IPSJ Transactions on Advanced Computing Systems Vol.7 No.4 56–66 (Dec. 2014)

Regular Paper

Parallel Java Code Generation for Layer-unified Coarse Grain Task Parallel Processing

Akimasa Yoshida[1,a)] Yuki Ochi[2] Nagatsugu Yamanouchi[2,b)]

Received: March 31, 2014, Accepted: July 30, 2014

Abstract: Multicore processors are widely used for various types of computers. In order to achieve high-performance on such multicore systems, it is necessary to extract coarse grain task parallelism from a target program in addition to loop parallelism. Regarding the development of parallel programs, Java or a Java-extension language represents an attractive choice recently, thanks to its performance improvement as well as its platform independence. Therefore, this paper proposes a parallel Java code generation scheme that realizes coarse grain task parallel processing with layer-unified execution control. In this parallel processing, coarse grain tasks of all layers are collectively managed through a dynamic scheduler. In addition, we have developed a prototype parallelizing compiler for Java programs with directives. In performance evaluations, the compiler-generated parallel Java code was confirmed to attain high performance. Concretely, we obtained 7.82 times faster speed-up for the Jacobi program, 7.38 times faster speed-up for the Turb3d program, 6.54 times faster speed-up for the Crypt program, and 6.15 times faster speed-up for the MolDyn program on eight cores of Xeon E5-2660.

Keywords: coarse grain parallelization, compiler, dynamic scheduling, Java, Multicore

1. Introduction

Recently, multicore processors are widely used for various kinds of computers, such as supercomputers, PCs, tablet computers, and embedded systems. To achieve high-performance on multicore processors, in addition to loop parallelization [1], [2], coarse grain task parallelization [3], [4], [5], [6] or multi-level parallelization [7], [8] is indispensable.

In coarse grain task parallel processing [3], the parallelizing compiler [9] extracts parallelism among coarse grain tasks (i.e., macrotasks), represents inter-macrotask parallelism as a hierarchical macrotask-graph (MTG), and generates a parallel code to assign macrotasks to hierarchical core groups. Given that macrotasks within a certain layer in a target program are executed on a corresponding layer of core groups, if the number of cores is not sufficient to utilize parallelism of all layers, the use of parallelism must be restricted.

To solve such a problem, the layer-unified execution control scheme [10] for coarse grain task parallel processing, namely the layer-unified coarse grain task parallel processing, was proposed. This scheme extends the hierarchical MTG [3] by addition of the layer-start macrotask that enables the uniform control of macrotasks of different layers.

In addition, although C and Fortran languages are conventionally used as target languages for parallel computing, the interest in Java for high performance computing is recently increasing [11]. This is because the performance gap between Java and native languages such as C and Fortran has been narrowing for the last few years, owing to the Just-In-Time compiler of the Java Virtual Machine. Also, an extended-Java-based research such as the Habanero extreme scale software research project is active for scientific computing [12].

Therefore, this paper proposes a parallel Java code generation scheme to realize layer-unified coarse grain task parallel processing, and develops a parallelizing compiler to generate the parallel Java code, which is platform-independent, from a Java program with parallelization directives. Moreover the parallel Java code that adopts the selective static data structure attains a higher performance on multicore processors.

The rest of this paper is organized as follows. Section 2 describes layer-unified coarse grain task parallel processing in Java. Section 3 describes a parallel Java code generation scheme for layer-unified coarse grain task parallel processing. Section 4 describes a parallelizing compiler to generate a parallel Java code. Section 5 evaluates the proposed scheme by using several scientific programs on a multicore system. Section 6 describes related works. Finally, section 7 presents our conclusions.

2. Layer-unified Coarse Grain Task Parallel Processing in Java

This section describes the coarse grain task parallel processing scheme with layer-unified execution control in Java. The parallel processing scheme is extended from its Fortran-based concept [10]. The details are described below.

2.1 Layer-unified Execution Control

This subsection describes the layer-unified execution control for coarse grain task parallel processing, and also shows the dif-

[1] Meiji University, Nakano, Tokyo 164–8525, Japan
[2] Toho University, Funabashi, Chiba 274–8510, Japan
[a)] akimasay@meiji.ac.jp
[b)] yamanouc@is.sci.toho-u.ac.jp

```
class Other {          //user-defined class
  public static void func() {
    /*mt*/ {
    macrotask code;                    //MT3-1
    }
    /*mt*/ {
    macrotask code;                    //MT3-2
    }
  }
}
```
MTG3

```
public class Main {
  public static void main(String[] args) {
    /*mt*/ {
    macrotask code;                    //MT1-1
    }
    /*mt*/ {
    for (int i=0; i<2; i++) {          //MT1-2
      /*mt*/ {
      macrotask code;                  //MT2-1
      }
      /*mt*/ {
      Other.func();                    //MT2-2
      }
    }
    }
    /*mt*/ {
    macrotask code;                    //MT1-3
    }
  }
}
```
MTG1 / MTG2

Fig. 1 Java program with parallelization directives.

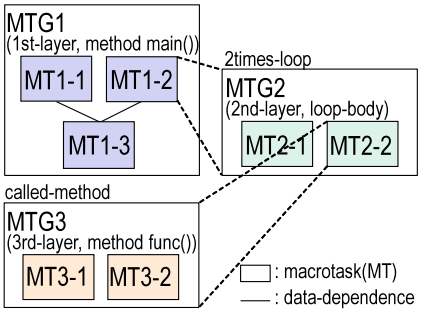

Fig. 2 Hierarchical macrotask-graph (MTG).

ference from the hierarchical execution control.

