Advanced Parallelizing Compiler Technology

2001 Annual Report

March 2002

Introduction

The Advanced Parallelizing Compiler (APC) project is the Millennium Project of the Ministry of Economy, Trade and Industry, a joint public-private research and development effort. Based on the Industrial Science and Technology Research and Development System (ISTRDS), a system for nurturing new industries, a body was created under the auspices of the Industrial Technology General Development Framework to conduct R&D in advanced parallelizing compiler technology beginning September 2000. Headed by Project Leader (PL) Dr. Hironori Kasahara, Professor at the Faculty of Science and Engineering of Waseda University with Sub Leaders (SLs) Prof. Hayato Yamana (Waseda University) and Dr. Hanpei Koike (AIST) and Group Leaders (GLs) Mr. Kohichiro Hotta (JIPDEC/Fujitsu) and Mr. Tokuro Anzaki (JIPDEC/Hitachi), the body consists of the Japan Information Processing Development Corporation, a team of 21 researchers from Hitachi, Ltd. and Fujitsu Ltd. nominated by JIPDEC, two collaborating research organizations (National Institute of Advanced Industrial Science and Technology(AIST) and Waseda University) and subcontractors University of Electro-Communications, Tokyo Institute of Technology, and Toho University.

The target of this project is to strengthen international competitive power in computer and related IT fields by doubling effective performance of multi-processor systems (computers that organically connect multiple CPUs to deliver high computing performance and are expected to be utilized for wide range of information processing systems including future microprocessors, various portable information devices, home-servers and so on) and improving the cost-performance and ease of use.

The key technology for improving the effective performance, cost effectiveness and ease of use is the automatic parallelizing compiler technology. The project aims at attaining about twice performance compared with automatic loop parallelizing compilers available in the market by using the innovative multigrain parallelizing compilation scheme launched from Japan. This target also means to double effective performance obtained on the same shared memory multiprocessor hardware or decrease manufacturing cost remarkably by reducing the number of processors to deliver the same performance by half.

Furthermore, this project has been researching and developing performance evaluation technology to fairly evaluate the accomplishment of the project target since there has been no scheme to judge the accomplishment of the numerical target for parallelizing compilers. So far, efforts for selecting programs for the performance

evaluation and defining performance measurement schemes have been made. It has been decided to choose suitable programs for performance evaluation of a multigrain parallelizing compiler out of world standard benchmark programs, such as SPEC and NAS Parallel Benchmarks.

Given the intense competition in the field of high-performance computing R&D, the project team aims to complete its work within a mere three years. APC members have been doing their best to attain world leading very difficult target fighting with current budget condition though several compiler functions must be degraded.

Some of major technical accomplishments in FY2001 are listed below:

- Manual implementation of the proposed automatic data distribution scheme using First Touch method for indirect array reference gave us 5.9 times speedup for NAS Parallel Benchmark CG (Class B) against SGI parallelizing compiler on 32 processor SGI Origin 2000 distributed shared memory multiprocessor server.
- Manual implementation of the proposed medium grain parallelism exploitation techniques realizing pipeline parallel processing with data locality optimization gave us 2.7 times speedup for SPEC2000 applu against Sun Forte parallelizing compiler on a 8 processor SUN SMP system and 2.4 times speedup against Compaq parallelizing compiler on Compaq Alpha Server GS160 Model6/73 8 processor server.
- The proposed multigrain parallelism exploitation infrastructure technology gave us 3.3 times speedup for SPEC95 tomcatv program with a little manual program restructuring against IBM XL parallelizing compiler, 1.9 times speedup for swim, 1.7 times speed up for su2cor, 4.3 times for mgrid and 1.7 times speedup for Perfect club benchmark arc2d on IBM 8 processor SMP server RS6000 604e high node.

These very good results give us bright perspective for the target accomplishment in the final year.

Parts of this year's accomplishment are presented as 8 reviewed full papers, a reviewed symposium paper, 13 technical reports, 6 annual convention papers, 5 patents and 2 panel discussion positioning talks in international conferences.

This report consists of 3 chapters, The first chapter describes the developed automatic multigrain parallelism exploitation technology and performance evaluation of the individual technology. Chapter 2 includes the proposed performance evaluation techniques without individual function evaluations. Chapter 3 shows a brief report of the International Cooperation Committee with International Advisory Board, which was introduced for an international assessment, international cooperation, and lists of presented papers and patents.

Contents

I. Development of Advanced Parallelizing Compiler Technology

1. Development of Automatic Multi-Grain Parallelizing Compiler Technology

To accelerate programs on multi-processor systems, automatic parallelizing compilers need to exploit not only simple parallilism among loop-iteretion in a program but also complexed parallelism such as coarse-grain parallelism between subroutine-calls, between loops including subroutine calls or between loops and also fine-grain parallelism by sets of basic blocks,. To solve this problem, we are researching and developing the automatic multi-grain parallelizing technologies that makes the best use of multi-grain parallelism in programs and the tuning technologies for parallel processing that enables to enhance the compiler's parallelization of programs by feedbacking run-time information or user's knowledge to the compiler. In this fiscal year, we have developed elements of technologies to solve these problems and also we have started to evaluate each facility.

1.1. Multi-Grain Parallelism Exploitation Technology

A target of the research and development of multi-grain parallelism exploitation technology, which is the base technology of automatic multi-grain parallel processing, is to research and develop the technology for analysis of parallelism in a sequential program and efficient use of the parallelism on shared memory multiprocessor systems.

This year, as the second year of the project, technology for multi-grain parallelism exploitation considering cache optimization on commercial shared memory multiprocessor systems, loop parallelization technology for efficient exploitation of time and space locality, parallelism exploitation technology for sets of basic blocks and interprocedural multi-grain parallelism exploitation technology extending interprocedural dependency analysis technology were researched and developed.

1.1.1. Infrastructure Technology for Multi-Grain Parallelism Exploitation

This section reports multi-grain parallelism exploitation infrastructure technology considering cache optimization on shared memory multiprocessor systems in the multi-grain parallel processing which exploits coarse-grain parallelism among basic blocks, loops and subroutines effectively in addition to the loop parallelism.

Though shared memory multiprocessor architecture has been widely used, with increase of the number of processors the difference between peak performance and effective performance has been getting larger. To cope with this problem, it is important to use multi-grain parallelism using coarse-grain parallelism and near

fine-grain parallelism in addition to traditional loop parallelism. Moreover, the speed gap between processor and memory is getting larger with the advance of the processor technology. Therefore, the effective use of memory hierarchy, especially cache memory, is very important to enhance the performance of multiprocessor systems.

The proposed multi-grain parallel processing scheme considering cache optimization on SMP generates a parallelized program using OpenMP, which is a standard API for SMP, to reuse shared data on cache memory among macro-tasks from a sequential program. The effectiveness of the proposed scheme is evaluated on commercial SMPs like IBM RS6000 SP 604e High Node server and SUN Ultra80 workstation.

In the individual evaluation, OpenMP Fortran programs generated from partially modified 'tomcatv' and 'mgrid' from SPEC95FP benchmarks by using the compiler module which is under development are compiled by IBM XL Fortran version 6.1 and SUN Forte 6 Update 1 and executed on IBM RS6000 and SUN Ultra80. The evaluation gave us the prospect to attain performance improvement such as 2.5 times in 'tomcatv', 2.8 times in 'swim' and 5.8 times in 'mgrid' against the loop parallelizing compiler IBM XL Fortran version 6.1 on 8 processor SMP RS6000, and 2.5 times in 'tomcatv' and 3.6 times in 'swim' against minimum execution time by SUN Forte 6 Update 1 compiler on 4 processor SMP Ultra80.

1.1.2. Medium-Grain Parallelism Exploitation Technology

We have been researching and developing the techniques extracting medium-grain (loop) level parallelism.

In 2001 fiscal year, we have developed the techniques extracting not only DOALL-parallelism without any synchronization among processors, but also pipeline parallelism which each processor synchronizes with only near processors. We, at first, have developed the methods extracting pipeline parallelism efficiently. And we also have developed the methods to implement pipeline parallelism with 'FLUSH' directive in openMP.

