Static Coarse Grain Task Scheduling with Cache Optimization Using OpenMP

Hirofumi Nakano†, Kazuhisa Ishizaka‡, Motoki Obata‡,
Keiji Kimura‡, Hironori Kasahara‡
{hnakano,ishizaka,obata,kimura,kasahara}@oscar.elec.waseda.ac.jp

†Waseda University,
3-4-1 Ohkubo, Shinjuku-ku, Tokyo, 169-8555, Japan
‡Waseda University & Advanced Parallelizing Compiler Project

Abstract. Effective use of cache memory is getting more important with increasing gap between the processor speed and memory access speed. Also, use of multigrain parallelism is getting more important to improve effective performance beyond the limitation of loop iteration level parallelism. Considering these factors, this paper proposes a coarse grain task static scheduling scheme considering cache optimization. The proposed scheme schedules coarse grain tasks to threads so that shared data among coarse grain tasks can be passed via cache after task and data decomposition considering cache size at compile time. It is implemented on OSCAR Fortran multigrain parallelizing compiler and evaluated on Sun Ultra80 four-processor SMP workstation, using Swim and Tomcatv from the SPEC fp 95. As the results, the proposed scheme gives us 4.56 times speedup for Swim and 2.37 times on 4 processors for Tomcatv respectively against the Sun Forte HPC 6 loop parallelizing compiler.

1 Introduction

With increasing gap between processor and memory access speeds, locality optimization for cache is getting more important to improve effective performance of multiprocessor system.

Also, it is getting difficult to improve performance of multiprocessor system dramatically using traditional loop parallel processing with maturity of loop parallelization techniques. To overcome the difficulty and to get scalable performance improvement, exploitation of multigrain parallelism, which hierarchically uses coarse grain parallelism among loops and subroutines, loop parallelism among loop iterations and (near) fine grain parallelism among instructions or statements [1–3], is needed.

As to cache optimization by compilers, there has been made various studies, such as, affine partitioning [4–6] which unifies multiple loop restructures, a vertical execution of tasks after loop decomposition [7], a cache optimization among coarse grain tasks for a single processor [8] and for shared memory multiprocessors with dynamic task scheduling [9].

This paper describes a static coarse grain task scheduling with cache optimization [2, 3, 9] based on data localization method [10, 11]. This scheme is implemented on OSCAR multigrain parallelizing compiler [12]. OSCAR compiler generates OpenMP Fortran whose coarse grain tasks are statically scheduled to parallel threads with cache optimization from a sequential Fortran program.

In Section 2, coarse grain task parallel processing is described. Section 3 proposes a static coarse grain task scheduling with cache optimization using OpenMP. Also, the effectiveness of the proposed schemes is evaluated on Sun Ultra80 four-processor SMP workstation using Swim and Tomcatv from the SPEC fp 95 benchmark suite in Section 4. Finally, concluding remarks are described in Section 5.

2 Coarse Grain Task Parallel Processing

This section describes coarse grain task parallel processing, which is a part of multigrain parallel processing. Coarse grain task parallel processing uses parallelism among three kinds of macro-tasks, or coarse grain tasks, namely, block of pseudo assignment statements (BPA), repetition block (RB) and subroutine block (SB). The compiler decomposes a source program into macro-tasks. Also, it generates macro-tasks hierarchically inside a sequential repetition block and a subroutine block.

Coarse grain task parallelization by OSCAR compiler is performed in the following steps.

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

In the following, the details of the above steps are described

2.1 Generation of Macro-Tasks [13]

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

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

2.2 Generation of Macro Flow Graph

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

In the Fig. 1(a), nodes represent macro-tasks, solid edges represent data dependencies among macro-tasks and dotted edges represent control flow. A small circle inside a node represents a conditional branch inside a macro-task. Though arrows of edges are omitted in the macro flow graph, it is assumed that the directions are downward.