A sample Java program as shown in **Fig. 1** is composed of three layers of macrotasks as follows: first-layer macrotasks such as MT1-1, MT1-2 and MT1-3; second-layer macrotasks such as MT2-1 and MT2-2 in the loop body of MT1-2; and third-layer macrotasks such as MT3-1 and MT3-2 in the method invoked by MT2-2. This program is also represented by a hierarchical MTG in **Fig. 2** where data-dependence edges among macrotasks are assumed for explanation.

When we apply coarse grain task parallel processing with layer-unified execution control to the program shown in Fig. 1, the execution image on four cores without grouping is shown in **Fig. 3** (a), where macrotasks of all layers are scheduled to cores uniformly by the dynamic scheduler. MT1-2S and MT2-2S represent the layer-start macrotask corresponding to MT1-2 and MT2-2 respectively as described in Section 2.3. This scheme can use parallelism across the layers of MT1-1, MT2-1, MT3-1 and MT3-2, and can reduce the execution time remarkably.

For comparison, in the hierarchical execution control [3], macrotasks of each layer are hierarchically assigned to a corresponding layer of core groups. For example, the execution on two core groups, each of which has two cores (four cores in total), is shown in Fig. 3 (b). In the first layer in Fig. 2, MT1-1 and MT1-2 are assigned to CoreGroup0 and CoreGroup1 respectively, and then MT1-3 is assigned to CoreGroup0. In the second layer within MT1-2, MT2-1 and MT2-2 are assigned to Core2 and Core3 within CoreGroup1 respectively. Therefore, in the case of MT2-2 assigned to Core3, third-layer macrotasks namely MT3-1 and MT3-2 need to be executed in serial.

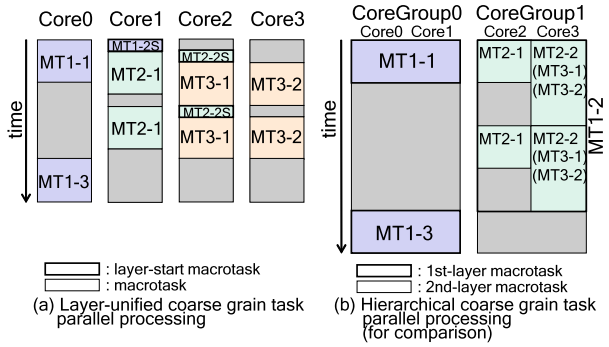

Fig. 3 Comparison of execution control for coarse grain task parallel processing.

2.2 Hierarchical Macrotask

In coarse grain task parallel processing [3], [10], the entire program is first decomposed into first-layer macrotasks. A macrotask is classified into one of three types: a basic block, a repetition block (e.g., a for-statement, a while-statement), and a subroutine block (e.g., a class method, an instance method).

Next, when a first-layer macrotask contains several sub-macrotasks, its sub-macrotasks are defined as second-layer macrotasks. Furthermore, $(L+1)$th-layer macrotasks are defined in the Lth-layer macrotask. Note that when a repetition block is a parallelizable loop or a reduction loop that requires a long processing time, the loop is decomposed into several partial loops, each of which is defined as a macrotask.

For example, in the Java program shown in Fig. 1, first-layer macrotasks, second-layer macrotasks and third-layer macrotask are defined hierarchically as shown in Fig. 2.

2.3 Layer-start Macrotask

In the layer-unified execution control [10], the layer-start macrotask is introduced to uniformly control macrotasks of all layers. The $(L-1)$th-layer macrotask that surrounds Lth-layer macrotasks is controlled by the layer-start macrotask of the Lth-layer. Given that the finish of the layer-start macrotask ensures that macrotasks of the layer are executable, macrotasks of all layers can be collectively controlled by the dynamic scheduler.

As shown in Fig. 2, in the case of the repetition block MT1-2 that contains MT2-1 and MT2-2, the macrotask MT1-2 is treated as a layer-start macrotask. In the case of the method invocation MT2-2 that contains MT3-1 and MT3-2, the macrotask MT2-2 is treated as a layer-start macrotask.

2.4 Earliest Executable Condition

After generation of macrotasks, the compiler analyzes control flow and data flow among macrotasks on each layer, and generates a hierarchical macro-flow graph [3]. Next, in order to extract

inter-macrotask parallelism taking into consideration control dependence and data dependence, the compiler analyzes the earliest executable condition [3]. The earliest executable condition, which represents parallelism among macrotasks, is reflected in the dynamic scheduler to assign macrotasks. The dynamic scheduler can detect newly executable macrotasks by examining the earliest executable condition, taking into consideration the finish notification and the branch notification [13].

The earliest executable condition and the finish notification of macrotasks in Fig. 2 are displayed in **Table 1**. In the earliest executable condition, the symbol *i* represents the finish of MT*i*; the symbol $(i)_j$ represents the branch from MT*i* to MT*j*; and the symbol i_j represents both the branch from MT*i* to MT*j* and the finish of MT*i*. Moreover, dummy macrotasks such as EndMT (i.e., the finish processing of a program), CtrlMT (i.e., the repetition control), RepMT (i.e., the renewal processing of repetition) and ExitMT (i.e., the finish processing of a layer) are used in dynamic scheduling.

For example, the earliest executable condition of MT1-3 shown in Fig. 2 is expressed as the logical-expression 1-1∧1-2, which means that MT1-3 is executable after MT1-1 and MT1-2 have finished. Also, MT2-3(CtrlMT) controls the condition of repetition. Concretely, MT2-3(CtrlMT) compares a loop induction variable with the loop's upper limit; it issues the branch notification to MT2-4(RepMT) like $2\text{-}3_{2\text{-}4}$ or the branch notification to MT2-5(ExitMT) like $2\text{-}3_{2\text{-}5}$; this branch notification can satisfy the earliest executable condition MT2-4 or MT2-5; then MT2-4 or MT2-5 is executable. The earliest executable condition of the macrotasks is also represented by a hierarchical MTG [3], where dummy macrotasks are omitted, as shown in Fig. 2.

