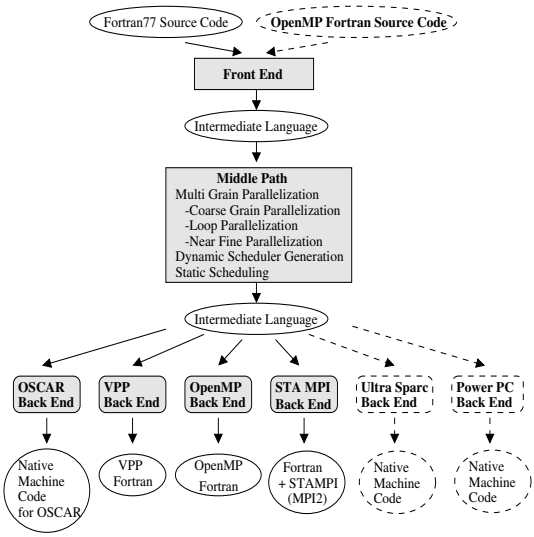

Fig. 3. Overview of OSCAR Fortran Compiler

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

Figure 3 shows the overview of OSCAR Fortran Compiler. It consists of Front End, Middle Path and Back Ends. OSCAR Fortran Compiler has various Back Ends for different target multiprocessor systems like OSCAR distributed/shared memory multiprocessor system[37], Fujitsu's VPP supercomputer, UltraSparc, PowerPC, 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 preprocessor that transforms from an ordinary sequential Fortran program to OpenMP Fortran program for SMP machines.

4.2 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

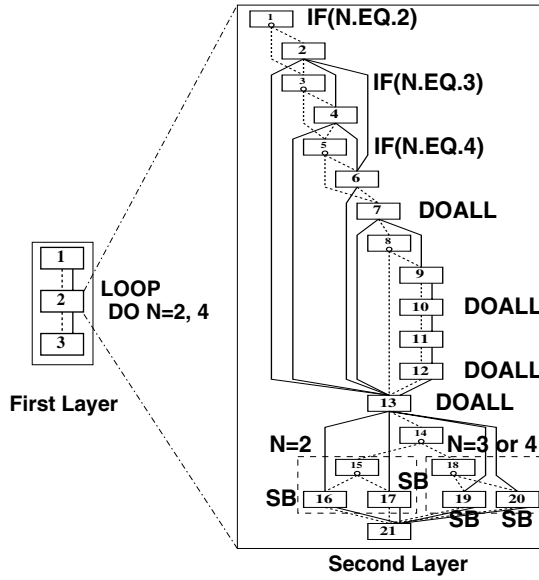

Fig. 4. Macro-flow graph of subroutine STEPFX in ARC2D

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.

As an example of the exploitation of coarse grain parallelism, parallelization of ARC2D is briefly explained. ARC2D has about 4500 statements including 40 subroutines. More than 90% of the execution time is spent in subroutine INTEGR. In the subroutine INTEGR, subroutines FILERX, FILERY, STEPFX, STEPFY consume 62% of the total execution time. Here, as an example, subroutine STEPFX is treated since it consumes 30% of the total execution time.

Macro Flow Graph of subroutine STEPFX is shown in Fig.4. The first layer in Fig.4 consists of three macro-tasks. Macro-Task(MT) 2 in the first layer is a sequential loop having three iterations with large processing time.

To minimize the processing time of MT2, the second layer macro-tasks are defined for a loop body of MT2, such as twelve Basic Blocks (MT1∼6, 8, 9, 11, 14, 15, 18), four DOALL loops (MT7, 10, 12, 13) and four Subroutine Blocks (MT16, 17, 19, 20).

