Hierarchical Parallelism Control for Multigrain Parallel Processing

Motoki Obata[†,††], Jun shirako[†], Hiroki Kaminaga[†], Kazuhisa Ishizaka[†,††], Hironori Kasahara[†,††]
Waseda University[†] Advanced Parallelizing Compiler Project[††]
3-4-1 Ohkubo, Shinjuku-ku, Tokyo, 169-8555, Japan
{obata,shirako,kaminaga,ishizaka,kasahara}@oscar.elec.waseda.ac.jp

Abstract

To improve effective performance and usability of shared memory multiprocessor systems, a multi-grain compilation scheme, which hierarchically exploits coarse grain parallelism among loops, subroutines and basic blocks, conventional loop parallelism and near fine grain parallelism among statements inside a basic block, is important. In order to efficiently use hierarchical parallelism of each nest level, or layer, in multigrain parallel processing, it is required to determine how many processors or groups of processors should be assigned to each layer, according to the parallelism of the layer. This paper proposes an automatic hierarchical parallelism control scheme to assign suitable number of processors to each layer so that the parallelism of each hierarchy can be used efficiently. Performance of the proposed scheme is evaluated on IBM RS6000 SMP server with 8 processors using 8 programs of SPEC95FP.

1. Introduction

As a parallel processing scheme on multiprocessor systems, loop level parallelism has been widely used by automatic parallelizing compilers[1, 2]. As examples of loop parallelizing research compilers, Polaris compiler[3, 4, 5] exploits loop parallelism by using inline expansion of subroutine, symbolic propagation, array privatization[4, 6] and run-time data dependence analysis[5] and SUIF compiler[7, 8, 9] parallelizes loops with inter-procedure analysis, unimodular transformation and data locality optimization [10, 11]. Effective optimization of data locality is more and more important because of the increasing gap between memory and processor speeds. Currently, many researches for data locality optimization using program restructuring techniques such as blocking, tiling, padding and data localization, has been proceeded for high performance computing and single chip multiprocessor systems [10, 12, 13, 14].

However, by those research efforts, the loop parallelization techniques are reaching maturity. In light of this fact, new generation parallelization techniques like multigrain parallelization are desired to overcome the limitation of the loop parallelization.

OSCAR FORTRAN compiler realizes multigrain parallelization [15, 16, 17] which uses coarse grain task parallelism [15, 16, 17, 18, 19, 20, 21] among loops, subroutines and basic blocks and near fine grain parallelism[22, 23] among statements inside a basic block in addition to conventional loop parallelism among loop iterations. Also, NANOS compiler[24, 25] based on Parafrase2 has been trying to exploit multi-level parallelism including the coarse grain parallelism by using extended OpenMP API. PROMIS compiler[26, 27] hierarchically combines Parafrase2 compiler[28] using HTG[29] and symbolic analysis techniques[30] and EVE compiler for fine grain parallel processing.

Based on OSCAR compiler, Advanced Parallelizing Compiler (APC) project[31] was started in Fiscal Year of 2000 to improve the effective performance, ease of use and cost performance of shared memory multiprocessor systems as a part of Japanese Government Millennium Project IT21 with industries and universities.

In the coarse grain parallelization in OSCAR multigrain compiler, a sequential program is decomposed into three kinds of Macro-Tasks, namely Block of Pseudo Assignment statements (basic block), Repetition Block (loop) and Subroutine Block. Earliest Executable Condition analysis is applied to the generated macro-tasks and generates a macro-task graph. A macro-task graph expresses coarse grain parallelism among macro-tasks. A sequential Repetition Block with a large loop body part or Subroutine Block is decomposed into coarse grain tasks hierarchically as shown in Figure 2. By these hierarchical definition of coarse grain tasks, OSCAR compiler can exploit more parallelism in a program in addition to the loop parallelism.

However, compiler must decide which layer should be parallelized and how many processors should be used for the layer. This decision is very difficult for the ordinary users since analysis of hierarchical parallelism and examination of the combination of hierarchical parallelism are

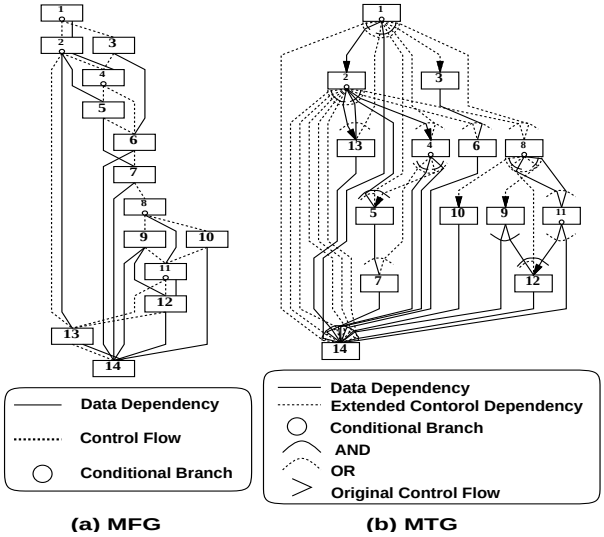

Figure 1. A macro flow graph (MFG) and a macro-task graph (MTG)

very hard. This paper proposes an automatic determination scheme of the number of processors to be assigned to each program layer.

2. Coarse grain task parallel processing

This section describes a coarse grain task parallel processing scheme to decompose a sequential code to coarse grain tasks hierarchically and to generate hierarchical macro-task graph.