Next, considering the layer-start macrotask, the earliest executable condition on each layer is extended for layer-unified execution control [10]. Specifically, when the earliest executable condition of the *L*th-layer macrotask is true, the condition is replaced with the finish of the layer-start macrotask MT*i*. The finish notification of the layer-start macrotask MT*i*, which is represented as the symbol *i*S, is issued by the layer-start macrotask itself. The finish notification for the *L*th-layer MTG within MT*i*, which is represented as the symbol *i*, is issued by the ExitMT of the *L*th-layer.

Table 1 Earliest executable condition.

MTG number	MT number	Earliest executable condition	Finish notification
1	MT1-1	true	1-1
1	MT1-2 †	true	**1-2S**
1	MT1-3	1-1∧1-2	1-3
1	MT1-4(EndMT)	1-3	1-4
2	MT2-1	**1-2S**	2-1
2	MT2-2 ††	**1-2S**	**2-2S**
2	MT2-3(CtrlMT)	2-1∧2-2	2-3
2	MT2-4(RepMT)	$2\text{-}3_{2\text{-}4}$	2-4
2	MT2-5(ExitMT)	$2\text{-}3_{2\text{-}5}$	**1-2**
3	MT3-1	**2-2S**	3-1
3	MT3-2	**2-2S**	3-2
3	MT3-3(CtrlMT)	3-1∧3-2	3-3
3	MT3-4(ExitMT)	$3\text{-}3_{3\text{-}4}$	**2-2**

Notes: † layer-start MT of second layer MTG2.
†† layer-start MT of third layer MTG3.

In Table 1, the earliest executable conditions of MT2-1 and MT2-2 are represented by 1-2S meaning the finish of the layer-start macrotask MT1-2. The finish notification 1-2 represents the finish of substantial MT1-2, and it is issued by the ExitMT namely MT2-5 within MT1-2.

2.5 Macrotask Scheduling

In macrotask scheduling using the layer-unified execution control, each macrotask of all layers is added to a ready macrotask queue when its earliest executable condition, as shown in Table 1, is satisfied. After that, the dynamic scheduler retrieves a macrotask from the ready macrotask queue in order of priority, namely the critical path (CP) length, and assigns the macrotask to its own core. Therefore, the layer-unified execution control can execute macrotasks of all layers on cores without grouping as shown in Fig. 3 (a).

As to the deployment of the dynamic scheduling code, the distributed scheduling policy is adopted, where each core executes a thread code including both a scheduling processing part and a macrotask processing part. The dynamic scheduler within the thread code works on a core as follows:

(i) Add macrotasks that satisfy their earliest executable conditions to a ready macrotask queue;
(ii) Retrieve a macrotask with the longest CP length from the ready macrotask queue and assign the macrotask to its own core;
(iii) Execute the macrotask on its own core;
(iv) Return to (i) and repeat this process until EndMT finishes.

3. Parallel Java Code Generation

To realize the layer-unified coarse grain task parallel processing on a multicore processor, thus far, we need to write a parallel code with the dynamic scheduling code manually. This is a difficult and error-prone work for a user.

Therefore, this paper focuses on the platform-independent Java language and proposes a generation scheme of parallel code written in Java. The parallel Java code is generated from a Java program annotated with parallelization directives by the newly developed parallelizing compiler automatically. According to this approach, layer-unified coarse grain task parallel processing can be realized on various parallel platforms easily. This section describes the structure of parallel Java code and its data structure.

3.1 Structure of Parallel Java Code

This subsection explains the structure of a parallel Java code by using a sample Java program shown in Fig. 1, which corresponds to the hierarchical MTG shown in Fig. 2. In this case, a proposed parallel Java code for layer-unified coarse grain task parallel processing is expressed as shown in **Fig. 4**.

In Fig. 4, the parallel Java code consists of four parts as follows: (1) the variable management classes such as `VARmanage`*i* to manage shared variables within MTG*i*, (2) the macrotask management classes such as `MTqueue`, `MTtable` and `MTmanage` to manage macrotasks in dynamic scheduling, (3) the user-defined class such as `Other`, and (4) the `Mainp` class that includes the `Scheduler` class and the `main()` method.

```
 1: class VARmanage1 {          //variable management of MTG1
 2:   declare shared-variables among macrotasks;
 3: }
 4: class VARmanage2 {          //variable management of MTG2
 5:   declare shared-variables among macrotasks;
 6: }
 7: class VARmanage3 {          //variable management of MTG3
 8:   declare shared-variables among macrotasks;
 9:   declare argument-variables;              //if necessary
10:   declare a return-value-variable;         //if necessary
11: }
12: class MTqueue {                          //macrotask queue
13:   declare a ready-macrotask-queue;
14: }
15: class MTtable {                          //macrotask table
16:   declare a finish-management-table;
17:   declare a branch-management-table;
18:   declare a ready-management-table;
19: }
20: class MTmanage {                  //macrotask management
21:   ArrayList<MTtable> dynamicfield
22:                    = new ArrayList<MTtable>(); //dynamic
23:   MTtable            staticfield;               //static
24: }
25: class Other {                         //user-defined class
26:   static void mt3_0() { ... }     //layer-start macrotask
27:   static void mt3_1() { ... }                   //MT3-1
28:   static void mt3_2() { ... }                   //MT3-2
29:   ...
30: }
31: class Mainp {                   //main parallel Java code
32:   static ArrayList<VARmanage1> varm1
33:                  = new ArrayList<VARmanage1>(); //dynamic
34:   or static VARmanage1 varm1 = new VARmanage1(); //static
35:   declare varm2;                      //dynamic or static
36:   declare varm3;                      //dynamic or static
37:   static MTqueue mtqueue = new MTqueue();
38:   static MTmanage[] mtm = new MTmanage[3];   //three MTGs
39:   static class Scheduler implements Runnable {
40:     int threadid;
41:     Scheduler(int thrid) { threadid = thrid; }
42:     public void run()    { scheduler(threadid); }
43:   }
44:   static boolean eeccheck(int mt) {
45:     check the earliest-executable-condition of macrotask;
46:   }
47:   static void scheduler(int threadid) {
48:     append ready-macrotasks to ready-macrotask-queue;
49:     while (EndMT does not finish) {
50:       retrieve MTi from ready-macrotask-queue;
51:       execute MTi and notify the finish of MTi;
52:       append ready-macrotasks to ready-macrotask-queue;
53:     }
54:   }
55:   static void mt1_1() { ... }                   //MT1-1
56:   static void mt1_2() { ... }                   //MT1-2
57:   static void mt1_3() { ... }                   //MT1-3
58:   static void mt2_1() { ... }                   //MT2-1
59:   static void mt2_2() { Other.mt3_0(); }        //MT2-2
60:   ...
61:   public static void main(String[] args) {
62:     ...
63:     create threads by using Scheduler();
64:     join threads;
65:   }
66: }
```