In the second layer of Fig.4, MT groups, namely, MTs 1 and 2, MTs 3 and 4, MTs 5 and 6, MTs 15, 16 and 17 and MTs 18, 19 and 20, are executed depending on a value of loop index N. For example, MT2 is executed by the result of conditional branch inside MT1 (a Basic Block) when the value of loop index N is 2. MT4 is executed when N=3. MT6 is executed when N=4. By the

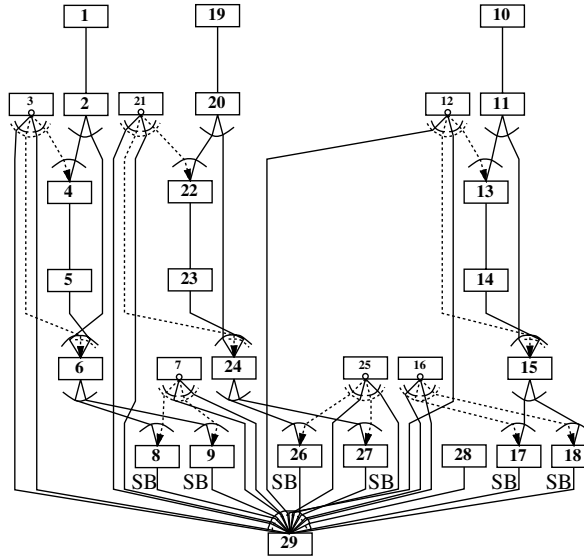

Fig. 5. Optimized macro-task graph of subroutine STEPFX in ARC2D

result of conditional branch inside MT14 depending on loop index, MT15,16 and 17 are executed when N=2, and MT18,19 and 20 are executed when N=3 and 4. Also, MT15 and 18 in the second layer of Fig.4 are Basic Blocks having conditional branches depending on the input variable from files. By the result of conditional branches inside MT 15 and 18, either MTs 16 and 19 or MTs 17 and 20 are executed.

At first, loop unrolling is applied to MT2 having 3 iterations, in the first layer of Fig.4. As the result of loop unrolling, OSCAR compiler can remove conditional branches inside MT 1, 3, 5 and 14 in the second layer of Fig.4 depending on loop index N. By loop unrolling, SB16, 17, 19 and 20 in the second layer are transformed to SB8, 9, 17, 18, 26 and 27 in Fig.5 . Either macro-task group composed of SBs 8, 17 and 26 or SBs 9, 18 and 27 in Fig.5 is executed by conditional branches inside MT7, 16 and 25. As the result of interprocedure analysis to these subroutine blocks, subroutine blocks inside each group can be processed in parallel. Figure 5 shows the optimized Macro-Task Graph.

OSCAR compiler applies the similar optimizations to other subroutines called from subroutine INTEGR. In addition, the inline expansion are applied to the subroutine calls in subroutine INTEGR for extracting more coarse grain parallelism.

Figure 6 shows the optimized Macro-Task Graph for the subroutine INTEGR.