Applying these techniques to 'applu' program of SPEC2000 benchmark suits, we may get the performance almost twice as that of the base compiler on the Alpha Server. Because we can extract both DOALL-parallelism and pipeline parallelism.

The results of this research has been published in SIG Notes of IPSJ.

1.1.3. Coarse-Grain Parallelism Exploitation Technology

It is important for automatic multigrain parallelizing compiler to exploit the coarse grain parallelism such as between subroutines, loops, and basic blocks to achieve good performance. We have been researching and developing the coarse grain parallelizing

mechanism, which can extract the coarse grain parallel tasks and generate the code not only for the speculative execution schema, but also for the non-speculative execution schema.

In 2001 fiscal year, we have developed the analysis routines, which recognize several points needed to find the coarse grain parallel task. That is, for each basic block, the use data definition point, the output data use point, the branch statement deciding the execution of it, and the branch statement deciding not to execute it are collected. These points and the length between them are used for deciding the initiation time and effect of parallel execution of each basic block. We have also been analyzing them for each loop nest. We consider that using these data, we can collect the fundamental data for extracting the coarse grain tasks.

1.1.4. Technique of Analyzing Interprocedural Multi-Grain Parallelism

Loop parallelization techniques cannot extract sufficient parallelism from the programs including such sections as those outside loops or sequential loops. So, the multi-grain parallelization technique, which can extract parallelism from multiple grains of sections (tasks) such as basic blocks or procedures in programs, is necessary. In this study we have been researching the technique of analyzing interprocedural multi-grain parallelism using the interprocedural automatic parallelizing compiler WPP (Whole Program Parallelizer) as a base compiler.

In this fiscal year, we have designed two techniques. One is the parallelism analysis technique that extracts parallelism from interprocedural hierarchical tasks. The other is the OpenMP program generation technique. The former technique uses a hierarchical control-flow graph (we call it a Control Flow Summary graph: CFS) made of a program by WPP. First, regarding each node of the CFS as a task, the technique analyzes control and data dependences between tasks. Second, for each layer of the CFS, it applies a static task scheduling method based on the CP/MISF only to the layer and estimates the execution time of the program with that scheduling. Third, it selects the layer with the scheduling that provides the shortest estimated execution time of the program and it applies a task parallelization to the layer. Fourth, it determines the location of each barrier in parallel regions. Last, the OpenMP program generation technique generates an OpenMP program with the above extracted parallelism. Using these techniques, much task parallelism is expected to be extracted from program sections that have no loop parallelism.

1.2. Data Dependency Analysis Technology

This technology is the basis of automatic parallelization. This fiscal year, we have

developed technologies for interprocedural data dependency analysis to enlarge the area for parallelization. And we have also developed and evaluated the predicated data-flow analysis and run-time data dependency analysis techniques that we started development last fiscal year.

1.2.1. Interprocedural Data Dependency Analysis Technology

The interprocedural dependency analysis is a key technology for an enhancement of an automatic parallelization.

To cancel parallel execution overheads and get an efficient effect of parallel execution, a compiler needs to recognize a parallelizable potion as coarsely as possible. But when a compiler recognizes a parallelizable potion, a border between procedures limits a scope of parallelizaion analysis.

A dependency analysis across the border between procedures make an analyzing scope wide. As a result this interprocedural dependency analysis make a coarse grain parallelizable portion.

In this research item we are studying the interprocedural dependency analysis technology for the medium grain parallelization function.

In this fiscal year, we have researched and developed following elemental functions.

- Output function for internal information of our compiler

  This function outputs internal information of our compiler to an external file.
- Input function for internal information of our compiler

  This function inputs an internal information of our compiler from an external file.
- Fusion function for internal information of some other procedures

  This function fills an internal interface of a caller with an internal interface of a callee.
- Analysis of procedure call tree

  This function analyzes procedure call tree.

These elemental functions are bases of the interprocedural dependency analysis.

We consider that we will be able to make up the interprocedural dependency analysis function for the medium grain parallelization function by using these elemental functions.

1.2.2. Predicated Data-Flow Analysis Technique

Our interprocedural parallelizing compiler WPP parallelizes loops by analyzing inter-procedural data dependences. There are some loops, which are not parallelized by WPP but can be parallelized principally. Such is the case where the data reference preventing parallelism executes only on a condition and the condition never holds when

using an input data. In this study, we have been researching and developing the predicated data-flow analysis in order to parallelize those loops.

In this fiscal year, we have been examining how to implement predicated data-flow analysis. Predicated data-flow analysis is implemented as follows. First, data-flow analysis phase analyzes for each statement an array reference region with a predicate, the condition the statement with the reference region executes. Second, data-dependence-analysis phase calculates for each loop such a condition that there is no loop-carried dependence using the array reference regions with predicates. Last, code-generation phase generates multi-versioned code, in which at runtime the condition is tested and selected is one of three loops; a serial loop and two parallel loops to which array privatization is applied and is not, respectively.

As a preliminary evaluation, we applied our technique by hand to such a loop in SPECfp95/apsi that WPP cannot parallelize. The parallel loop to which array privatization is applied is selected at runtime and its scalability is expected to achieve about six on eight processors.

1.2.3. Run-Time Data Dependency Analysis Technique

Most parallelizing compilers analyze loop-carried data dependences in a loop and judge the parallelizability of the loop. Those compilers, however, cannot parallelize the loop that includes an indirect-referenced array and has a possible loop-carried data dependence between two references of the array because the compilers can not judge the parallelizability of the loop at compile time. In this study, we have been researching and developing the run-time data-dependence analysis in order to parallelize the loop.

In this fiscal year we have implemented the run-time data-dependence analysis on our interprocedural parallelizing compiler WPP. The analysis finds target loops in an input program and outputs a code with the following workings. First, the code records the accesses to variables referenced in a target loop executing the loop in parallel. Second, the code checks the data dependences using the access record and re-executes the loop sequentially if the result of the check inhibits the parallelization of the loop. As a result, we can expect the good performance of a loop if the loop is judged to be parallelized at run time.

1.2.4. Parallelism Analysis Techniques for Fortran90

Recently, there are cases where we use Fortran90 as a programming language for the scientific application programs. The Fotran90 includes some new features in addition to the conventional FORTRAN77 specifications. So, this new language provides the parallelizing compilers with some challenges. In this study, aiming at

extracting more parallelism from Fortran90 programs we have been developing parallelism analysis techniques for Fortran90, expanding the interprocedural analysis techniques of the WPP.

In this fiscal year we have been designing a procedure cloning technique for the procedures including optional arguments or automatic arrays. This technique is realized in the following steps: the detection of optional arguments and automatic arrays, the elimination of optional arguments and calls to the PRESENT functions in the cloned procedures, and the change of automatic arrays to the fixed-size arrays. This technique is expected to extract more parallelism from Fortran90 programs than before.

1.3. Automatic Data Distribution Technology

The automatic data distribution technology is the compiler technology that partitions data and assigns each of them to the local memory of the most appropriate processor. There is a gap between the logical memory view and the physical memory structure on physically distributed shared-memory processors. So, different memory models need different optimization techniques. In this fiscal year, we have examined and designed the automatic data distribution technique for distributed shared-memory processors and developed the optimization technique of data locality for the processors with distributed shared memories or distributed shared caches.

1.3.1. Automatic Data Distribution Technology for Distributed Shared-Memory Multiprocessors

In recent years, the distributed shared-memory multiprocessors (DSMs) have attracted attention of users because of their performance scalability and their ease of parallel programming; the former is due to physically distributed memories and the latter logically shared memories. Although usual memory-referencing instructions for DSMs can access physical memories on remote processors as well as those on local processors, any reference to remote data takes more time than one to local data. For this reason, data distribution, which determines how to assign data to processors, is important to obtain good performance for DSMs. In this study, aiming at determining the most appropriate data distribution by compilers, we have been researching and developing automatic data distribution techniques for DSMs.