Fig. 4 Structure of parallel Java code.

Each macrotask processing code is implemented as a class method within the `Mainp` class, such as `mt1_1()`. On the other hand, a macrotask processing code within the class defined by a user is implemented as a class method within the user-defined class, such as `mt3_1()` within the `Other` class. Several user-defined classes may exist in a target Java program.

3.2 Dynamic Scheduling Code

In this scheme, to enhance the core utilization factor, the dynamic scheduler with layer-unified execution control is implemented in a distributed scheduler manner. Each core executes a thread code composed of both a scheduling processing part and a macrotask processing part. The multi-threaded codes are written by the Runnable interface in the Java language.

If the ready macrotask queue for dynamic scheduling is empty at scheduling time, the dynamic scheduler enters into the wait set. Conversely, if a new executable macrotask is appended to the ready macrotask queue by the finish of a data dependence predecessor macrotask, a thread within the wait set turns to the execution state.

In Fig. 4, the dynamic scheduling code is composed of several parts as follows: the `MTqueue` class at lines 12–14, the `MTtable` class at lines 15–19, the `MTmanage` class at lines 20–24, the variable `mtqueue` at line 37, the variable `mtm` at line 38, the `Scheduler` class at lines 39–43, the `eeccheck()` method at lines 44–46 which performs the earliest executable condition check as mentioned in Section 2.4, and the `scheduler()` method at lines 47–54 which performs the scheduling processing as mentioned in Section 2.5.

In the `main()` method at line 63 in Fig. 4, a thread per core is created by using the constructor `Scheduler()` once at start time. After that, the thread performs the scheduling processing and the macrotask processing with a low overhead. In each thread, when the execution of a macrotask has finished on a core binding to the thread, the core performs scheduling by using the scheduling processing code and executes a newly assigned macrotask. To access the ready macrotask queue, mutual exclusion using the synchronized block from the Java language is performed.

3.3 Dynamic Data Structure

For shared variables within a MTG and macrotask's execution control data, the parallel Java code adopts two kinds of management methods selectively. One is the dynamic data structure policy and the other is the static data structure policy. The former is described in this subsection and the latter is described in Section 3.4.

First, variables within a MTG are categorized as shared variables, argument variables or a return variable. A set of these variables is treated as `VARmanage`*i* class that corresponds to MTG*i* as shown at lines 1–11 in Fig. 4. In the case of MTG1 in Fig. 2, the corresponding `VARmanage1` class is defined at lines 1–3 in Fig. 4 and its instance variable is declared as `varm1` at lines 32–33.

Basically, if a MTG is executed only once, an instance of VARmanage corresponding to the MTG needs to be created. Conversely, if a MTG that corresponds to a method body could be invoked simultaneously by several method invocations on different threads, several instances of VARmanage must be created. Thus, as shown at lines 32–33, instances of VARmanage are managed by an ArrayList-type variable. For example, when a macrotask invokes a method to be parallelizaed, the macrotask is treated as a layer-start macrotask mentioned in Section 2.3 and it creates an instance of `VARmanage`*i* that corresponds to the method. The created instance is appended to the ArrayList-type variable `varm`*i*. In this paper, such an implementation is called the dynamic data structure policy.

Note that apart from shared variables within a MTG, there are global variables (static variables within user-defined class) and local variables within a macrotask. In the parallel Java code, global

```
 1: class Other {
 2:   static int v[]=new int[2];
 3:   static void method(int id) {
 4:     int s;
 5:     int i;
 6:     /*mt*/ {
 7:     s=0;
 8:     i=0;
 9:     }
10:     /*mt*/ {
11:     while (i<4) {
12:       s+=(id+1);
13:       i++;
14:     }
15:     }
16:     /*mt*/ {
17:     v[id]=s;
18:     }
19:   }
20: }
21:
22: class Main {
23:   public static void main(String[] args) {
24:     /*mt*/{
25:     Other.method(0);
26:     }
27:     /*mt*/{
28:     Other.method(1);
29:     }
30:   }
31: }
```

Fig. 5 Java program with two method invocations.

variables are implemented in the same way as the original source program.

Secondly, as to the management of the macrotask's execution control data, the parallel Java code adopts the `MTmanage` class including instances of the `MTtable` class as shown at lines 15–24 in Fig. 4. Instances of the `MTmanage` are declared as an array that has elements equivalent to the number of MTGs at line 38 in Fig. 4.

In the `MTmanage` class, an ArrayList-type variable `dynamicfield` is used at lines 21–22 in Fig. 4. An instance of MTtable corresponding to a MTG is dynamically appended to `dynamicfield` by the layer-start macrotask corresponding to the MTG.