The macro-tasks on a macro-task graph are assigned to processor clusters(PC) or processor elements(PE) by a static or dynamic task scheduling method.

2.1. Generation of coarse grain tasks

In the coarse grain task parallelization, a Fortran source program is decomposed into three kinds of macro-tasks, namely, Block of Pseudo Assignment statements (BPA), or Basic Block(BB), repetition Block(RB), or an outermost natural loop in the treated hierarchy, and Subroutine Block(SB). RBs composed of sequential loops having large processing cost and SBs to which inline expansion can not be applied effectively, are hierarchically decomposed into macro-tasks as shown in Figure 2.

2.2. Exploitation of coarse grain parallelism

After the generation of macro-tasks in each layer, or each nest level, control and data flow among macro-tasks

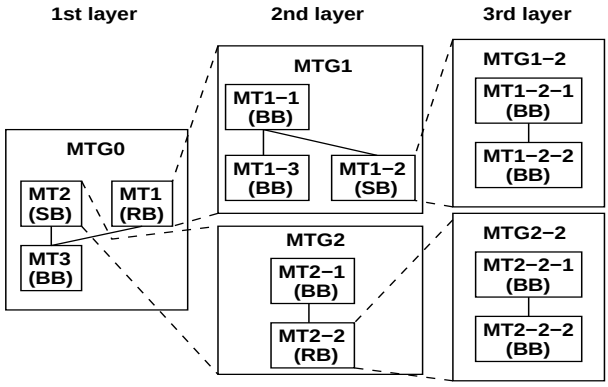

Figure 2. Hierarchical macro-task graph

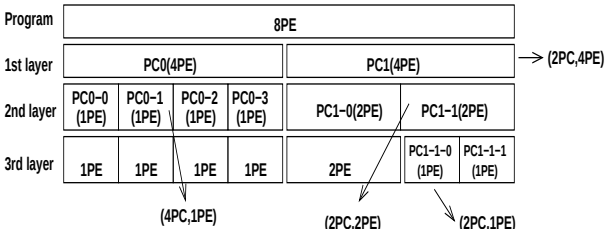

Figure 3. Hierarchical definition of processor clusters and processor elements

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

Then, compiler analyzes Earliest Executable Condition[15, 18, 19] of all macro-task to exploit coarse grain parallelism among macro-tasks. This condition shows parallelism among macro-tasks considering both data dependency and control dependency. Earliest executable condition of each macro-task is shown as macro-task graph in Figure 1(b). In the macro-task graph, nodes represent macro-tasks. A small circle inside nodes represents conditional branches. Solid edges represent data dependencies. Dotted edges represent extended control dependencies. Extended control dependency means ordinary normal control dependency and the condition on which a data dependent predecessor of a macro-task is not executed.

2.3. Processor clusters and processor elements

In the coarse grain parallelization, macro-tasks on hierarchical macro-task graphs are assigned to processor clusters, or groups of processors. OSCAR compiler groups processor elements(PE) into processor clusters(PC) logically, and assigns macro-tasks in a macro-task graph to PCs. If a hierarchical macro-task graph is defined inside a macro-task as shown in Figure 2, processors are also grouped by software logically into PCs hierarchically as shown in Figure 3. Figure 3 shows 8 processors are grouped into 2 processor clusters PC0 and PC1 having 4 processor elements respectively in the first layer. In the second layer, 4 processors in PC0 are grouped into 4 processor clusters PC0-0∼PC0-3. On the other hand, PC1 is decomposed hierarchically to 2 processor clusters PC1-0 and PC1-1 having 2 processor elements respectively in the second layer. Again, PC1-1 having 2 processor elements is grouped hierarchically into 2 processor clusters PC1-1-0 and PC1-1-1 each of which has one PE in the third layer.

Compiler must decide how many PCs should be assigned to each layer of macro-task graphs to exploit full hierarchical parallelism efficiently. Next section handles this problem.

3. Automatic determination of parallel processing layer

This section describes how to decide the number of PCs to be assigned to each layer of macro-task graph (MTG). The parallel processing in the upper layer reduces overheads for synchronization and scheduling because the upper layer tasks usually have larger processing cost compared with overheads. The proposed scheme allows us to use coarse grain task parallelism and loop level parallelism.

3.1. Estimation of macro-task execution cost

First, the compiler estimates processing cost of each macro-task. Sequential cost of each macro-task graph is the sum of sequential cost of macro-tasks considering control flow. If a macro-task is a DO-loop with undefined number of loop iterations and arrays with loop index are accessed in the loop, the compiler estimates the loop processing cost using the dimension size of arrays as the number of loop iterations. However, when the array appeared in the loop doesn't have any relationship with loop index or the number of iterations, the compiler assigns the all PE to the outermost parallelism. If conditional branches are included in a macro-task graph, execution cost is calculated by using branch probability. In this paper, since it is assumed that the compiler doesn't use execution profiles, the cost of macro-tasks is estimated using equal branch probability of 50% for the both conditional branch directions. However, if execution profile can be used, the cost of macro-tasks can be estimated more precisely. The compiler estimates sequential execution cost of each macro-task graph by using the hierarchical sum of inner macro-tasks.

3.2. Calculation of parallelism of each layer of MTG

Coarse grain task parallelism of each macro-task graph(MTG) is calculated by sequential execution cost and critical path length of each MTG. Coarse grain task parallelism $Para_i$ of MTG$_i$ is defined as

$$Para_i = Seq_i / CP_i$$

where CP_i is critical path length and Seq_i is a sequential execution cost in MTG$_i$. Therefore, $\lceil Para_i \rceil$ shows the minimum number of processor clusters(PC) to execute MTG$_i$ in CP_i.