In this fiscal year, we have been designing and evaluating our data distribution technique for indirectly referenced arrays. Our automatic data distribution technique, the first-touch control data distribution method (FTC), realizes complex data distribution accurately using the first-touch page allocation mechanism of the

operating system. If a program includes an indirectly referenced array, our technique generates a code where a temporary array is used until the value of an index array of the array is determined and the indirectly referenced array is distributed by the FTC immediately after that.

As preliminary evaluation, we compared two versions of the NPB 2.3 serial/CG program. One is the program to which we apply our technique by hand. The other is the original program to which the OS applies the first-touch mechanism naturally. The former version is expected to run 5.9 times faster than the latter version on SGI(TM) Origin(TM) 2000 (32 processors).

1.3.2. Data Locality Optimization Technology

In multi-grain parallel processing on a multiprocessor system having distributed caches and distributed shared memories, in order to achieve high performance, it is required to develop automatic data distribution techniques which can reduce data transfer overhead among coarse-grain tasks by using distributed caches or distributed shared memories effectively.

This annual report presents a data-localization scheme to utilize coarse-grain task parallelism and data locality in multi-grain parallel processing. Concretely, so as to realize loop-aligned decomposition on large regions composed of loops in macrotask-graphs, a macrotask selection method for data-localization and an inter-loop dependence analysis method to resolve iteration-based data dependencies among loops inside data-localization target regions are proposed. This report also describes preliminary performance evaluation using manually generated codes on a multiprocessor system SGI Origin 2000. The evaluation shows that coarse-grain task parallel processing with data-localization can achieve 6.78 times speedup on 8 PEs in SPECfp95 'tomcatv' program compared with sequential processing.

1.4. Speculative Execution Technology

In this technological item, we research and develop the speculative execution scheme that is one of the element technologies of "Automatic Parallelizing Compiler". Our target of the speculative execution is not the branch prediction used in the conventional processor, that is, only instruction level speculation, but a medium grain size such as loop iteration level, and course grain size, such as between subroutines, loops, and basic blocks, is targeted.

In 2001 fiscal year, we continue to research and develop the algorithm of speculative execution for the medium grain tasks. Then, we have developed the algorithm. As for the speculation for the course grain size tasks, we began to research and develop

the effective speculation scheme by optimizing the task size and its initiation time. Moreover, we continue to develop the support mechanism to apply speculative execution effectively by collecting the dynamic information of a program.

1.4.1. Speculation Techniques for Medium-Grain Tasks for Multi-Grain Parallelization

The following four features are indispensable to adopt the speculative execution for medium grain tasks: (1) dividing a program into a set of tasks that are suitable for speculative execution, (2) selecting a task to be speculated, (3) dynamic scheduling to decide the initiation time of tasks, and (4) discarding the tasks that became unnecessary.

In 2001 fiscal year, we have developed the algorithm to adopt the above four mechanisms based on the research and development for (1) and (2) in 2000 fiscal year. Then, we have confirmed the usefulness of the scheme using the “compress program” from SPECcpu95 benchmark. Generally, since the execution time of loops holds the large portion of the total execution time, the conventional loop parallelization scheme improves the program performance, dramatically. However, when the data dependence cannot be analyzed statically, the conventional parallelization scheme assumes that the data dependence exists. For this reason, such a loop cannot be parallelized even if the loop carried dependence (LCD) occurs only once in 10,000 times, dynamically. However, the speculative execution scheme has been known to speedup such a loop.

In this technological item, we propose the scheme to apply the speculative execution alternatively only to the portion expected to be speeduped effectively, using the overhead parameter required for the book-keeping process when the speculation fails. Such overhead has not been considered on the conventional speculative execution schemes. The proposed scheme enables the alternative speculative execution using the overhead parameter for book-keeping, the LCD existence probability, and the timing of the speculative execution initiation. As a result, we have confirmed the usefulness of the algorithm through the implementation using “compress program”. The result of this research has been published in SIG Notes of IPSJ.

1.4.2. Speculative Execution Technology for Coarse-Grain Tasks

For course grain parallel execution, such as between subroutines, loops, and basic blocks, we have been researching and developing a frame-work which involves speculative execution and non-speculative execution, and optimizes the task size and its initiation time.

In 2001 fiscal year, we have been developing the speculative coarse grain task selection routine, in which each loop nest is checked how much earlier it can be executed speculatively from the branch statement deciding the execution of it. Based on conditions, such as the length between the use data definition point and the output data use point, and the length between the use data definition point and the branch statement deciding the execution, the routine decides whether the loop nest should be executed speculatively. We have been also developing the code generation routine for speculative execution. It generates the initiation code, the synchronization code, the code to store the output data in temporary area, the copying code from the temporary area to the original area when the speculation succeeds, and the cancellation code when the speculation fails, and other code for parallel thread generation.

1.5. Scheduling Technology

To efficiently execute programs in parallel on a multiprocessor system, a minimum-execution-time multiprocessor scheduling problem must be solved which determines the assignment of tasks to processors and the execution order of the tasks so that the execution time is minimum. It is known that the time complexity of this problem is strong NP-hard for a general problem which assumes arbitrary task processing time, arbitrary number of processors, arbitrary shapes of task graphs, and arbitrary inter-processor data communication time. Because of the intractability of the scheduling problem, it is necessary for parallelizing compilers to develop heuristic algorithms. Especially, it is important to develop scheduling algorithms which reduce data transfer overheads among tasks considering data transfer among tasks using distributed shared memory or cache memory on a multiprocessor system.

This section describes a coarse grain static task scheduling scheme DT-Gain/CP/MISF considering the cache optimization. DT-Gain/CP/MISF assigns a macrotask (MT) to a processor as follows. At a scheduling time instance, the scheduler calculates amounts of shared data among previously assigned tasks onto each processor and ready tasks (data transfer gain), and choose a combination of a ready task and processor which gives us the largest amount of shared data. After the calculation of the amount of shared data among MTs that are already assigned to processors and a ready MT, a combination of a processor and a MT that shares the most common data is chose. If there are several combinations of ready tasks and processors having the same amount of shared data, the combination including a task having the largest CP/MISF priority is selected.

The proposed scheduling algorithm is implemented on OSCAR Fortran multigrain parallelizing compiler modules and generates OpenMP Fortran after coarse grain task

scheduling considering cache optimization. The individual performance evaluation of the scheduling algorithm partly unified with the multigrain parallelism exploitation technology shows perspective we will be able to obtain 6.16 times speedup (4.56 times faster than Forte auto parallelization) for SPEC95 'swim' and 3.05 times speedup (2.37 times faster than Forte auto parallelization) for 'tomcatv' on Sun Ultra80 4 processor workstation though the sequential source codes of 'swim' and 'tomcatv' are partly modified.

Currently, we started development of an algorithm reducing memory access overheads using cache prefetch functions in addition to the above scheduling algorithm.

1.6. Extensions of a parallel programming language

This research item is necessary for the combination of some functions.

In this fiscal year, we have researched some specifications for the interface for parallelization ratio tuning and the interface for parallel execution efficiency tuning.

1.6.1. Interface for Parallelization Ratio Tuning

Compiler's automatic parallelization is widely used to obtain a good performance for programs on shared-memory multiprocessors. The performance, however, is limited because some programs need dynamic information for the parallelization judgement but most parallelizing compilers just use static information. Although there are some methods that judge the parallelizability of a program at run time, they cause a runtime overhead. So, the tuning technology for parallel processing and the directives for the tuning is important: the former uses a user's knowledge about a program and the latter makes the knowledge to be reflected in the program. In this study, aiming at developing some tuning directives that make possible to extract more parallelism from programs, we have been researching the extensions of a parallel programming language.

In this fiscal year, we have been examining directives that are the interface between our parallel tuning tool and our parallelizing compiler. We propose six directives or clauses. They are classified into three items. One provides the compiler with some hints such as the relation between variables. Another is concerned about a non-standard parallelization: the directive specifying the target loop and the variable name for the runtime parallelization. The other is concerned about the OpenMP: the clause specifying the name of an induction variable that is difficult to find for compilers. By inserting the above directives or clauses into a user's program, we can expect the speedup of the parallel performance of the program.