2.3 Generation of Macro Task Graph

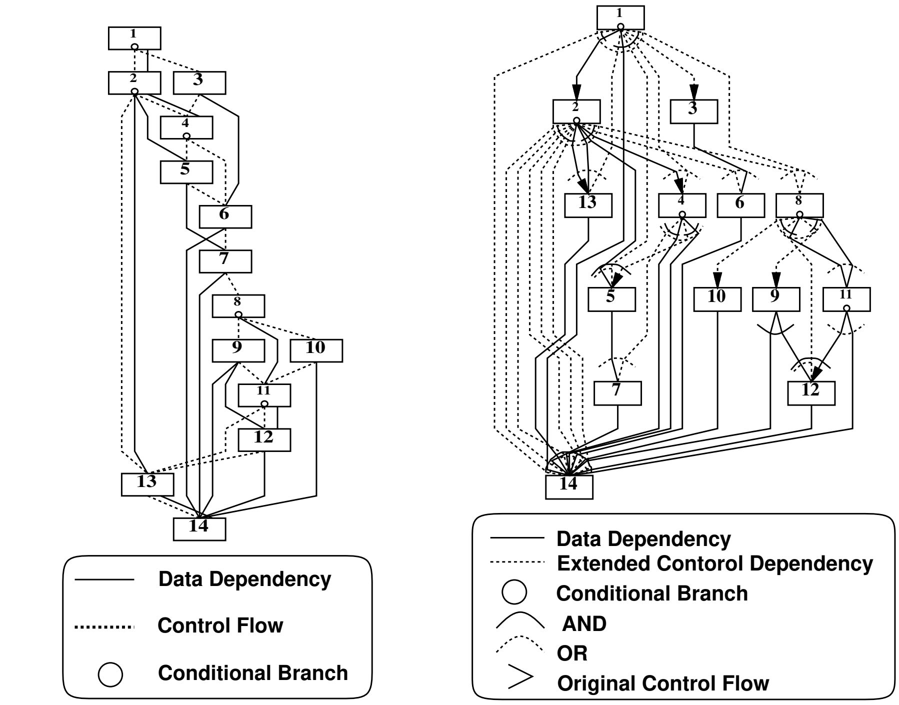
To extract parallelism among macro-tasks from macro flow graph, compiler analyses Earliest Executable Condition of each macro-task. Earliest Executable Condition represents the conditions on which macro-task may begin its execution earliest.

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

In the MTG, nodes represent macro-tasks. A small circle inside nodes represents conditional branches. Solid edges represent data dependencies. Dotted edges represent extended control dependencies. Extended control dependency means ordinary normal control dependency and the condition on which a data dependence predecessor macro-task is not executed. Solid and dotted arcs connecting solid and dotted edges have two different meanings. A solid arc represents that edges connected by the arc are in AND relationship. A dotted arc represents that edges connected by the arc are in OR relationship. In macro task graph, though arrows of edges are omitted assuming downward, an edge having arrow represents original control flow edges, or branch direction in macro flow graph.

2.4 Macro-Task Scheduling

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



(a) Macro Flow Graph (MFG) **(b) Macro Task Graph (MTG)**

Fig. 1. Macro Flow Graph and Macro Task Graph

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

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

3 Static Coarse Grain Task Scheduling with Cache Optimization

In this section, the static scheduling algorithm considering cache optimization for coarse grain task parallel processing is described. In this paper, macro-tasks are assigned to processors, or threads, because the SMP machine used for this performance evaluation has only four processors.

The overview of the proposed algorithm is shown below. When a macro-task MT_i is executed on a processor, the data defined or referred in MT_i is likely to be on cache of the processor when the macro-task finishes. If a succeeding macro-task MT_j that shares a lot of data with MT_i is assigned to the same processor immediately after MT_i, a large portion of the shared data can be transferred from MT_i to MT_j through a cache.

3.1 Macro-Task Decomposition

In the case where the amount of data defined or referred in a macro-task MT_i is much larger than cache size, even if a macro-task MT_j which shares a large amount of data with MT_i is assigned immediately after MT_i, a large part of shared data would be already replaced, and couldn't be transferred through a cache to MT_j. Therefore, in this case macro-tasks and data should be decomposed into smaller macro-tasks with data fitting to cache.