As an example of method invocations, we show a sample Java program in **Fig. 5**. In Fig. 5, we assume `Other.method(0)` and `Other.method(1)` within the `main()` method are invoked in parallel. This sample Java program is compiled into a parallel Java code with a dynamic data structure by our compiler as shown in **Fig. 6**. Several shared variables among macrotasks within `Other.method()` in Fig. 5 are defined as the VARmanage1 class in Fig. 6. At runtime, when `mt2_1()` including `Other.m1(0)` was executed, `Other.m1()` method (the layer-start macrotask) adds an instance of VARmanage1 to the ArrayList-type variable `varm1`. On the other hand, when `mt2_2()` including `Other.m1(1)` was executed, `Other.m1()` method (the layer-start macrotask) adds another instance of VARmanage1 to the ArrayList-type variable `varm1`. This is why these instances are managed by ArrayList-type where the identifier `__ec` for the instance is introduced. By using this identifier `__ec`, macrotasks such as `mt1_1()`, `mt1_2()` and `mt1_3()` can access to variables within the appropriate instance.

3.4 Selective Static Data Structure

In the parallel Java code, as mentioned in Section 3.3, the dynamic data structure policy is used by default. However, the access cost to an instance pointed by an element of ArrayList-type variable is larger than the access cost to an instance pointed by a scalar variable.

To reduce such access cost via ArrayList-type, this paper

```
 1: class VARmanage1 {
 2:   int id;
 3:   int s;
 4:   int i;
 5: }
 6:
 7: class Other {
 8:   static int[] v = (new int[2]);
 9:   static void m1(int id) {
10:     int __ec;
11:     VARmanage1 __t_varm1 = new VARmanage1();
12:     __t_varm1.id = id;
13:     synchronized(Mainp.varm1) {
14:       Mainp.varm1.add(__t_varm1);
15:       __ec = Mainp.varm1.size()-1;
16:     }
17:     ...
18:   }
19:   static void mt1_1(int __ec) {
20:     Mainp.varm1.get(__ec).s = 0;
21:     Mainp.varm1.get(__ec).i = 0;
22:   }
23:   static void mt1_2(int __ec) {
24:     while (Mainp.varm1.get(__ec).i < 4) {
25:       Mainp.varm1.get(__ec).s +=
          (Mainp.varm1.get(__ec).id + 1));
26:       Mainp.varm1.get(__ec).i++;
27:     }
28:   }
29:   static void mt1_3(int __ec) {
30:     v[Mainp.varm1.get(__ec).id] = Mainp.varm1.get(__ec).s;
31:   }
32: }
33:
34: class Mainp {
35:   static ArrayList<VARmanage1> varm1 =
      new ArrayList<VARmanage1>();
36:   ...
37:   static void mt2_1(int __ec) {
38:     Other.m1(0);
39:   }
40:   static void mt2_2(int __ec) {
41:     Other.m1(1);
42:   }
43:   public static void main(String[] args) { ... }
44: }
```

Fig. 6 Parallel Java code with dynamic data structure.

```
 1: class VARmanage1 {
 2:   int id;
 3:   int s;
 4:   int i;
 5: }
 6:
 7: class Other {
 8:   static int[] v = (new int[2]);
      ...
19:   static void mt1_1() {
20:     Mainp.varm1.s = 0;
21:     Mainp.varm1.i = 0;
22:   }
23:   static void mt1_2() {
24:     while (Mainp.varm1.i < 4) {
25:       Mainp.varm1.s += (Mainp.varm1.id + 1);
26:       Mainp.varm1.i++;
27:     }
28:   }
29:   static void mt1_3() {
30:     v[Mainp.varm1.id] = Mainp.varm1.s;
31:   }
32: }
33:
34: class Mainp {
35:   static VARmanage1 varm1 = new VARmanage1();
      ...
44: }
```

Fig. 7 Parallel Java code with static data structure.

adopts the static data structure policy selectively unless macrotasks within a MTG are executed in parallel simultaneously on different threads. The static data structure is applied to the MTGs annotated by the parallelization directive `/*mt stmt*/` as described in Section 4.2. Namely, the user can select the target MTG to be implemented by means of the static data structure.

Regarding a management of shared variables within a MTG based on the static data structure policy, the scalar variable `varm`*i* at line 34 in Fig. 4 is used instead of the ArrayList-type variable `varm`*i* at lines 32–33. Regarding a management of macrotask's execution control data, a scalar variable `staticfield` at line 23 is used instead of the ArrayList-type variable `dynamicfield` at line 21-22.

In Fig. 5 mentioned above, even if `Other.method(0)` and `Other.method(1)` are executed in serial, our compiler can generate the parallel Java code using static data structure as shown

in **Fig. 7**. In this case, only one instance of VARmanage1 is created as a scalar variable `varm1` and macrotasks such as `mt1_1()`, `mt1_2()` and `mt1_3()` can access to variables within the instance.

4. Parallelizing Compiler

This section describes the newly developed parallelizing compiler for layer-unified coarse grain task parallel processing.

4.1 Specification of Parallelizing Compiler

The layer-unified coarse grain task parallel processing is advantageous in terms of the use of parallelism. However, it is not easy for a user to generate the above-mentioned parallel Java code manually. Therefore, we have developed a parallelizing compiler as a prototype.

The parallelizing compiler reads a Java program with parallelization directives as mentioned in Section 4.2 as a source file, and generates a parallel Java code to perform layer-unified coarse grain task parallel processing. When a Java program with parallelization directives is composed of several files, their files need to be concatenated into one file. However, source files without parallelization directives and class files can be treated as independent files.

The target source program for this compiler needs to be written in the JDK1.2 grammar because of its front-end implementation. Also, the exception code such as the try-catch block is executed in serial. However, such constraints are not a major problem in parallel processing of general scientific computing programs.