Next, $Para\_ALD_i$ ($Para\ After\ Loop\ Division$) is defined as total parallelism of coarse grain and loop iteration level parallelism. In the proposed determination scheme of parallel processing layer, T_{min} is defined as a minimum task cost for loop parallelization considering overheads of parallel thread fork/join and task scheduling on each target multiprocessor system. This scheme assumes that parallelizable RB in MTG$_i$ is divided into sub RBs having larger cost than T_{min}. However, if the cost of iteration of RB is larger than T_{min}, the maximum number of decomposed tasks is the number of loop iterations of the RB. This task decomposition is considered for only calculation of $Para\_ALD$ and real task decomposition isn't performed at this phase. Critical path length after the temporary task decomposition is represented as CP_ALD_i. Therefore, $Para\_ALD$ is defined by using Seq_i and CP_ALD_i.

$$Para\_ALD_i = Seq_i / CP\_ALD_i$$

If MT$_i$ including MTG$_i$ as a loop body is parallelizable loop, it is necessary to reflect loop parallelism of MT$_i$ itself in $Para\_ALD_i$ hierarchically. In this case, $Para\_ALD_i$ is the product of the inner $Para\_ALD$ of MTG$_i$ and the number of task decomposition of MT$_i$, where the generated macro-tasks by the decomposition of MT$_i$ have larger cost than T_{min}. $\lceil Para\_ALD_i \rceil$ is the total number of processors which is necessary to execute MTG$_i$ in CP_ALD_i and shows the suitable number of processor clusters to balance execution cost among processor clusters. If more processors than $\lceil Para\_ALD_i \rceil$ were assigned to the MTG, possibility of processors being idle is high.

Also, as the enough number of processors for using all parallelism in lower layers of MTG$_i$, $Para\_max_i$ is defined as the following equation:

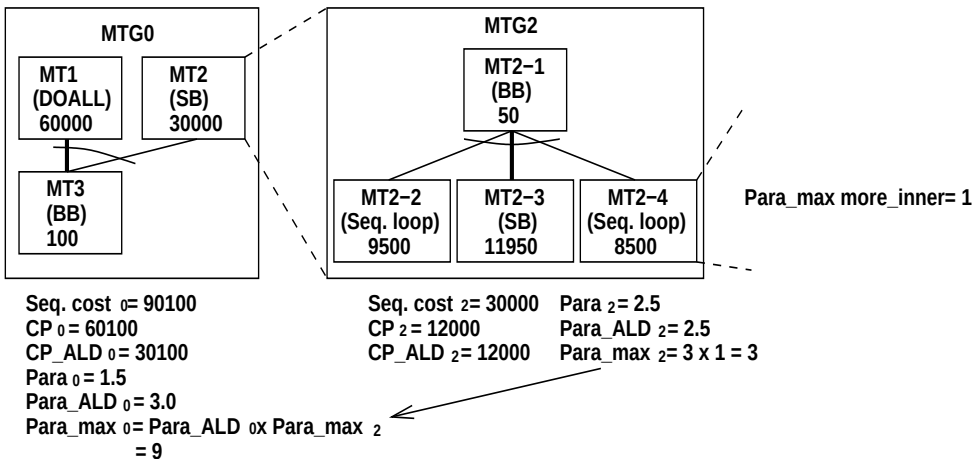

Figure 4. Calculation of $Para, Para\_ALDPara\_max$

$$Para\_max_i = \lceil Para\_ALD_i \rceil \times \lceil Para\_max_{inner} \rceil$$

where $Para\_max_{inner}$ is the maximum $Para\_max$ among macro-tasks in MTG$_i$. However, if RB is a parallelizable loop, the proposed scheme assumes that the loop is divided by $\lceil Para\_ALD_i \rceil$ and $Para\_max$ of the parallelizable loop in MTG$_i$ is calculated by considering the loop decomposition of parallelizable loop. Practically, after the number of processor clusters is determined, the actual number of decomposed tasks of a parallelizable loop is determined in the later stage of compilation by considering the number of processor clusters or cache size. In this phase calculating maximum parallelism, it is assumed that the parallelizable loop in MTG$_i$ is divided by $\lceil Para\_ALD_i \rceil$ which is the number of necessary processors in MTG$_i$.

As an example, $Para$, $Para\_ALD$ and $Para\_max$ in Figure 4 is shown. In Figure 4, "DOALL" shows parallelizable loop, "Seq. loop" shows un-parallelizable loop, namely sequential loop. Thick edges show critical path and numbers within nodes are sequential execution costs. Here, T_{min}, which is the minimum cost to realize efficient loop parallel processing, is defined as 10000. To explain simply, it is assumed that there are no parallelism in the body of MT1(DOALL) in MTG$_0$, MT2-2, MT2-3 and MT2-4 in MTG$_2$ which is inner macro-task graph of MT2(SB). Though hierarchical macro-task graphs can be generated inside these macro-tasks practically, the inner macro-task graphs of MT1, MT2-2, MT2-3 and MT2-4 are omitted in this example. First, $Para$, CP, $Para\_ALD$, CP_ALD and $Para\_max$ are calculated from the deepest layer of a program. Since macro-task graphs within MT2-2, MT2-3 and MT2-4 don't have any parallelism as mentioned above, $Para = Para\_ALD = Para\_max = 1$ for these loops.