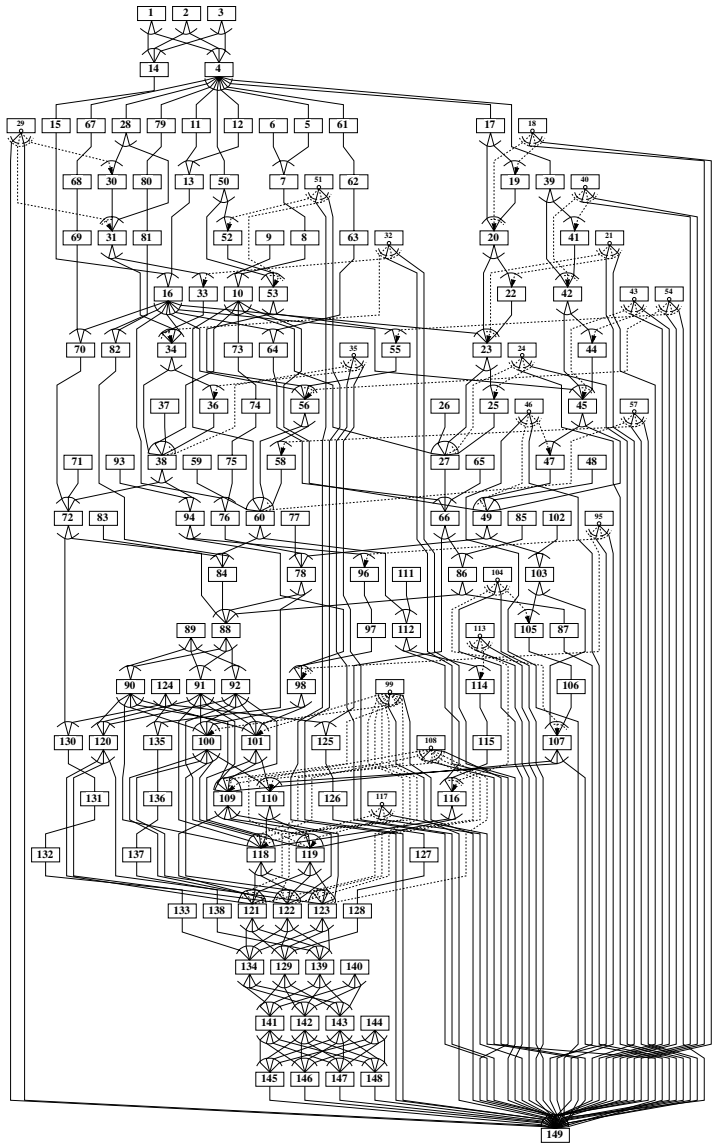

Fig. 6. Optimized macro-task graph of subroutine INTEGR in ARC2D.

4.3 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.

4.4 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[38] and executed on 1 through 8 processors of RS6000 SP 604e High Node. The performance of OSCAR compiler with XL Fortran compiler is compared with native IBM XL automatic parallelizing Fortran compiler[39]. In the compilation by a XL Fortran, maximum optimization option "-qsmp=auto -O3 -qmaxmem=-1 -qhot" is used.

Figure 7(a) 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. 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 number of loop iterations consuming large execution time in ARC2D has a small number of iterations. Therefore, only use of the loop level parallelism cannot attain large performance improvement even if more than 4 or 5 processors are used. The performance difference between OSCAR compiler and XL Fortran compiler in Fig.7(a) come from the coarse grain parallelism detected by OSCAR compiler.

Next, Fig.7(b) 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 7(c) 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 7(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 processors compared with the sequential execution time, OSCAR Fortran compiler gave us 8.1 times speed-up (128s) compared with sequential execution time.

Finally, Fig.7(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 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.