4.2 Parallelization Directives

To perform layer-unified coarse grain task parallel processing, it is necessary to insert the parallelization directives listed in **Table 2** into a target Java program. Then the proposed parallelizing compiler generates a parallel Java code.

For example, the program shown in Fig. 1, whose MTG corresponds to Fig. 2, is a source program with parallelization directives. To define a macrotask as mentioned in Section 2.2, the parallelization directive `/*mt*/` is used. The only directive `/*mt*/` is indispensable for parallelization. Macrotasks can be defined hierarchically by nesting `/*mt*/` within a repetition block (e.g., a for-statement, a while-statement) or within a class method. Non-parallelizable regions such as preprocessing and postprocessing should be denoted by parallelization directives such as `/*premt*/` and `/*postmt*/`.

Optionally, several parallelization directives to enhance performance are prepared as follows. The directive `/*mt decomp=`*value*`*/` specifies the number of decomposition of a parallelizable loop. The directive `/*mt stmn*/` applies the selective static data structure management to a MTG including its macrotask. The directive `/*mt cp=`*value*`*/` indicates the critical path (CP) length of a macrotask. The directive `/*mt` *logical-expression*`*/` indicates the earliest executable condition of a macrotask, where the *logical-expression* is written by the MTG number, the macrotask number, the symbol '`&`' meaning AND, the symbol '`|`' meaning OR and the symbol '`*`' meaning branch. Note that such an earliest executable condition can by analyzed within a method automatically, but its directive is effective for exploiting parallelism among methods.

Table 2 Parallelization directives.

Denotation	Definition
`/*mt*/`	a macrotask
`/*premt*/`	a preprocessing part
`/*postmt*/`	a postprocessing part
`/*mt decomp=`*value*`*/`	the number of loop decomposition
`/*mt stmn*/`	the static data structure management
`/*mt cp=`*value*`*/`	the critical path length
`/*mt` *logical-expression*`*/`	the earliest executable condition

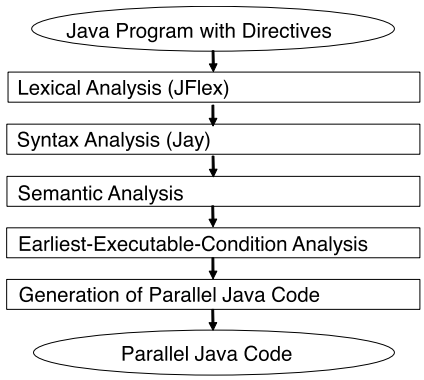

Fig. 8 Overview of parallelizing compiler.

4.3 Implementation of Parallelizing Compiler

This subsection describes an implementation of the parallelizing compiler to generate a parallel Java code. The structure of the compiler is shown in **Fig. 8**. The compiler is developed by using the Java language, whose lexical analysis and syntax analysis are implemented by the Jay/JFlex parser generator that deals with the LALR(1) grammar.

The compiler recognizes macrotasks denoted by the parallelization directives in Table 2, analyzes data dependence and control dependence, analyzes the earliest executable condition as shown in Table 1, and generates a parallel Java code (e.g., `Mainp.java`) including a dynamic scheduler that reflects the earliest executable condition.

In this prototype compiler, the data dependence related to the primitive type can be analyzed automatically, but the data dependence related to the reference type is preserved in consideration of the alias problem. The earliest executable condition of macrotasks within a method can be generated automatically, but a strong inter-procedural analysis is not implemented. Thus, to effectively extract inter-method parallelism, a user can partially denote the earliest executable condition as a parallelization directive.

5. Performance Evaluation on a Multicore System

This section demonstrates the performance evaluation using compiler-generated parallel Java codes on the multicore system DELL PowerEdge R620.

5.1 Environment of Performance Evaluation

In the evaluation on the DELL PowerEdge R620, Intel Xeon E5-2660, namely eight cores of 2.20 GHz, is used. The speci-

Table 3 Performance evaluation programs.

Property	Jacobi	Integral	Turb3d	Crypt	MolDyn
Code length	99	27	2,290	300	557
Number of inserted parallelization directives	9	2	68	4	24
Maximum depth of MTG's layer	4	1	7	1	4
Number of MTGs	5	1	21	1	4
Number of macrotasks after loop decomposition	143	36	470	35	266
Number of executed macrotasks	1,382	43	3,009	35	12,903
Sequential execution time on Xeon E5-2660 without Hotspot optimization	43.255 s	4.167 s	195.722 s	—	—
Sequential execution time on Xeon E5-2660 with Hotspot optimization	5.522 s	1.020 s	10.165 s	2.809 s	27.600 s

Table 4 Access to variables shared among macrotasks.

Property	Jacobi	Integral	Crypt	MolDyn
Number of accesses to global variables	2.50×10^9	5.00×10^7	8.50×10^8	1.55×10^{10}
Number of accesses to shared variables (dynamic or static)	20	1.50×10^8	0	6.50×10^3
Ratio of shared var. to both shared var. and global var.	less than 1%	75%	0%	less than 1%

fication is as follows: 64 GB of memory; 32 KB/core of L1 instruction cache; 32 KB/core of L1 data cache; 512 KB/core of L2 cache; 20 MB of L3 cache; the operating system (OS) is CentOS6.5; and the Java system is JDK1.7.

In the performance evaluation, five programs are used as shown in **Table 3**. This table also indicates the programs' properties, such as the code length, the number of inserted parallelization directives, the maximum depth of MTG's layer, the number of MTGs, the number of macrotasks after loop decomposition, the number of executed macrotasks, the sequential execution time on JVM of Xeon E5-2660 without/with Hotspot (i.e., Just-In-Time compilation) optimization. Sequential execution means the execution of an original Java program without our parallelization directives on one core. The execution time presented in this paper is expressed as an average of the middle three data within five measurements.