Sequential cost of MTG$_2$ is 30000 and CP_2, CP_ALD_2 are 12000. Therefore, $Para_2$ and $Para\_ALD_2$ of MTG$_1$ are $Para_2 = 30000/12000 = 2.5$, $Para\_ALD_2 = 30000/12000 = 2.5$ Also, since $Para\_max = 1$ in MT2-2, MT2-3 and MT2-4 of MTG$_2$ and $Para\_max_2 = \lceil Para\_ALD_2 \rceil \times Para\_max_{more\_inner} = 3 \times 1 = 3$, the suitable number of processors which is assigned to MTG$_2$ is 3.

Also, parallelizable loop MT1 (DOALL) can be divided into 6 (60000/10000) sub macro-tasks. Therefore, $Para_1 = 1$, $Para\_ALD_1 = 6$ for MT1(DOALL). Then, each parameter in MTG$_0$ is calculated. In Figure 4, $Seq_0 = 90100$ and $CP_0 = 60100$. $CP\_ALD_0 = 30100$ is the sum of sequential cost of MT2(SB) and MT3(BB) if MT1(DOALL) is divided into 6 tasks (60000/10000). Therefore, $Para_0 = 90100/60100 = 1.5$, $Para\_ALD_0 = 90100/30100 = 3.0$. For $Para\_max_0$, macro-tasks within MTG$_0$ are MT1 through MT3 and it is assumed that MT1 is divided by $Para\_ALD_0 = 3$. $Para\_max_2 = 3$ of MT2(SB) which has MTG$_2$ as mentioned above. Though original $Para\_max$ of MT1(DOALL) before task division is $Para\_max = 6$, $Para\_max = 6/3 = 2$ in one of divided MT1(DOALL) since MT1(DOALL) is divided by $Para\_ALD_0 = 3$. Therefore, MT having maximum $Para\_max$ in MTG$_0$ is MT2 and $Para\_max_0 = \lceil Para\_ALD_0 \rceil \times Para\_max_2 = 9$.

3.3. Determination scheme of PC and PE assignment to each layer

This section describes an automatic determination scheme of PC and PE assignment to each layer is described by using parameters obtained in section 3.2.

Step 1 Since execution costs of tasks in an upper layer are

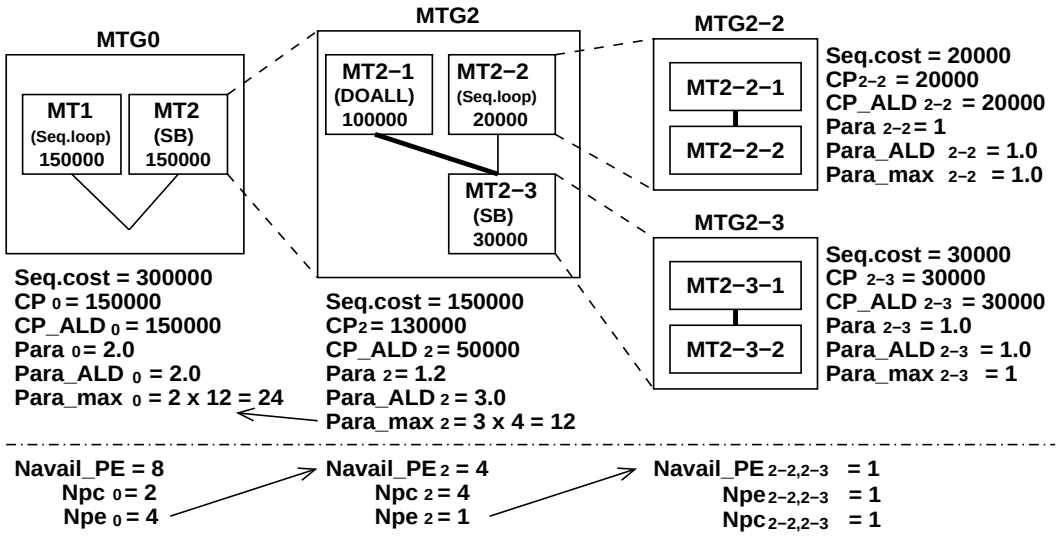

Figure 5. Determination of N_{PC} and N_{PE}

larger than tasks in a lower layer in a program in general, relative overheads of task scheduling and synchronization are relatively small. Thus, the proposed scheme tries to use more parallelism in an upper layer. Let's assume the number of processors which can be used in MTG$_i$ is N_{Avail_PEi}. The number of processor clusters and processor elements are denoted as N_{PCi} and N_{PEi}. Relationship between $Para_i$ and N_{PCi} should be $Para_i \leq N_{PCi}$ to fully use coarse grain task parallelism. Furthermore, if the number of processor clusters is larger than $Para\_ALD_i$, the some processor clusters may become idle. Therefore, the combination of [N_{PCi}, N_{PEi}] is defined as follows:

$$Para_i \leq N_{PCi} \leq Para\_ALD_i$$

$$N_{PCi} \times N_{PEi} = N_{Avail\_PEi}$$

If $Para_i = Para\_ALD_i$, the number of processor clusters N_{PCi} for MTG$_i$ is selected as the minimum number which satisfies $Para_i \leq N_{PCi}$ and $N_{PCi} \times N_{PEi} = N_{Avail\_PEi}$ to use coarse grain task parallelism in MTG$_i$ as much as possible. If $Para_i \geq N_{Avail\_PEi}$, the combination of processor clusters and processor elements will be $N_{PCi} = N_{Avail\_PEi}$ and $N_{PEi} = 1$.