To reduce data transfer overhead among macro-tasks assigned to different processors and to transfer the shared data through a cache between macro-tasks assigned to the same processor, a loop aligned decomposition [10, 11] considering both amount of shared data and parallelism among macro-tasks is useful.

The loop aligned decomposition can be applied to arbitrary macro-task graph, in which RBs like doall loops and reduction loops are connected by data dependence edges.

3.2 Static Scheduling Algorithm DT-Gain/CP/MISF_DLG

This section proposes DT-gain/CP/MISF_DLG algorithm (Data Transfer Gain/ Critical Path/Most Immediate Successors First considering Data Localization Group). This algorithm schedules macro-tasks to processors so that tasks inside a Data Localization Group (DLG), which is a group of tasks generated by Loop Aligned Decomposition sharing the large data, are assigned to the same processor and a task outside DLG or an entrance task of a DLG are assigned to a processor having the largest Data Gain, or the most shared data to be accessed by the task. If there are multiple combinations of a ready task and a processor having the same Data Gain, a combination with a task having the largest path length to the exit node on the MTG (Macro Task Graph) is chosen. Furthermore, if there are multiple such combinations, a combination with a task having the largest number of immediate successors is chosen.

The details of the algorithm are follows.

Step 1 : Calculate the largest path length, or CP, from each task node to the exit node on a target Macro Task Graph (MTG).

Step 2 : Find ready tasks of which the all preceding tasks finish their execution or preceding task does not exist.

Step 3 : If there is a ready task belonging to a Data Localization Group of which one or more preceding tasks inside the same DLG are already assigned to a processor, assign the ready task to the same processor as the preceding tasks.

Repeat Step 3 until such ready tasks does not exist

Step 4 : Calculate Data Gain for every combination of ready tasks out side DLG or a ready task that is an entrance node of a DLG and idle processors. Here, Data Gain is an amount of shared data existing on each processor to be accessed by the ready task and means the data transfer amount to be reduced if the ready task is assigned to the processor.

Assign a ready task to a processor in the combination with the largest Data Gain. If there are combinations with the same largest data gain, choose a combination including a ready task having the largest CP as the above combination. Furthermore, if there are combinations having the largest CP, choose a combination including a ready task having the most immediate succeeding tasks.

If there exist tasks that have not been assigned, go to Step 2. Otherwise finish scheduling.

4 Performance Evaluation

This section describes performance evaluation. Performance is evaluated on Sun Ultra80 four-processor SMP workstation using Swim and Tomcatv from the SPEC fp 95.

4.1 OSCAR Fortran Multigrain Parallelizing Compiler

OSCAR Fortran multigrain parallelizing compiler, on which the proposed scheduling scheme, consists of a front end, a middle path and multiple back ends is as shown in Fig. 2.

A front end reads in Fortran77 and OpenMP and generates intermediate language.

A middle path analyses control flow and data dependence, restructures program, generates macro-tasks and exploits parallelism. It statically schedules coarse grain tasks considering cache optimization and generates parallelized intermediate code.

In OSCAR Fortran compiler, variety of back ends as shown in Fig. 2 are provided for different target machines like OSCAR multiprocessor system [13], UltraSparc processors, Fujitsu VPP vector supercomputers, heterogeneous cluster computing system with STAMPI and SMP machines with OpenMP. They generate assembly codes or parallelized Fortran codes with library calls or directives for each target machines.

The proposed static scheduling scheme for coarse grain tasks considering cache optimization is implemented on the middle path. Our coarse grain task parallel processing using OpenMP [3, 9] uses "one time single level thread generation method" which forks and joins threads only once at the beginning and the end of execution respectively to reduce thread generation overhead. However, this method realizes hierarchical coarse grain task parallel processing using OpenMP, regardless single level thread generation, by generating different codes for each thread [9]. The generated OpenMP Fortran is compiled by a native compiler for target machine and executed. In other words, OSCAR Fortran multigrain parallelizing compiler is used as a preprocessor that translates Fortran77 into parallelized OpenMP Fortran.