1.6.2. Interface for Parallel Execution Efficiency Tuning

As an interface for the tuning of parallel execution efficiency, we have examined some specifications from two angles. These are specifications that assist to get an efficient effect of parallel execution.

- Interface between Optimization Function
- Interface for Enhancement of Parallelization Function of OpenMP Fortran

We have examined five kinds of specification. As a result, we have decided that the prohibition of optimization is an interface between optimization functions and that the SECTION PRIVATE and processor binding are the interface for enhancement of parallelization function of OpenMP Fortran.

2. Development of Tuning Technology for Parallel Processing

Our goal in this research and development is to establish the interactive and platform-free parallelization tuning technology that speeds up the execution of a given program making the best use of dynamic information, which can not be obtained from compiler's static analyses. To achieve our goal we research and develop the following element techniques of the tuning technology for parallel processing: the program visualization technique (the technique summarizing, extracting, and visualizing the factors inhibiting parallelization), the technique for profiling and utilizing run-time information (the technique profiling run-time information and reflecting it to compiler's optimization), and the feedback-directed selection technique of compiler directives (the technique tuning the combination of compiler's optimizations). In this fiscal year we have been conducting the examination and development of each of those element techniques.

2.1. Program Visualization Technique

To obtain high performance on multiprocessors program parallelization is indispensable. So, compiler's automatic parallelization has been widely used. The automatic parallelization, however, is not enough for getting the maximal performance of a program. The parallelization tuning using user's knowledge is very important. To inspect the parallelism of a program efficiently it is important for tuning tools to provide users with helpful information such as compiler's analysis results. For example, there are some tools that show pairs of statements that have data-dependence relationship prohibiting parallelization. Showing those statements helps users to find causes of prohibiting parallelization. These tools, however, can not show any statement having data-dependence relationship in a procedure called within a loop. So, users have to find such statements for themselves; that is a laborious task for them. In this study, we are aiming at developing an effective parallelization-tuning tool for this case and are researching program-slicing technique that shows statements having data-dependence relationship beyond procedure boundaries.

In this fiscal year, we have been developing the following two tools. One is the interprocedural data-dependence-locating tool, which finds all the statements having data-dependence relationships in a loop including procedure calls even if those statements exist in a procedure called within the loop. The other is the interprocedural program slicing tool, which finds a program slicing beyond procedure boundaries. The former tool finds the statements with data dependences by comparing those array reference regions in the following way. First, the tool finds automatically all the

data-dependence relationships between assignment statements or calls to procedures in the same procedure as the target loop. Second, when the user specifies a call to a procedure, the tool finds on demand all the data-dependence relationships between any statement in the callee procedure and the statement that has the data dependence with the call to the procedure.

The latter tool finds the set of statements that affect the values of arrays involved in a given statement beyond procedure boundaries. As above two tools can support users to judge whether data dependences exist or not for loops including procedure calls, we can expect the parallelization tuning to be done efficiently.

2.2. Techniques for Profiling and Utilizing Run-Time Information

In the parallelization tuning, we first inspect the causes of poor performance for each part of a program. This inspection needs the hardware-counter information such as the number of data-cache misses. In the past, the collecting methods of these kinds of information or the kind of information that can be collected were different for each machine. So, it was difficult for users who use different kinds of machines to tune their machines. To ease such tuning, a platform-free library PAPI (Performance Application Programming Interface) that has the capability of collecting some hardware-counter information is proposed.

When the cause of poor performance for a loop is found by the above inspection, it sometimes happens that the cause is due to an inappropriate transformation by an optimizing compiler. That transformation is considered to be conducted based on indefinite information about the execution time of the loop or the loop trip counts, whose values are sometimes difficult to obtain at compile time. So, the technique for profiling, utilizing runtime information, and generating an optimized code has attracted attention of users. In this study, aiming at developing a platform-free interface that can collect hardware-counter information and loop-execution information, we have been researching the technique for profiling and utilizing runtime information.

In this fiscal year we have provided the precise definition for the PAPI specifications, defined output functions for the collected information, and implemented the PAPI library and the output functions on a Hitachi SR8000 parallel processor. We have also made a demand specification for the library collecting the loop-execution information and developed that library. In the former work, we clarify the PAPI specifications using the reference code that the PAPI project provides because there are some obscure points in the PAPI 1.1.5 specifications. We also define new output functions for the collected information because the PAPI library includes no such functions. In the latter

work, we include the following as the demand specification: the execution time for a loop, the loop trip counts, the stride of the loop, and the execution time per a iteration. We also determine the specification and the functions for eight output functions. Using the above results, it is expected to conduct the parallelization tuning using the platform-free interface that has the capability of collecting some hardware-counter information and loop-execution information.

2.3. Feedback-Directed Selection Technique of Compiler Directives

Optimizing compilers apply many kinds of loop transformation to a given loop nest. However, it is difficult for the compilers to select the optimized loop transformation or to determine the optimized loop expanding parameter. So, the option tuning, which determines the optimized compiler options or compiler directives based on runtime information, is important. There are two option-tuning tools for user programs. One optimizes compiler options, which are specified to the whole program. The other optimizes compiler directives, which can be specified to each loop. The former can not specify different options for different loops. The latter does not consider the optimized combination of directives that are effective to multiple loop nests as a whole. In this study, aiming at developing the option-tuning tool that is effective to multiple loop nests and finds the optimized combination of parallel and optimization directives in a short time, we have been researching the feedback-directed selection technique of compiler directives.

In this fiscal year we have conducted the examination of existing research, some case studies, and the design of our tool. This tool has the following two features. One is that it applies the same combination of parallel and optimization directives to each multiple loop nest in one trial and it measures the execution time of each multiple loop nest. The other is that it uses the fractional factorial design to determine the combination of directives for multiple loop nests. Using this tool, it is expected to determine the optimized combination of directives for multiple loop nests as a whole in a small number of combinations.

II. Development of performance evaluation for parallelizing compilers

The goal of this research is to establish the technology for more objective performance evaluation of a parallelizing compiler for SMP machines. We are developing this technology along with the evaluation of Research Theme I: "Development of Advanced Parallelizing Compiler Technology", and will use it in the next year's final evaluation of this project. Because our Parallelizing Compiler Technology developed by this project includes several optimization functions, the evaluations of each function are needed. We use the kernel-level programs and compact applications to evaluate these functions. We planed to use full-scaled application level benchmarks for the total evaluation. This year, we have investigated the parallelism of these benchmarks, built and checked the evaluation environments, selected the benchmarks, developed the guidelines for evaluating the performance of our technology.

1. Development of evaluation methods for individual functions

This year we started collecting the benchmarks for evaluating each optimization function. Also we investigated the parallelism of loop-level and coarse-grain level in the programs that are included in NPB, and had some knowledge from this investigation, which is described in 2.5. Other evaluations of each function are included in each function's description in Chapter I.

2. Development of an overall evaluation method

This year, we first built the environments for evaluating the performance of the benchmarks, checked whether the candidate benchmarks can run on these environments, then selected the benchmarks to use in the final evaluation, chose compile options, evaluated the performance of the selected benchmarks, and developed the guidelines for the final evaluations. We will describe our three evaluation environments in 2.3.

Our goal is to get the double performance on the same SMP machine compared with the objects generated by the commercial compilers that were released at the time this project began (Oct, 2000). Because of the nature of parallel execution, the best performance is not always obtained by using the maximum number of CPUs. So in this situation we will use CPUs by which we can obtain the best performance by these compilers.

2.1. Choice of benchmark programs

First, it will be necessary to use the well-known benchmarks for an overall evaluation. Also the benchmarks will be needed to have some scale to evaluate the parallel execution, while we can run these benchmarks on our environments. Of course some of these benchmarks can be parallelized by the current technology, which means it is impossible to achieve the double performance by our technology. Also some benchmarks may not have any parallelism that means there is no room of applying our technology at all.

Here we define these groups of attributes as follows.

- High level parallelism benchmarks:
  Even the commercial compilers can already achieve more than 50% scalability factor of the number of CPUs.

- Low level parallelism benchmarks:
  The scalability exists but not more than 50% of the number of CPUs by the commercial compilers.