Step 2 $MaxN_{PEi}$ is defined as the maximum $Para\_max$ among macro-tasks which are not parallelizable loop in MTG$_i$. $MaxN_{PEi}$ means an upper limit of the number of processors which can be assigned to lower layer of MTG$_i$, namely the upper limit of N_{PEi}. If $N_{PEi} > MaxN_{PEi}$ is used, possibility of unnecessary synchronization and number of overheads of task scheduling are high since excessive processor elements are assigned to a lower layer. To avoid such case, $N_{PEi} = MaxN_{PEi}$ is chosen in this step.

Step 3.a If all of the macro-tasks in MTG$_i$ is not parallelizable loops, current N_{PCi} is chosen as the number of processor clusters assigned to the MTG$_i$.

Step 3.b If $MaxN_{PEi}$ is smaller than current N_{PEi}, N_{PEi} is set to $MaxN_{PEi}$ in Step 2. However, when MTG$_i$ has parallelizable loops and $N_{PCi} \times N_{PEi} < N_{Avail\_PEi}$ is satisfied, possibility of lack of available processors to parallelize these loops effectively is high. In this case, $N_{PCi} \times MaxN_{PEi}$ is set to be over the upper limit of the number of processors in MTG$_i$ $MaxN_{PCPEi} = Para\_max_i$. $MaxN_{PCPEi}$ means an upper limit of total number of processors which are assigned to MTG$_i$ and its lower layer. Therefore, the proposed scheme chooses the minimum N_{PCi} which satisfies $N_{PCi} \times MaxN_{PEi} \geq MaxN_{PCPEi}$. However, if $N_{PCi} \times MaxN_{PEi} > N_{Avail\_PEi}$, the proposed scheme chooses the maximum N_{PCi} which is $N_{PCi} \times MaxN_{PEi} \leq N_{Avail\_PEi}$.

These steps are started from highest layer in a program to lower layers until $N_{Avail\_PEi} = 1$.

Figure 5 explains the processor assignment scheme. Figure 5 consists of 4 hierarchical macro-task graphs, namely the highest, or the first, level MTG0, the second level MTG2 and the third level MTG2-2 and MTG2-3. Sequential cost, CP, Para and so on are shown in Figure 5. It is assumed that

MTG2-2 and MTG2-3 have no parallelism and no lower layers. Therefore, $Para = Para\_ALD = Para\_max = 1$. The number of available processors is eight, namely $N_{Avail\_PE0} = 8$. For MTG_0, since $N_{PC0} = 2$ from $Para_0 = 2 \leq N_{PC0} \leq Para\_ALD = 2$ in Figure 5 , the combination of [N_{PC0}, N_{PE0}] is [2PC, 4PE] from Step 1. Also, $MaxN_{PE0} = Para\_max_2 = 12$ since MT1 and MT2 in MTG_0 aren't parallelizable loops. Therefore, the proposed scheme understands that the lower layer of MTG_0, or MTG_2, needs 4 processors, and determines $N_{PE0} = 4$ from Step 2. The combination [N_{PC2}, N_{PE2}] is determined. Since $N_{PE0} = 4$, $N_{Avail\_PE2} = 4$ in MTG_2. Here, the possible combinations of [N_{PC2}, N_{PE2}] are [1, 4], [2, 2] and [4, 1]. $N_{PC2} = 2$ is satisfied $Para_2 = 1.2 \leq N_{PC2} \leq Para\_ALD_2 = 3.0$ in Figure 5 and available combinations of [N_{PC2}, N_{PE2}] from Step 1. Though $N_{PE2} = 2$, $MaxN_{PE2} = Para\_max_{2-2,2-3} = 1$. Thus $N_{PE2} = 1$ from Step 2. By Step 3b, since $MaxN_{PCPE2} = 12$ in Figure 5, N_{PC2} which satisfies $N_{PC2} \times MaxN_{PE2} \leq MaxN_{PCPE2} = 12$ is $N_{PC2} = 4$, thus [N_{PC2}, N_{PE2}] = [4PC, 1PE]. Here, the process of assignment of PC and PE is ended since $N_{Avail\_PE2-2,2-3} = 1$.

Therefore, the proposed scheme determines [N_{PC0}, N_{PE0}] = [2PC, 4PE], [N_{PC2}, N_{PE2}] = [4PC, 1PE] and [$N_{PC2-2,2-3}$, $N_{PE2-2,2-3}$] = [1PC, 1PE].

4. Performance evaluation

This section evaluates the performance of the proposed parallelism control scheme for multigrain parallel processing on IBM RS6000 SP 604e High Node 8 processors SMP server. This scheme was implemented in OSCAR multigrain parallelizing compiler.

4.1. Evaluation environment

In this evaluation, OSCAR compiler with the proposed scheme is used as a parallelizing pre-processor and generates a coarse grain task parallel program with OpenMP API. This OpenMP program uses the "one time single level thread generation" scheme which can minimize thread generation overhead by forking and joining parallel threads at the beginning and the end of the program only once and realize hierarchical coarse grain task parallel processing [32, 33].

The generated OpenMP program is compiled by IBM XL Fortran for AIX Version 7.1 and executed on IBM RS6000 SP 604e High Node. This machine is a SMP server having 8 Power PC 604e processors (200MHz). Each processor has 32 Kbytes instruction / data L1 cache respectively and 1 MB unified L2 cache. A size of shared main memory is 1 GB.