5.2 Evaluation of the Parallel Java Code with Selective Static Data Structure

In this subsection, by using two scientific computing programs written in Java, we evaluate the performance of a parallel Java code with selective static data structure.

The parallel processing scheme is based on layer-unified coarse grain task parallel processing (namely layer-unified parallelization), and our compiler generates two types of parallel Java codes having different data structures, as follows: (1) ordinary dynamic data structure, (2) selective static data structure. Each parallelizable loop is decomposed into 40 partial loops by the parallelization directive. The compiler-generated parallel Java code is compiled into the Java bytecode by the Java compiler (javac), and the Java bytecode is executed on JVM of Xeon E5-2660.

First, the Jacobi program to solve the system of linear equations has properties as shown in Table 3. The code length is 99 lines, and a main convergent loop includes three method invocations each of which is composed of for-statements. The size of the matrix is 10,000*10,000. **Figure 9** and **Fig. 10** show the speed-up ratio relative to one thread execution on JVM of Xeon E5-2660 without/with Hotspot optimization respectively.

As shown in Fig. 9, in ordinary layer-unified coarse grain task parallel processing with dynamic data structure on eight threads (eight cores), the execution time is 7.53 times faster than ordinary

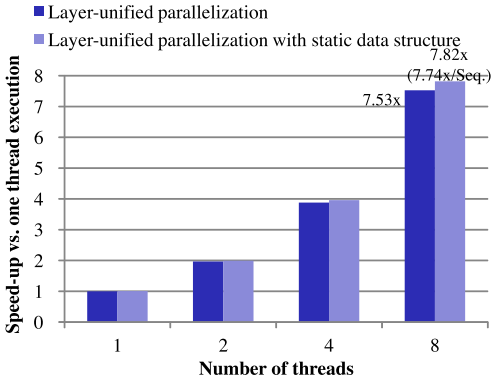

Fig. 9 Jacobi program in layer-unified coarse grain task parallel processing on Xeon E5-2660 without Hotspot optimization.

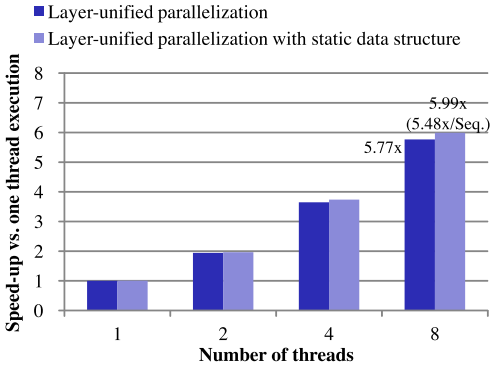

Fig. 10 Jacobi program in layer-unified coarse grain task parallel processing on Xeon E5-2660 with Hotspot optimization.

one thread execution with dynamic data structure. Additionally, when the selective static data structure is applied, the execution time is 7.82 times faster than one thread execution (i.e., 7.74 times faster than sequential execution). The selective static data structure reduced the parallel processing time by 3.7%. As the Jacobi program originally declares the matrix as global variable (static variable within the user-defined class), the number of accesses to shared variables to be subjected to static data structure is small as shown in **Table 4**. On the contrary, as the number of executed macrotasks is 1,382 in Table 3, the dynamic scheduling overhead is reduced by static data structure. In the case of JVM execution with Hotspot optimization in Fig. 10, the speed-up ratio of the proposed parallel Java code on eight threads can achieve 5.99 times versus one thread execution (i.e., 5.48 times speed-up ver-

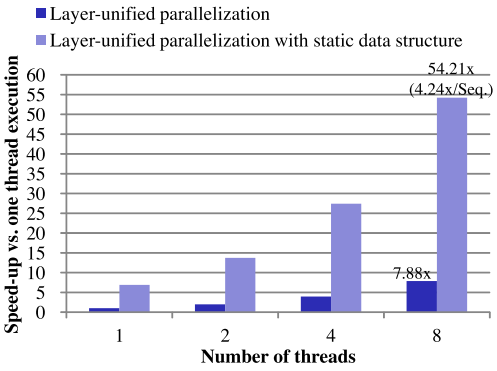

Fig. 11 Integral program in layer-unified coarse grain task parallel processing on Xeon E5-2660 without Hotspot optimization.

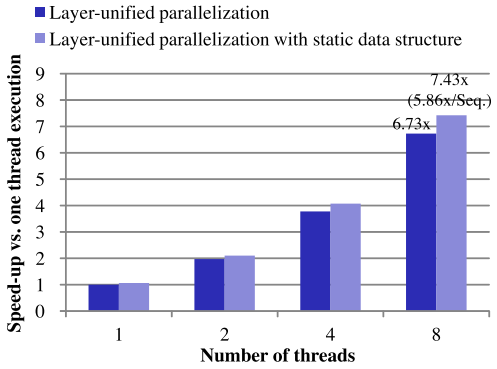

Fig. 12 Integral program in layer-unified coarse grain task parallel processing on Xeon E5-2660 with Hotspot optimization.

sus sequential execution).

Next, the Integral program with trapezoidal rule that solves Pi is categorized in Table 3. The length of the source code is 27 lines, and this program computes the definite integral of the function $4/(1+x^2)$ from $x = 0$ to 1. The number of divisions of the integration interval is 50 million. The parallel execution results on JVM of Xeon E5-2660 are shown in **Fig. 11** and **Fig. 12**.