- Difficult to parallelize benchmarks:
  Parallel execution time is the same level or even slower than the serial execution by the commercial compilers.

In this research we selected the benchmarks from SPECfp2000 and NPB, which are well-known in the scientific and engineering area, and tested whether these benchmarks can run on our environments each of which consists of the SMP machines and the commercial compilers.

SPECfp2000 benchmark suite is developed by SPEC/OSG, announced in 1999 as the successor of SPEC CFP95 benchmark suite. The performance result of this suite have been published by over 10 major benders and more than 230 systems including all the SMP machines of our environments. We selected 6 benchmarks from this suite written in FORTRAN77 as candidate of our evaluation benchmarks. FORTRAN77 is the only supported language developed by this project. The detailed analysis of these benchmarks is described in 2.4 and 2.6.

NPB benchmark suite is provided by NAS(Numerical Aerospace Simulation) program of NASA Ames Research Center, and targets the development of 21th century's aerospace vehicle using CFD (Computational Fluid Dynamics) computation. NPB simulate the computation and data transformation of the CFD programs and consists of 5 kernel benchmarks and 3 virtual application programs. All of these 8

benchmarks has 4 problem sizes to run, and these are identified by Class A, B, C and W. Because the IS benchmark is not floating point program, we select 7 benchmarks from these 8 benchmarks. And we tested Class A, B, C of these benchmarks for our evaluation. We omitted Class W because the dataset size will be too short for parallel execution. The result showed that the result of Class B and Class C are very much the same attribute from the viewpoint of parallelism except the case of CG. Class B and C cannot run on some of the machines because of the lack of memory, so we select Class A of 7 benchmarks and Class B CG as the candidates. Class B CG can run on all of our evaluation environments.

The scalability results of SPECfp2000 are below,

wupwise : 1.1times/20PE by SGI
swim : 4.5times/7PE by IBM
mgrid : 17.3times/32PE by SGI
applu : 17.3times/29PE by SGI
sixtrack : 1.0times/2PE by SGI
apsi : 1.0times/2PE by SGI

The scalability results of NPB are below,

EP Class A: 1.0times/2PE by SGI
MG Class A: 3.0times/8PE by COMPAQ
CG Class A: 31.2times/27PE by SGI
CG Class B: 3.0times/7PE by COMPAQ
FT Class A: 1.1times/4PE by SGI
LU Class A: 5.7times/31PE by SGI
SP Class A: 2.0times/5PE by IBM
BT Class A: 1.1times/1PE by SGI

From the result of these tests we can categorize the benchmarks as follow. We use the best scalability of our three environments in this categorization. In addition, some benchmarks are classified as high level parallelism benchmarks on some of the environments, but these are classified as low level parallelism benchmarks on other environments. For example, ‘applu’ is classified as high level on SGI, but classified as low level on COMPAQ and IBM.

- High level parallelism benchmarks
  swim, mgrid, applu, CG Class A

- Low level parallelism benchmarks
  MG Class A, CG Class B, LU Class A, SP Class A

- Difficult to parallelize benchmarks
  wupwise, sixtrack, apsi, EP Class A, FT Class A, BT Class A

The guideline of evaluation will be as follows.

For high level parallelism benchmarks, it will be impossible to achieve the double performance, but any of the performance improvement must be achieved.

For low level parallelism benchmarks, it will be the best candidate to achieve the double performance.

For difficult to parallelize benchmarks, if some of our technology can apply to these benchmarks, the performance improvement will be more than the double. In this case we can use any number of CPUs to improve the performance.

Currently, we have not yet proved some of these benchmarks has no parallelism, so we will use these 14 benchmarks to evaluate next year. In addition we can add some other well-known benchmarks such as SPECfp95 for the total evaluation based on these guidelines to enforce our evaluation.

2.2. Choice of compile options

To evaluate the benchmarks it is important to choose the performance compiler options. The compiler developed by this project also uses these commercial compilers as back-end, so the same option set will be used for the final evaluation for fairness of this evaluation. This means that these options set must also be robust enough.

So we started with the options published by each bender's SPECfp2000-base reports with automatic parallelization options of each compilers. The compiler options are as follows.

- SGI: -Ofast=ip27 -LNO:fusion=2 -apo
- COMPAQ: -v -arch ev6 -O5 -fkapargs='-ur=1' -fkapargs='-conc'
- IBM: -O3 -qarch=ppc –qhot -qsmp=auto

The detail results of each benchmark of each environment are described in 2.3.

Then we added some of the options that are similar to our developing functions. We tested 6 cases options on SGI environment, 6 cases options on COMPAQ, 3 cases on IBM.

But the maximum differences of every benchmark from the point of scalability factors are from 9% slower to 9% faster than the original compiler options. This means that there are only little impacts on the scalability factors of these benchmark performance. So we decided to use SPECfp2000_base compiler options not only for the evaluation of the commercial compilers but also the backend compiler options for our compiler.

2.3. Setting and confirming the evaluation environment, and preliminary evaluation results

Our evaluation environments are consist of SMP machines and a set of compilers as follows.

- SGI Origin2000
  R10000@195MHz, 32CPUs, 11GB Memory

  MIPSpro Fortran90 V7.30

- COMPAQ AlphaServer GS160 Model 6/731
  Alpha21264@731MHz, 8CPUs, 4GB Memory

  Compaq Fortran V5.4-1283-46ABA

  KAP Fortran V4.3

- IBM RS/6000
  PowerPC604@200MHz, 8 CPUs, 1GB Memory

  IBM XL Fortran 7.1.0

The scalability results of each environment and each benchmark are listed below. We use SPECfp2000_base compiler options of each compiler described in section 2.2 to get these results. As far as we use these compiler options, all benchmarks can run correctly, so we can use these environments as the total evaluation.

SPECfp2000 :

wupwise : 1.1times/20PE(SGI), 0.1times/1PE(COMPAQ), 0.6times/8 PE(IBM)
swim : 9.5times/31PE (SGI), 2.7times/8PE (COMPAQ), 4.5times/7 PE (IBM)
mgrid : 17.3times/32PE (SGI), 4.1times/8PE (COMPAQ), 4.0times/6 PE (IBM)
applu : 17.3times/29PE (SGI), 1.2times/4PE (COMPAQ), 1.6times/8 PE (IBM)
sixtrack : 1.0times/2PE (SGI), 0.9times/1PE (COMPAQ), 0.9times/2 PE (IBM)
apsi : 1.0 times/2PE (SGI), 0.2 times/1PE (COMPAQ), 0.8 times/2 PE (IBM)

NPB :

EP Class A : 1.0 times/2PE (SGI), 0.7 times/1PE (COMPAQ), 0.9 times/8 PE (IBM)
EP Class B : 1.0 times/2PE (SGI), 0.7 times/1PE (COMPAQ)
EP Class C : 1.0 times/2PE (SGI), 0.7 times/1PE (COMPAQ)
MG Class A : 1.0 times/1PE (SGI), 3.0 times/8PE (COMPAQ), 0.9 times/3 PE (IBM)
MG Class B : 1.0 times/2PE (SGI), 3.1 times/8PE (COMPAQ)
MG Class C : 1.0 times/2PE (SGI), 4.3 times/8PE (COMPAQ)

CG Class A : 31.2 times/27PE(SGI), 1.0 times/8PE(COMPAQ), 4.5 times/6 PE(IBM)

CG Class B : 4.4 times/31PE (SGI), 3.0 times/7PE (COMPAQ)

CG Class C : 5.9 times/32PE (SGI), 2.3 times/8PE (COMPAQ)

FT Calss A : 1.1 times/4PE (SGI), 1.0 times/1PE (COMPAQ), 1.0 times/2 PE (IBM)

FT Class B : 1.0 times/4 PE (COMPAQ)

LU Class A : 5.7 times/31 PE, 1.0 times/1PE (COMPAQ), 1.4 times/4 PE (IBM)

LU Class B : 4.8 times/31PE (SGI), 1.3 times/4PE (COMPAQ)

LU Class C : 7.4 times/20PE (SGI), 1.1 times/4PE (COMPAQ)

SP Class A : 1.0 times/1PE (SGI), 1.0 times/1PE (COMPAQ), 2.0 times/5 PE (IBM)

SP Class B : 1.0 times/1PE (SGI), 0.9 times/1PE (COMPAQ)

SP Class C : 1.0 times/1PE (SGI), 0.9 times/1PE (COMPAQ)

BT Class A : 1.1 times/1PE (SGI), 1.0 times/1PE (COMPAQ), 0.6 times/2 PE (IBM)

BT Class B : 1.0 times/1PE (SGI), 1.0 times/1PE (COMPAQ)

BT Class C : 1.0 times/1PE (SGI), 1.0 times/1PE (COMPAQ)

2.4. Analysis of SPEC benchmarks

As described in 2.1, the six benchmark programs (‘swim’, ‘mgrid’, ‘wupwise’, ‘apsi’, ‘applu’, ‘sixtrack’) written in FORTRAN77 have been selected from SPEC CFP2000 as the programs which are used for the performance evaluation. This section shows the properties of these programs by measuring the parallel processing times of these programs and the same programs in SPEC OMP2001.