In this evaluation, the best compile options, by which XL Fortran compiler gave us minimum processing time for sequential and parallel execution respectively, are used. However, the other parameter tuning for OS and runtime library isn't performed to only evaluate the pure performance of compilers.

4.2. Evaluation result of SPEC95FP

In this evaluation, 8 programs from SPEC95fp, such as SWIM, TOMCATV, MGRID, HYDRO2D, SU2COR, TURB3D, APPLU and FPPPP, are used and the performance of OSCAR compiler and XL Fortran compiler is compared. Compilation options for native XL Fortran compiler are "-O3 -qsmp=noauto -qhot -qarch=ppc -qtune=auto -qcache=auto -qstrict" for OpenMP programs generated by OSCAR compiler, "-O5 -qsmp=auto -qhot -qarch=ppc -qtune=auto -qcache=auto" for automatic parallelization by XL Fortran compiler and "-O5 -qhot -qarch=ppc -qtune=auto -qcache=auto" for sequential execution. For two programs, such as SU2COR and TURB3D, manual restructuring, such as inline expansion, array renaming, loop distribution were made to avoid OSCAR compiler's bugs and the same restructured programs are used for sequential execution, automatic loop parallelization and coarse grain parallelization.

Execution time of SPEC95FP is shown in Table 1. Also, Figure 6 shows speedup ratio of each SPEC95FP program by 8 processors against sequential execution time. In Table 1, the sequential processing time by XL Fortran, the automatic loop parallel processing time by XL Fortran using 8 processors, the shortest execution time by XL Fortran using up to 8 processors and the coarse grain task parallel processing time by OSCAR compiler using 8 processors are shown for each SPEC95FP programs.

In SWIM on Table 1, the sequential execution time was 549.1 seconds and the shortest parallel processing time by automatic loop parallelization by XL Fortran was 112.6 seconds. OSCAR compiler using the proposed scheme was 64.4 seconds and gave us 8.53 times speedup against the sequential time as shown in Figure 6. OSCAR compiler's "one time single level thread generation" with the parallelism control could boost up 1.75 times the maximum performance of XL Fortran though XL Fortran suffered from large thread management overhead.

In TOMCATV and HYDRO2D, sequential execution times for TOMCATV and HYDRO2D were 636.8 and 987.7 seconds. OSCAR compiler's execution times were 107.2 seconds for TOMCATV and 116.7 seconds for HYDRO2D as shown in Table 1. The shortest execution time by automatic loop parallelization of XL Fortran was 373.0 seconds for TOMCATV using 3 processors and 426.2 seconds for HYDRO2D using 4 processors. OSCAR compiler

Table 1. Execution time(seconds) of SPEC95FP on 8 processors IBM RS6000 SP 604e High Node

Benchmark	SWIM	TOMCATV	HYDRO2D	MGRID	SU2COR	TURB3D	APPLU	FPPPP
Sequential	549.1	636.8	987.7	592.0	517.2	649.0	707.4	505.9
XLF for 8PEs	130.6	1180.5	620.7	344.8	941.9	2071.9	489.9	506.3
XLF minimum (PE)	112.6(6)	373.0(3)	426.2(4)	193.0(4)	197.9(4)	649.0(1)	489.9(8)	505.9(1)
OFC for 8PEs	64.4	107.2	116.7	93.6	120.1	197.9	423.7	506.0

*OFC: OSCAR FORTRAN COMPILER, XLF: XL Fortran, ( ): the number of PEs giving the minimum time