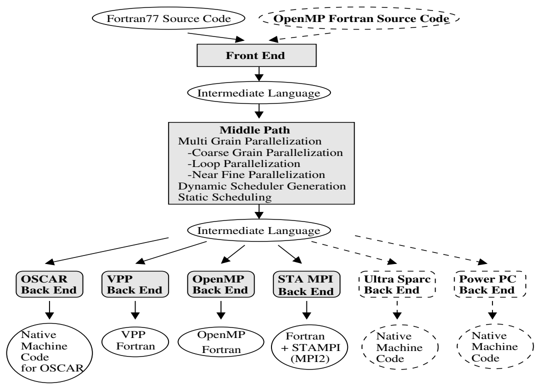


Fig. 2. A Composition of OSCAR Fortran Multigrain Parallelizing Compiler

4.2 Evaluation Environment

This subsection describes multiprocessor workstation Sun Ultra80, its compiler and benchmark programs used for the evaluation.

The specification of Sun Ultra80 four-processor SMP workstation and Forte loop automatic parallelizing compiler are shown in Table 1. Also, the used compile option for Forte compiler is shown in Table 2. In Table 2, Forte means compile options used in Forte compiler for a single processor and for automatic parallelization. Also OSCAR means compile options when OpenMP codes generated by OSCAR compiler are compiled by Forte compiler.

As application programs for the evaluation, Swim and Tomcatv from the SPEC fp 95 benchmark suite are used. Ref data set is used as input data.

In this evaluation, some parts of program sources are restructured by hand because the current version of OSCAR compiler has concentrated on developing original schemes and has not implemented some traditional program restructuring techniques. For example, in Swim, three subroutines, CALC1, CALC2 and CALC3, consume large execution time. These subroutines share a large amount

Table 1. The Specification of Sun Ultra80

Vender	Sun Microsystems
CPU	450MHz UltraSPARC-II 4 processors SMP
L1 Instruction Cache	16Kbyte Pseudo 2-Way Set Associative Line size: 32byte
L1 Data Cache	16Kbyte, Direct-Map Line size: 32byte (Two 16byte sub-blocks) Write-through Non-allocating
L2 Unified Cache	4Mbyte, Direct-Map Line size: 64byte Write-back, Allocating
Main Memory	1024Mbyte
OS	Solaris8
Compiler	Forte[tm] HPC 6 update 1

Table 2. Compile Option

<table>
<tr><th></th><th>Single processor</th><th>Multiprocessor</th></tr>
<tr><td>Forte</td><td rowspan="2">-fast</td><td>-fast -parallel
-reduction
-stackvar</td></tr>
<tr><td>OSCAR</td><td>-fast
-explicitpar
-mp=openmp</td></tr>
</table>

of data. For these three subroutines, inline expansion and flexible cloning [14] are manually applied. Also, in Tomcatv, an array size is input from an input file at runtime. However, in this paper, array size of ref data set is described as a parameter to use static scheduling. Also, loop interchange and array subscript exchange are manually applied for Tomcatv.

Performance evaluation results for the restructured Swim and Tomcatv are shown in Fig. 3, 4, respectively. These figures show speedup against sequential execution time by Forte compiler for a single processor. Also, each execution time was measured five times and the fastest time was as the plotted result.

In Fig. 3, the sequential execution time of Swim compiled by Forte was 99.7 seconds. The speedups (execution times) of automatic loop parallelization by Forte compiler were 1.51 times (66.1 seconds) for 2PEs, 1.60 times (62.5 seconds) for 3PEs and 1.66 time (60.2 seconds) for 4PEs. When OSCAR compiler was used as a preprocessor of Forte compiler, the speedups (execution times) were

1.23 times (81.3 seconds) for 1PE, 2.08 times (47.9 seconds) for 2PEs, 3.59 times (27.8 seconds) for 3PEs and 7.55 times (13.2 seconds) for 4PEs. The speedup by the proposed scheme was super linear by successful cache optimization.

In Fig. 4, the sequential execution time of Tomcatv compiled by Forte was 107.8 seconds. The speedups (execution times) of automatic loop parallelization by Forte compiler were 1.28 times (84.3 seconds) for 2PEs, 1.36 times (79.3 seconds) for 3PEs and 1.37 time (78.6 seconds) for 4PEs. When OSCAR compiler was used as a preprocessor of Forte compiler, the speedups (execution times) were 1.07 times (101.1 seconds) for 1PE, 1.68 times (64.0 seconds) for 2PEs, 2.12 times (50.9 seconds) for 3PEs and 3.26 times (33.1 seconds) for 4PEs.