2.4.1. Target benchmark programs

All of the six benchmark programs selected from CFP2000 are written in FORTRAN77.

The smallest program, ‘swim’, consists of 435 lines, 5 subroutines/functions and one file. The largest program, ‘sixtrack’, consists of 47,252 lines, 235 subroutines/functions and 123 files.

2.4.2. Measurement environment

Following list shows the multiprocessor systems which are used for this measurement.

a Sun Ultra80, 4CPU(450MHz, 4MB-L2, 1GB memory), SunOS5.8 7/01

b Sun Ultra80, 4CPU(450MHz, 4MB-L2, 2GB memory), SunOS5.8 7/01

c DELL Power Edge 6400, 4CPU(700MHz, 2MB-L2, 512MB memory),
Red Hat Linux 6.2 Kernel 2.2.14-5.0smp.

Each machine is a shared memory type multiprocessor. The size of main memory

differentiates a. and b.

Following list shows the parallelizing compilers and the parallelizing preprocessor which are used for this measurement.

- Sun Forte Developer 6 update 2, f95 parallelizing compiler
- PGI Workstation 3.2, pgf90 parallelizing compiler
- Visual KAP for OpenMP 3.9 parallelizing preprocessor

Sun and PGI compilers can generate a parallel object code by parallelizing a serial source program and can generate an object code from OpenMP source program.

Visual KAP for OpenMP is a preprocessor which parallelizes the source program and generates OpenMP code.

Four types of object codes listed below are executed on each machine.

- A serial object code which is generated by Forte or PGI compiler from the CFP2000 serial source program.
- A parallel object code which is parallelized and generated by Forte or PGI compiler from the CFP2000 serial source program.
- A parallel object code which is generated by Forte or PGI compiler from OpenMP code parallelized by Visual KAP from the CFP2000 serial source program.

A parallel object code which is generated by Forte or PGI compiler form OMP2001 parallelized source program.

2.4.3. Measurement results and property of each program

The highest parallel processing effect is obtained on Ultra80(2G memory). On Ultra80(1G memory), each execution time is almost the same as on Ultra80 (2G memory) except that each execution time is little bit longer than the time on Ultra80(2G memory). On Power Edge 6400, excepting ‘apsi’ and ‘applu’ each execution time shows the same parallel processing effect.

Based on these results, we can characterize each program as described below.

- swim, mgrid :

  Parallelized CFP2000 code compiled by each compiler can get the parallel processing effect nearly equivalent to that on OMP2001 program. It suggests that these programs can be fully parallelized by loop-level parallelizing compiler and are not appropriate for the evaluation of the multigrain parallelizing compiler.

- applu :

  Parallelized CFP2000 code compiled by each compiler can get parallel processing effect. It suggests that this program can be parallelized by loop-level parallelizing compiler and is not appropriate for the evaluation of the multigrain parallelizing compiler. It should be noted that we can not compare the execution

times of CFP2000 and OMP2001 directly, since CFP2000 and OMP2001 differ in the dimension of the main array variables and the way of subroutine calls of this program.

- apsi , wupwise :

  Parallelized CFP2000 code compiled by each compiler can not get parallel processing effect while parallel processing effect can be obtained on OMP2001 program. It suggests that that these programs can not be parallelized by conventional parallelizing compilers at all and can be appropriate for the evaluation of the multigrain parallelizing compiler.

  For example, the fact that the both of the loop 100 and 200 in the subroutine 'muldoe' (and 'muldeo'), which have enough parallelism, can not be parallelized by the conventional parallelizing compiler due to the subroutine calls in the loops implies that the subroutine muldoe and muldeo are suitable for the evaluation of the function to detect parallelism of loops which contain subroutine calls.

- sixtrack :

  Although OMP2001 does not contain this program and we can not compare the results, the fact that the parallel execution time becomes larger than the serial execution time shows that this program is the hardest one to parallelize. It suggests that that this program can not be parallelized by conventional parallelizing compilers at all and can be appropriate for the evaluation of the multigrain parallelizing compiler.

2.5. Analysis of programs in NAS benchmark suite

The goal of the advanced parallelizing compiler includes reducing execution time of a program by extracting multigrain parallelism of the source program. Although there are many works to investigate loop parallelism of benchmark programs, coarse grain parallelism of those have not been well investigated. This is the reason of difficulty to estimate performance of coarse grain parallelization and to select a benchmark program for evaluation of multigrain parallelizing compiler. In order to solve this problem, we investigated coarse grain parallelism of benchmark programs, which are included in the NAS Parallel Benchmarks, and estimated performance of multigrain parallelizing compiler for the NAS benchmark. Also, we discussed if the NAS benchmark is appropriate for performance evaluation of our compiler.

2.5.1. Analysis of parallelism for benchmark programs

There are several works to analyze parallelism of benchmark programs. Eigenmann et al. investigated loop level parallelism of the Perfect Benchmarks and estimated

performance of loop parallelization techniques for the benchmark. These results were used to develop their new parallelizing compiler, Polaris. However, the results contain only loop level parallelism and there are no results for coarse grain parallelism.

The NAS Parallel Benchmarks is developed as a benchmark program to evaluate performance of a parallel computer. The suite includes program codes parallelized by MPI or OpenMP. Thus, we can easily obtain information of loop level parallelism for each program code. However, coarse grain parallelism of a program is not investigated.

2.5.2. Method for analysis

Our analysis of coarse grain parallelism is similar to that performed in our advanced parallelizing compiler, that is, we decompose a source program into coarse grain tasks called macrotasks and analyze data/control dependencies among the macrotasks. Here, a macrotask is composed of a basic block, a loop, or a subroutine. The analyzed parallelism is represented by the macrotask graph. Then, we fuse small macrotasks, or a macrotask that has short execution time, into a larger macrotask in order to reduce relative scheduling overhead at runtime. For a macrotask that includes a subroutine call, we analyze coarse grain parallelism within the subroutine in hierarchical manner. However, we performed this hierarchical analysis only for a subroutine that has long execution time.

2.5.3. Results

We analyzed coarse grain parallelism for six serial programs that were included in the NAS Parallel Benchmarks 2.3-Serial, CG, MG, FT, LU, BT and SP. The analysis was performed manually, and we estimated performance of coarse grain parallelization for the programs.

The analysis for the kernel benchmark programs shows that CG, MG and FT, have no coarse grain parallelism that reduce execution time of the programs significantly. The reason is that the kernel benchmark program represents computation for a primary part of a real application program; thus, a number of macrotasks, or code size, is small and a single macrotask, a loop or a subroutine, dominates execution time of program. We conclude that the kernel benchmark programs have no coarse grain parallelism that significantly reduce execution time of the programs while they have loop level parallelism that may contribute to reduce the execution time.

Programs in the application benchmarks, LU, BT and SP, have similar characteristics, that is, they have similar macrotask graphs. While a number of macrotasks in the program is larger and structure of the graph is more complicated compared with those of the kernel benchmark program, there is no coarse grain

parallelism that reduces execution time of the program significantly. The reason is that a single particular subroutine spends about 90% of total execution time in the program. We conclude that the application benchmark programs also have no coarse grain parallelism that significantly reduces execution time of the programs.