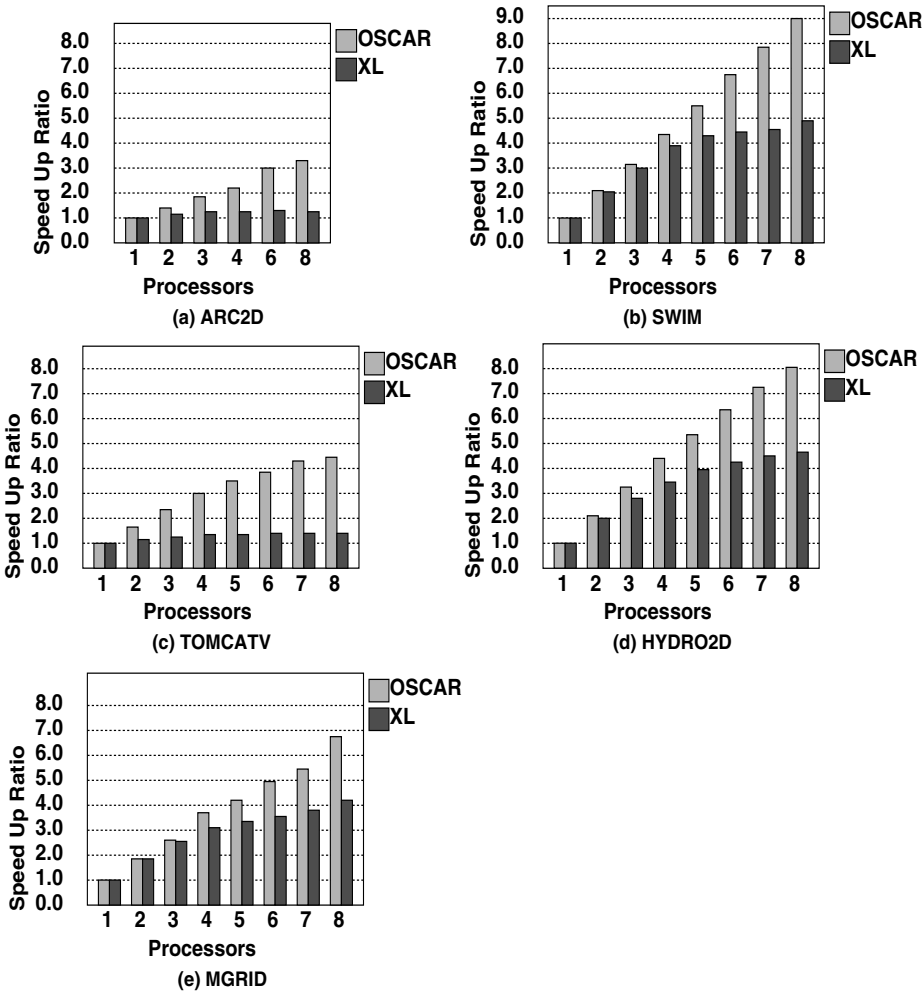

Fig. 7. Speed-up of several benchmarks on RS6000

5 Conclusions

This paper has presented the implementation scheme of the automatic coarse grain task parallel processing using OpenMP API as an example of realization and its performance on an off the shelf SMP machine.

OSCAR compiler generates coarse grain parallelized code which forks threads only once at the beginning of a program and joins only once at the end to minimize the overhead though hierarchical coarse grain task parallelism are automatically exploited.

In the performance evaluation, OSCAR compiler with XL Fortran compiler gave us scalable speed up for 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 improving the implementation scheme to further reduce dynamic scheduling overhead and data transfer overhead using data localization scheme to use distributed cache efficiently.

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] H. Han, G. Rivera, and C.-W. Tseng. Software Support for Improving Locality in Scientific Codes. *8th Workshop on Compilers for Parallel Computers (CPC'2000)*, Jan. 2000.
[19] G. Rivera and C.-W. Tseng. Locality Optimizations for Multi-Level Caches. *Super Computing '99*, Nov. 1999.
[20] 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.
[21] PROMIS. http://www.csrd.uiuc.edu/promis/.
[22] C. J. Brownhill, A. Nicolau, S. Novack, and C. D. Polychronopoulos. Achieving Multi-level Parallelization. *Proc. of ISHPC'97*, Nov. 1997.
[23] Parafrase2. http://www.csrd.uiuc.edu/parafrase2/.
[24] 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.
[25] 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.
[26] 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.
[27] OpenMP: Simple, Portable, Scalable SMP Programming http://www.openmp.org/.
[28] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared Memory Programming. *IEEE Computational Science & Engineering*, 1998.
[29] H. K. et al. A Multi-grain Parallelizing Compilation Scheme on OSCAR. *Proc. 4th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1991.
[30] 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.
[31] 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.
[32] 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.
[33] 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.

[34] H. Kasahara. Parallel Processing Technology. *Corona Publishing*, Tokyo (in Japanese), Jun. 1991.
[35] 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.
[36] J. E. Moreira and C. D. Polychronopoulos. Autoscheduling in a Shared Memory Multiprocessor. *CSRD Report No.1337*, 1994.
[37] H. Kasahara, S. Narita, and S. Hashimoto. OSCAR's Architecture. *Trans. IEICE (in Japanese)*, J71-D-I(8), Aug. 1988.
[38] IBM. *XL Fortran for AIX Language Reference*.
[39] 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.