At these evaluations, L2 cache misses in Swim compiled by Forte compiler were about 3.4×10^8 for 1PE, 3.2×10^8 for 2PEs, 3.1×10^8 for 3PEs and 3.0×10^8 for 4PEs. On the contrary, L2 cache misses of OSCAR compiler were about 2.0×10^8 for 1PE, 1.8×10^8 for 2PEs, 9.3×10^7 for 3PEs and 4.5×10^7 for 4PEs. Also, though L2 cache misses in Tomcatv compiled by Forte compiler were about 2.7×10^8 for 1PE, 3.8×10^8 for 2PEs, 3.9×10^8 for 3PEs and 4.0×10^8 for 4PEs, L2 cache misses of OSCAR compiler were about 2.0×10^8 for 1PE, 2.4×10^8 for 2PEs, 1.3×10^8 for 3PEs and 9.3×10^7 for 4PES.

These results show the proposed static coarse grain task scheduling scheme realizes 4.56 times and 2.37 times speedup against Forte compiler for Swim and Tomcatv by the improvement of L2 cache hit.

5 Conclusions

This paper has proposed a static coarse grain task scheduling with cache optimization using OpenMP. The proposed scheme is implemented on OSCAR Fortran multigrain parallelizing compiler, sequential Fortran is input and OpenMP Fortran whose coarse grain tasks are statically scheduled with cache optimization is output.

Its performance is evaluated on Sun Ultra80 four-processor SMP workstation, using Swim and Tomcatv from the SPEC fp 95. The results of evaluation show us that speedups of Swim and Tomcatv for 4 processors were 7.55 times and 3.26 times respectively against sequential execution time of them compiled by Forte.

A part of this research has been supported by METI/NEDO Millennium Project IT21 “Advanced Parallelizing Compiler” and STARC “Compiler cooperative single chip multiprocessor” project.

References

1. APC. http://www.apc.waseda.ac.jp/.
2. M. Okamoto, K. Aida, M. Miyazawa, H. Honda, and H. Kasahara. A hierarchical macro-dataflow computation scheme for oscar multi-grain compiler. *Trans. IPSJ*, 35(4):513–521, 1994.
3. H. Kasahara, M. Obata, and K. Ishizaka. Automatic coarse grain task parallel processing on smp using openmp. In *Proc. 12th Workshop on Languages and Compilers for Parallel Computing*, Aug 2000.