2.5.4. Performance estimation for the loop aligned decomposition

The loop aligned decomposition is one of effective optimization methods in the multigrain parallelization scheme. In this method, a compiler decomposes multiple DOALL loops that access same array data into sub loops so that sequent sub loops access the same region of the array data in order to minimizes communication overhead. The method is mainly used for data localization on a multiprocessor system where each processing unit has local memory. However, it is also effective to improve advantage of a cache on a current SMP.

We analyzed the program code of SP and estimated performance of the loop aligned decomposition for the program. The analysis is performed by manually finding loops that may be effectively parallelized by the loop aligned decomposition and by performing preliminary evaluation on SUN Ultra80, where the program code manually applied the loop aligned decomposition is executed. The results shows that the loop aligned decomposition reduced execution time by more than 10% compared conventional loop parallelization for primary subroutine in SP. We conclude that SP can be an appropriate benchmark program for our performance evaluation of advanced parallelizing compiler. The other application benchmark programs, LU and BT are also appropriate for our performance evaluation, because they have similar program structures as SP.

2.6. Consideration of SPEC OMP2001 benchmark suite

The SPEC OMP2001 benchmark suite, which was written in OpenMP Application Program Interface, consists of 11 scientific technical computation application programs. It can be a set of highly tuned parallel programs written by users who know the applications very well and the result can be the goal of our automatic parallelization. We considered the result of SPEC OMP2001 benchmark and investigated what is important for efficient parallel execution.

2.6.1. Evaluation

Using the Fujitsu Parallelnavi Fortran V1.0.2 and C/C++ V1.0.2 compilers, we evaluated SPEC OMP2001 M-size benchmark suite on PRIMEPOWER2000, a SMP computation server of Fujitsu's. The result of each application program can be distinguished as follows:

- Excellent scaling
  wupwise, swim, applu, apsi, and gafort
- Good scaling up to 64 CPU's
  mgrid, equake, and art
- Poor scaling
  galgel, fma3d, and ammp

2.6.2. Analysis of the result

'Apsi', which provided finally high performance, was not scalable in the first trial. In 'apsi', large arrays are allocated frequently at the top of subroutines in the dynamic extent of the parallel region. Such allocation performed in parallel often causes partial serialization and lock conflict.

'Galgel' is one of the hardest benchmark in the suite to get high performance. It includes many PARALLEL DO blocks that enclose only a few assignment statements without nested-DO loops. This kind of PARALLEL DO block cannot be executed effectively because the cost of thread fork/join is relatively high compared to the parallel computation inside the block. Therefore, a naive implementation could cause even lower performance than the serial execution.

2.6.3. Contribution to automatic parallelization

All through the SPEC OMP2001 benchmarks, we met performance problems related to the memory allocation. In the Fortran90 implementation, array expressions, array assignments, and WHERE statements cause dynamic allocation generated by the compiler. For high performance, all of these allocations must be reduced and handled in parallel with the least number of conflicts between the threads. As mentioned in the example of 'apsi', avoiding memory allocation gives the best results.

As shown in 'galgel' and 'equake', the OpenMP program sometimes contains many PARALLEL DO/FOR directive blocks enclosing a small amount of computation. We would like to recommend a programming style in which many DO/FOR directive blocks are enclosed in a large PARALLEL directive block. Thus the number of thread fork/join can be reduced as much as possible.

III. Report of International Cooperation Committee, Lists of Accomplishment and Other Reference Materials

1. Report of International Cooperation Committee

The Advanced Parallelizing Compiler project has been established International Cooperation Committee containing International Advisory Board with world leading researchers as a new trial for self-assessments of project target, research & development accomplishment, international cooperation and dispatching project accomplishment to the world. Members of International Advisory Board are listed below:

Professor David A. Padua (University of Illinois at Urbana-Champaign)

Professor Monica S. Lam (Stanford University)

Professor Rudolf Eigenmann (Purdue University)

Professor Francois Irigoin (Ecole des Mines de Paris).

The first International Cooperation Committee was held in Waseda University in September 2001 as the attached agenda.

In the committee, purpose of the project, numerical target, R&D plan and R&D accomplishment for the first year were introduced to the International Advisory Board members.

With the related discussion and technical presentations by the board members, objective assessments for the project by the board members are given.

The followings are overview of the assessments.

- The numerical target that tries to double the performance of the parallelizing compiler compared with commercial loop parallelizing compilers available on the market in September 2001, or at the project starting time point, is very ambitious value considering that performance improvement by automatic parallelizing compilers for past ten years was about several ten percents. It would be important to focus on development of compilation technologies themselves and analysis of their performance rather than chasing the numerical value.
- Multigrain parallel processing, especially coarse grain task parallel processing, and affine partitioning in this project are interesting and will be world leading technologies if they are successfully completed.
- How unify results by competent competitive companies is a very difficult problem.
- Work is proceeding as scheduled and there are already impressive performance results for the coarse grain task parallel processing.
- Research on automatic parallelizing compilers is important and difficult. Long-

term research and development is desired.

The following is the agenda of the first International Cooperation Committee.

First Advanced Parallelizing Compiler Technology
International Cooperation Committee（Agenda）

1. Date and time:
   September 5, 2001　10:00～18:00
   September 6, 2001　10:00～17:45
2. Venue:
   School of Science and Engineering,Waseda University
   (September 5,10:00-13:00　September 6,10:00-14:45)
   Building No.55-N 2F　Conference Room
   (September 5,13:00-18:00　September 6,14:45-17:45)
   Building No.62 01-07 Conference Room
3. Meeting schedule

September 5 (Wednesday)

10:00～10:10	Greeting(Izumi, METI)
10:10～10:40	Summary of the APC Project (Professor Kasahara, Project Leader)
10:40～10:50	APC R&D Organization (Yamana)
10:50～11:20	The Compiler Technology of Advanced Parallelizing Compiler (Hotta)
11:20～11:40	The Performance Evaluation of Advanced Parallelinzing Compiler (Anzaki)
11:40～12:00	Deliberations on the project profile
13:00～13:30	Medium Grain Level Parallelization Technique(Fujitsu)
13:30～14:00	Automatic Data Distribution Method using First Touch Control for Distributed Shared Memory Multiprocessors(Hitachi)
14:30～14:45	Coffee Break
14:45～16:15	Past and Future Parallelizing Compilers (Prof. Padua)
16:30～18:00	Interprocedual Analyses and Compilers (Prof. Irigoin)

September 6 (Thursday)

10:00～11:30	Improving Parallelism and Locality using Affine Partitioning (Prof. Lam)
11:30～13:00	Lunch Time
13:00～14:30	Performance Evaluation of Parallelizing Compilers (Prof. Eigenmann)
14:45～16:15	Deliberations on the project in general　Comments on APC project targets, plan, and progress status, assessment system, coordination with other countries, final symposium
16:30～17:30	Comments from board members (entry of evaluation and comment sheet)
17:30～17:45	Comments from Advisory Board Chairman

2. Lists of Accomplishment and Other Reference Materials

2.1. Fujitsu Ltd.

（1）Eiji YAMANAKA, Hidetoshi IWASHITA, Kohichiro HOTTA: “Implementation of OpenMP on Parallel Process Environment”, 63rd National Convention of IPSJ Volume 1.

（2）Akira ASATO, Motoyuki KAWABA, Toshihiro Ozawa: "A multiprocessor system simulator for evaluation of parallelizing compiler", 63rd NationalConvention of IPSJ Volume 1.