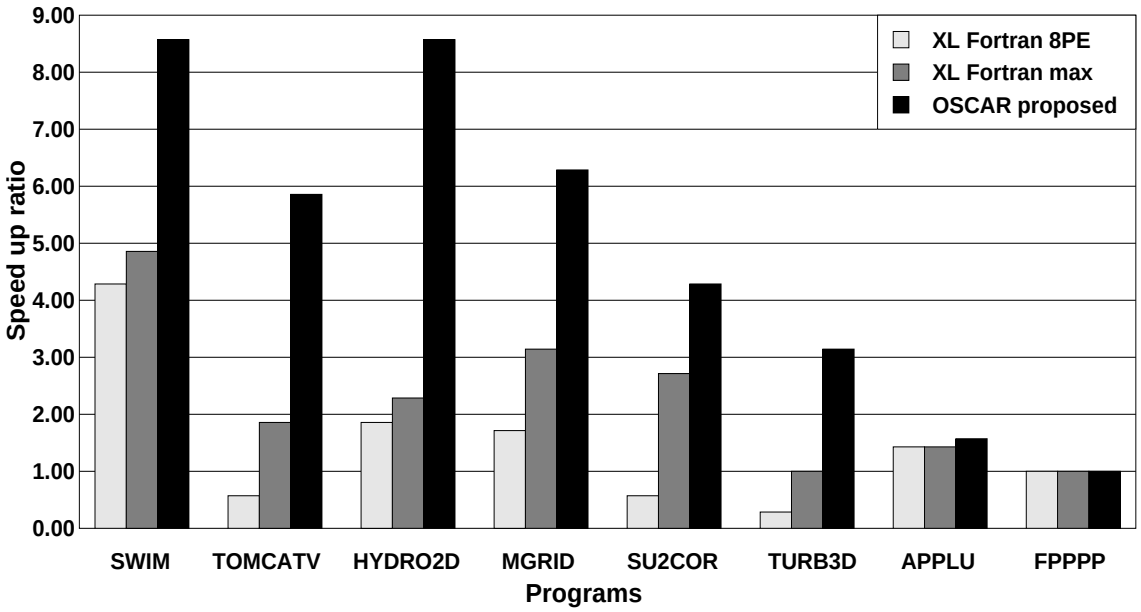

Figure 6. Speedup ratio of SPEC95FP using 8 processors

gave us 5.94 times speedup for TOMCATV and 8.46 times speedup for HYDRO2D compared with the sequential execution as shown in Figure 6 and boosted up 3.48 times for TOMCATV and 3.65 times for HYDRO2D the maximum performance of XL Fortran. Though TOMCATV and HYDRO2D consists of parallelizable loops, the proposed parallelism control scheme could find parallelizable loops which should not be processed in parallel.

In MGRID of Table 1, sequential execution time was 592.0 seconds and the shortest automatic loop parallel processing time by XL Fortran was 193.0 seconds. OSCAR compiler gave us 93.6 seconds, or 6.32 times speedup against the sequential execution as shown in Figure 6. The proposed scheme assigns all processors to the outermost parallelism in MGRID as section 3.1 because this program uses adjustable array and change array dimension in subroutines and function calls.

In SU2COR of Table 1, sequential execution time was 517.2 seconds. Though the execution time of loop parallelization by XL Fortran was 197.9 seconds, OSCAR compiler obtained 120.1 seconds, or 4.31 times speedup against the sequential execution as shown in Figure 6. Therefore OSCAR compiler boosted up 1.65 times the performance of XL Fortran. XL Fortran compiler used loop level parallelism in the deepest nest level with relatively small cost. On the contrary, the proposed scheme could find coarse grain task parallelism in the upper layer which is the inside of loop block "DO 400" in subroutine LOOPS. Since this layer has $Para = 1.90$, $Para\_ALD = 3.00$, the proposed scheme determines the combination of PC and PE [2PC, 4PE] and can use coarse grain parallelism effectively.

In TURB3D of Table 1, execution time by XL Fortran was 649.0 seconds for sequential execution time and the shortest time by loop parallel processing using up to 8 processors. The OpenMP code generated by OSCAR compiler gave us 197.9 seconds, and boosted up 3.28 times the performance of XL Fortran as shown in Figure 6. In TURB3D, coarse grain parallelism was extracted and $Para = 5.98$ was calculated by OSCAR compiler in RB of subroutine TURB3D. Therefore, the proposed scheme chooses the

combination of [8PC, 1PE] since $Para \leq N_{PC}$, and gave us better performance.

In APPLU of Table 1, sequential execution time was 707.4 seconds. Though automatic loop parallel processing time by XL Fortran was 489.9 seconds, coarse grain parallel processing time by OSCAR compiler was 423.7 seconds, or 1.67 times speedup against sequential execution time as shown in Figure 6. APPLU has five subroutine including parallelizable blocks having large execution cost, namely JACLD, JACU, RHS, BUTS and BLTS. Current OSCAR compiler uses parallelism in subroutines JACLD, JACU and RHS and can not parallelize subroutine BUTS and BLTS. As the results, the proposed parallelism control scheme parallelizes the inside of subroutine JACLD, JACU and RHS automatically.

Finally, in FPPPP, execution time by XL Fortran and OSCAR compiler was the same 506 seconds as shown in Table 1. XL Fortran compiler and OSCAR compiler found no parallelism in FPPPP. However, there is statement level near fine grain parallelism in subroutine FPPPP and the parallelism can be exploited by OSCAR multigrain compiler on OSCAR chip multiprocessor[23].

5. Conclusions

This paper has proposed the automatic determination scheme of parallel processing layer and the number of processors to be assigned to each layer of macro-task graph for multigrain parallel processing. The performance evaluation of OSCAR compiler with the proposed scheme using SPEC95FP on IBM RS6000 SP 604e High Node 8 processors SMP server showed OSCAR compiler gave us 8.53 times speedup for SWIM, 5.94 times speedup for TOMCATV, 8.46 times speedup for HYDRO2D, 6.32 times speedup for MGRID, 4.31 times speedup for SU2COR, 3.28 times speedup for TURB3D, 1.67 time speedup for APPLU and 1.00 times speedup for FPPPP compared with the sequential execution by XL Fortran for AIX Version 7.1. Also, OpenMP coarse grain task parallel code generated by OSCAR compiler boosted up the performance of XL Fortran 1.75 times for SWIM, 3.48 times for TOMCATV, 3.65 times for HYDRO2D, 2.06 times for MGRID, 1.65 times for in SU2COR, 3.28 times for TURB3D and 1.16 times for in APPLU. From these results, it was confirmed the proposed scheme could find suitable or un-suitable layer for parallel processing and assign the suitable number of processors to each layer of macro-task graph without performance degradation with the increase processors.

Currently, the authors are researching on a chip multiprocessor[23] which supports the multigrain parallel processing and performance evaluation of the OSCAR compiler on larger scale multiprocessor systems.

Acknowledgment

A part of this research has been supported by Japan Government Millennium Project METI/NEDO Advanced Parallelizing Compiler Project (http: //www.apc.waseda.ac.jp) and Waseda University Grant for Special Research Projects No.2000A-154.

References

[1] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[2] U. Banerjee. Loop parallelization. *Kluwer Academic Pub.*, 1994.