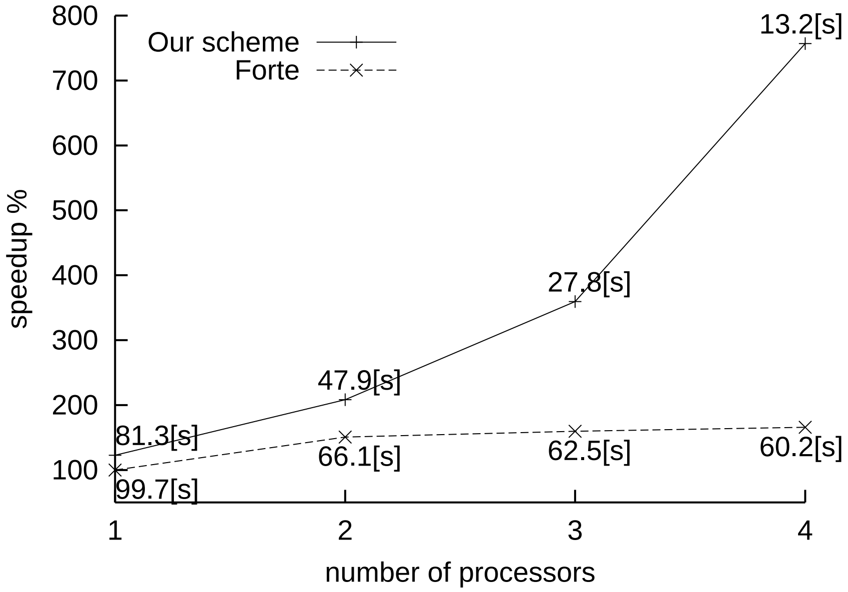


Fig. 3. Speedup of Swim

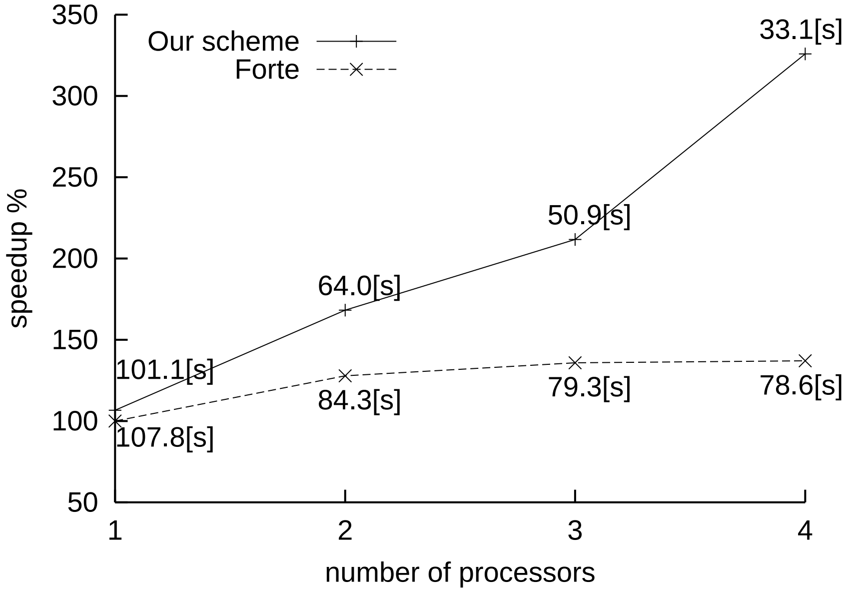


Fig. 4. Speedup of Tomcatv

4. A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proc. 13th ACM SIGARCH International Conference on Supercomputing*, Jun 1999.
5. A. W. Lim, S. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proc. of the Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Jun 2001.
6. A. W. Lim and M. S. Lam. Cache optimizations with affine partitioning. In *Proc. of the Tenth SIAM Conference on Parallel Processing for Scientific Computing*, Mar 2001.
7. S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malony, S. Shende, R. Oldehoeft, and S. Smith. Smarts: exploiting temporal locality and parallelism through vertical execution. In *Proc. of the 1999 international conference on Supercomputing*, Jun 1999.
8. D. Inaishi, K. Kimura, K. Fujimoto, W. Ogata, M. Okamoto, and H. Kasahara. A cache optimization with earliest executable condition analysis. In *Technical report of IPSJ*, Aug 1998.
9. K. Ishizaka, M. Obata, and H. Kasahara. Coarse grain task parallel processing with cache optimization on shared memory multiprocessor. In *Proc. 14th Workshop on Languages and Compilers for Parallel Computing*, Aug 2001.
10. 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. IPSJ*, 40(5):2054–2063, 1999.
11. A. Yoshida, S. Yagi, and H. Kasahara. A data-localization scheme for macrotask-graph with data dependencies on smp. In *Technical report of IPSJ, 2001-ARC-141*, Jan 2001.
12. H. Kasahara. *Parallel Processing Technology.* CORONA PUBLISHING CO., LTD., 1991.
13. H. Kasahara, H. Honda, A. Mogi, A. Ogura, K. Fujiwara, and S. Narita. A multi-grain parallelizing compilation scheme for oscar. *Proc. 4th Workshop on Languages and Compilers for Parallel Computing*, Aug 1991.
14. K. Yoshii, G. Matsui, M. Obata, S. Kumazawa, and H. Kasahara. An analysis-time procedure inlining scheme for multi-grain automatic parallelizing compilation. In *Technical report of IPSJ, ARC/HPC*, Mar 2000.