（3）Hidetoshi IWASHITA, Eiji YAMANAKA, Kohichiro HOTTA: “A Study of OpenMP Programming and the Language Processor -- An Evaluation on Fujitsu PRIMEPOWER2000”, IPSJ SIG Notes, Vol.2002, No.9, pp.61-66 (2002.2)

（4）Akira Hosoi, Toshihiro Ozawa: “The Evaluation of the medium grain level Parallelization Technique”, IPSJ SIG Notes, Vol.2002, No.9, pp.49-54 (2002.2)

（5）Hidetoshi Iwashita, Eiji Yamanaka, Naoki Sueyasu, Matthijs van Waveren, Kenichi Miura, ``The SPEC OMP2001 Benchmark on the Fujitsu PRIMEPOWER System", Third European Workshop on OpenMP (EWOMP2001), Barcelona, Spain, September

2.2. Hitachi, Ltd.

（Predicated Data-Flow Analysis Technique）

・Anual Convention（1）

（1）Motoyasu Takabatake, ``Evaluation of Predicated Dataflow Analysis on Automatic Parallelizing Compiler", Proc. 64th Annual Convention IPSJ 5W-05, Mar., 2002.

（Automatic Data Distribution Technology for Distributed Shared-Memory Multiprocessors）

・Papers（1）

（2）Takashi Hirooka, Hiroshi Ohta, Takayoshi Iitsuka, ``Automatic Data Distribution Method using First Touch Control for Distributed Shared Memory Multiprocessors", Proc. of 14th International Workshop on Languages and Compilers for Parallel Computing (LCPC2001), Aug., 2001.

（Program Visualization Technique）

・Technical Reports（1）

（3）Makoto Satoh, Kiyomi Wada, ``Interprocedural Data-dependence Locating Method", IPSJ, 2002-ARC-146-8, Feb., 2002.

2.3. Waseda University, Kasahara laboratory

Infrastructure Technology for Multi-grain Parallelism Exploitation Journal（2）:

（1）Keiji Kimura, Takayuki Kato, Hironori Kasahara, ``Evaluation of Processor Core Architecture for Single Chip Multiprocessor with Near Fine Grain Parallel Processing'', Trans. of IPSJ, Vol. 42, No. 4, Apr., 2001.

（2）Hironori Kasahara, Motoki Obata, Kazuhisa Ishizaka, ``Coarse Grain Task Parallel Processing on a Shared Memory Multiprocessor System'', Trans. of IPSJ, Vol. 42, No. 4, Apr., 2001.

International Conference（3）:

（3）Motoki Obata, Kazuhisa Ishizaka, Hironori Kasahara, ``Automatic Coarse Grain Task Parallel Processing Using OSCAR Multigrain Parallelizing Compiler'', Ninth International Workshop on Compilers for Parallel Computers(CPC 2001), Edinburgh, Scotland UK, pp.173-182, Jun., 2001.

（4）Akimasa Yoshida, Satoshi Yagi, Hironori Kasahara, ``A Data Localization Scheme for Coarse Grain Task Parallel Processing on Shared Memory Multiprocessors'', Proc. of IEEE International Workshop on Advanced Compiler Technology for High Performance and Embedded Systems, pp.111-118, Jul.2001.

（5）Kazuhisa Ishizaka, Motoki Obata, Hironori Kasahara, ``Coarse Grain Task Parallel Processing with Cache Optimization on Shared Memory Multiprocessor'', Proc. of 14th International Workshop on Languages and Compilers for Parallel Computing (LCPC2001), Aug., 2001.

Technical Reports（6）:

（6）Takeshi Kodaka, Naohisa Miyashita, Keiji Kimura, Hironori Kasahara, ``Near Fine Grain Parallel Processing on Multimedia Application for Single Chip Multiprocessor'', Technical Report of IPSJ, ARC2001-140-11, Aug., 2001.

（7）Takayuki Uchida, Takechi Kodaka, Keiji Kimura, Hironori Kasahara, ``Multigrain Parallel Processing on Single Chip Multiprocessor'' Technical Report of IPSJ, ARC2002-146-3, Feb., 2002.

（8）Takeshi Kodaka, Takayuki Uchida, Keiji Kimura, Hironori Kasahara, ``Multigrain Parallel Processing for JPEG Encoding Program on an OSCAR type Single Chip Multiprocessor'' Technical Report of IPSJ, ARC2002-146-4, Feb., 2002

（9）Motoki Obata, Kazuhisa Ishizaka, Hiroki Kaminaga, Hirofumi Nakano, Akimasa Yoshida, Hironori Kasahara, ``Coarse Grain Task Parallel Processing on Commercial SMPs'', Technical Report of IPSJ, ARC2002-146-10, Feb., 2002.

（10）Shin-ya Kumazawa, Kazuhisa Ishizaka, Motoki Obata, Hironori Kasahara, ``An Analysis-time Procedure Inlining and Flexible Cloning Scheme for Coarse-grain

Automatic Parallelizing Compilation", Technical Report of IPSJ, ARC, Mar., 2002.

（11）Satoshi Yagi, Hiroki Itagaki, Hirofumi Nakano, Kazuhisa Ishizaka, Motoki Obata, Akimasa Yoshida, Hironori Kasahara, ``A Macrotask selection technique for Data-Localization Scheme on Shared-memory Multi-Processor", Technical Report of IPSJ, ARC, Mar., 2002.

Scheduling Technology

Journal（1）:

（12）Takao Tobita, Hironori Kasahara, ``A standard task graph set for fair evaluation of multiprocessor scheduling algorithms", Journal of scheduing, John Wiley & Sons Ltd, 2002.

International Conference（1）:

（13）Hirofumi Nakano, Kazuhisa Ishizaka, Motoki Obata, Keiji Kimura, Hironori Kasahara, ``Static Coarse Grain Task Scheduling with Cache Optimization Using OpenMP", WOMPEI, 2002.

Technical Reports（1）:

（14）Hirofumi Nakano, Kazuhisa Ishizaka, Motoki Obata, Keiji Kimura, Hironori Kasahara, ``A Static Scheduling Scheme for Coarse Grain Tasks considering Cache Optimization on SMP", Technical Report of IPSJ, ARC2001-140-12, Aug., 2001.

2.4. Waseda University, YAMANA Laboratory.

（1）Fumiko SAITO, Hayato YAMANA: “The Latest Technical Trends in Speculative Execution”, IPSJ SIG Notes, Vol.2001, No.116, pp.67-72 (2001.11)

（2）Shunsuke ISHIKAWA, Hayato YAMANA：“An Efficient Speculative Execution Scheme for Loops”, IPSJ SIG Notes, Vol.2002, 2001-HPC-89 (to appear) (2002.03)

2.5. Toho University, Yoshida Laboratory.

・International Conference

（1）A. Yoshida, S. Yagi, H. Kasahara: "A Data Localization Scheme for Coarse Grain Task Parallel Processing on Shared Memory Multiprocessors", Proc. of IEEE International Workshop on Advanced Compiler Technology for High Performance and Embedded Systems, pp. 111-118, Jul. 2001.

・Symposium with Review

（2） A. Yoshida: "Dynamic Scheduling Scheme with Overlapping Assignment for Coarse-Grain Task Parallel Processing", Joint Symposium on Parallel Processing JSPP2001, pp.351-358, Jun. 2001.

・Technical Report

（3） S. Yagi, H. Itagaki, H. Nakano, K. Ishizaka, M. Obata, A. Yoshida, H. Kasahara:

"A Macrotask Selection Technique for Data-Localization Scheme on Shared-Memory Multi-Processor", SIG Notes of IPSJ, 2002-ARC-147-34, Mar. 2002.

・Annual Convention

（4） T. Aramaki, A. Yoshida, "Multi-Level Task Scheduling for Coarse Grain Task Parallel Processing ", Proc. 63rd Annual Convention IPSJ, 2L-5, Sep. 2001.

2.6. Tokyo Institute of Technology

（1）Yoshiaki Ishii, Kento Aida, ``Analysis for Coarse Grain Parallelism of NAS Parallel Benchmarks," IPSJ SIG Notes, ARC (2002.03)

2.7. The University of Electro-Communications

（1）Hiroki Honda," Parallelisms in Programs Used for Evaluation of Multi-Grain Parallel Processing”, IEICE 2002 General Conference, D-6-4, Mar., 2002.

アドバンスト並列化コンパイラ技術

発　行　平成１４年３月
発行所　財団法人日本情報処理開発協会
〒105‐0011　東京都港区芝公園３‐５‐８
機会振興会館内
電　話　03‐3432‐9390
ＦＡＸ　03‐3431‐4324