[3] Polaris. http://polaris.cs.uiuc.edu/polaris/.

[4] R. Eigenmann, J. Hoeflinger, and D. Padua. On the automatic parallelization of the perfect benchmarks. *IEEE Trans. on parallel and distributed systems*, 9(1), Jan. 1998.

[5] L. Rauchwerger, N. M. Amato, and D. A. Padua. Run-time methods for parallelizing partially parallel loops. *Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona, Spain*, pages 137–146, Jul. 1995.

[6] P. Tu and D. Padua. Automatic array privatization. *Proc. 6th Annual Workshop on Languages and Compilers for Parallel Computing*, 1993.

[7] M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S. Liao, , and M. S. Lam. Interprocedural parallelization analysis: A case study. *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing (LCPC95)*, Aug. 1995.

[8] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, 1996.

[9] S. Amarasinghe, J. Anderson, M. Lam, and C. Tseng. The suif compiler for scalable parallel machines. *Proc. of the 7th SIAM conference on parallel processing for scientific computing*, 1995.

[10] M. S. Lam. Locallity optimizations for parallel machines. *Third Joint International Conference on Vector and Parallel Processing*, Nov. 1994.

[11] A. W. Lim and M. S. Lam. Cache optimizations with affine partitioning. *Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing*, Mar. 2001.

[12] A. Yoshida, K. Koshizuka, M. Okamoto, and H. Kasahara. A data-localization scheme among loops for each layer in hierarchical coarse grain parallel processing. *Trans. of IPSJ (japanese)*, 40(5), May. 1999.

[13] G. Rivera and C.-W. Tseng. Locality optimizations for multi-level caches. *Super Computing '99*, Nov. 1999.

[14] H. Han, G. Rivera, and C.-W. Tseng. Software support for improving locality in scientific codes. *8th Workshop on Compilers for Parallel Computers (CPC'2000)*, Jan. 2000.

[15] H. Kasahara, H. Honda, A. Mogi, A. Ogura, K. Fujiwara, and S. Narita. A multi-grain parallelizing compilation scheme on oscar. *Proc. 4th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1991.

[16] M. Okamoto, K. Aida, M. Miyazawa, H. Honda, and H. Kasahara. A hierarchical macro-dataflow computation scheme of oscar multi-grain compiler. *Trans. of IPSJ (japanese)*, 35(4):513–521, Apr. 1994.

[17] H. Kasahara, M. Okamoto, A. Yoshida, W. Ogata, K. Kimura, G. Matsui, H. Matsuzaki, and H.Honda. Oscar multi-grain architecture and its evaluation. *Proc. International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, Oct. 1997.

[18] H. Kasahara, H. Honda, M. Iwata, and M. Hirota. A macro-dataflow compilation scheme for hierarchical multiprocessor systems. *Proc. Int'l. Conf. on Parallel Processing*, Aug. 1990.

[19] H. Honda, M. Iwata, and H. Kasahara. Coarse grain parallelism detection scheme of fortran programs. *Trans. IEICE (in Japanese)*, J73-D-I(12), Dec. 1990.

[20] H. Kasahara. *Parallel Processing Technology*. Corona Publishing, Tokyo (in Japanese), Jun. 1991.

[21] H. Kasahara, M. Obata, and K. Ishizaka. Automatic coarse grain task parallel processing on smp using openmp. *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing (LCPC2000)*, Aug. 2000.

[22] H. Kasahara, H. Honda, and S. Narita. Parallel processing of near fine grain tasks using static scheduling on oscar. *Proc. IEEE ACM Supercomputing'90*, Nov. 1990.

[23] K. Kimura, T. Kato, and H. Kasahara. Evaluation of processor core architecture for single chip multiprocessor with near fine grain parallel processing. *Trans. of IPSJ (japanese)*, 42(4), Apr. 2001.

[24] X. Martorell, E. Ayguade, N. Navarro, J. Corbalan, M. Gozalez, and J. Labarta. Thread fork/join techniques for multi-level parllelism exploitation in numa multiprocessors. *ICS'99 Rhodes Greece*, 1999.

[25] E. Ayguade, X. Martorell, J. Labarta, M. Gonzalez, and N. Navarro. Exploiting multiple levels of parallelism in openmp: A case study. *ICPP'99*, Sep. 1999.

[26] PROMIS. http://www.csrd.uiuc.edu/promis/.

[27] C. J. Brownhill, A. Nicolau, S. Novack, and C. D. Polychronopoulos. Achieving multi-level parallelization. *Proc. of ISHPC'97*, Nov. 1997.

[28] Parafrase2. http://www.csrd.uiuc.edu/parafrase2/.

[29] M. Girkar and C. Polychronopoulos. Optimization of data/control conditions in task graphs. *Proc. 4th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1991.

[30] M. R. Haghighat and C. D. Polychronopoulos. *Symbolic Analysis for Parallelizing Compliers*. Kluwer Academic Publishers, 1995.

[31] http://www.apc.waseda.ac.jp/.

[32] H. Kasahara, M. Obata, and K. Ishizaka. Coarse grain task parallel processing on a shared memory multiprocessor system. *Trans. of IPSJ (japanese)*, 42(4), Apr. 2001.

[33] M. Obata, K. Ishizaka, and H. Kasahara. Automatic coarse grain task parallel processing using oscar multigrain parallelizing compiler. *Ninth International Workshop on Compilers for Parallel Computers(CPC 2001)*, Jun. 2001.