As can be seen from Fig. 11, layer-unified coarse grain task parallel processing on eight threads can achieve 7.88 times faster speed-up versus one thread execution. Moreover, when the selective static data structure is applied, we achieved 54.21 times faster speed-up versus one thread execution (i.e., 4.24 times speed-up versus sequential execution). Namely, the selective static data structure reduced execution time by 85.5% compared to ordinary layer-unified coarse grain task parallel processing. This extreme performance improvement mainly results from the static data structure for shared variables among macrotasks. In this case, the ratio of access to shared variables against all variables except local variables is 75% as shown in Table 4 and all shared variables are managed as the static data structure. In the case of JVM execution with Hotspot optimization in Fig. 12, the proposed parallel Java code can achieve 7.43 times faster speed-up versus one thread execution (i.e., 5.86 times speed-up versus sequential execution). Since the code length of this program is short, the overhead due to Just-In-Time compilation is small.

As a result, the parallel Java code with the selective static data structure was confirmed to achieve good performance in the environment both without/with Hotspot optimization.

Table 5 Comparison of execution scheme for Turb3d program on Xeon E5-2660 without Hotspot optimization.

Execution scheme	1 thread	2 threads	4 threads	8 threads
Sequential execution	195.722s	—	—	—
Pseudo loop para.	250.360 s	142.770 s	78.939 s	45.926 s
Layer-unified para.	249.277 s	136.046 s	73.573 s	39.202 s
Improvement by layer-unified para.	—	4.94%	7.29%	17.2%

5.3 Evaluation of the Parallel Java Code to Extract Coarse Grain Task Parallelism

This subsection presents the parallel execution results obtained by using the Turb3d program from the SPECfp95 benchmark suite. The original program, written in Fortran, simulates turbulence by solving the Navier-Stokes equation. The program consists of 21 subroutines and one main routine. The main routine is composed of iterative loops that call appropriate subroutines by inner conditional branches. In such a program, it is difficult for conventional loop parallelization to utilize parallelism effectively because this program does not have sufficient loop parallelism (inter-iteration parallelism). In other words, such a program is expected to improve the performance by coarse grain task parallelization.

We prepared a Java-converted Turb3d program in advance by means of f2j tool [14]. The f2j tool converts a subroutine in Fortran to a class including a method in Java. We concatenate a set of separate Java files into one file and append the Complex class to it, because the f2j cannot generate a Java class that corresponds to the Fortran COMPLEX type. After that, for a Java program with a code length of 2,290 lines, we annotate 68 parallelization directives and then our parallelizing compiler generates a parallel Java code. A parallelizable loop with the parallelization directive is decomposed into 32 partial loops, each of which is defined as a macrotask. The code is compiled by javac and it is executed on JVM of Xeon E5-2660.

First, we compare ordinary layer-unified coarse grain task parallel processing and pseudo loop parallelization in terms of parallel execution time. Pseudo loop parallelization is the restricted execution mode that exploits only loop parallelism in the environment of layer-unified coarse grain task parallel processing. As shown in **Table 5**, layer-unified coarse grain task parallel processing could exploit inter-macrotask parallelism in addition to loop parallelism. Thus, it could reduce the parallel execution time against pseudo loop parallelization by 17.2% on eight threads (eight cores). This result indicates that layer-unified coarse grain task parallel processing is superior to loop parallelization. Table 5 also represents the sequential execution time without layer-unified coarse grain task parallel processing.

Secondly, according to the increase of the number of cores, the speed-up ratio due to layer-unified coarse grain task parallel processing on JVM of Xeon E5-2660 is shown in **Fig. 13**. In the execution on eight threads (eight cores), ordinary layer-unified coarse grain task parallel processing with dynamic data structure resulted in 6.36 times faster speed-up versus one thread execution. Furthermore, layer-unified coarse grain task parallel processing with selective static data structure resulted in 7.38 times faster speed-up versus one thread execution (i.e., 5.80 times faster

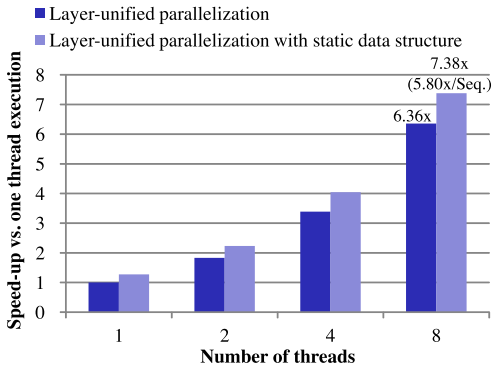

Fig. 13 Turb3d program in layer-unified coarse grain task parallel processing on Xeon E5-2660 without Hotspot optimization.

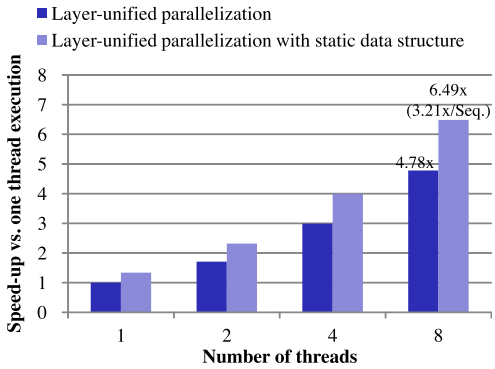

Fig. 14 Turb3d program in layer-unified coarse grain task parallel processing on Xeon E5-2660 with Hotspot optimization.

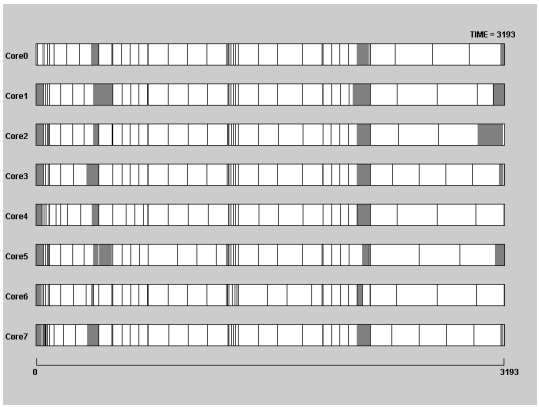

Fig. 15 Partial execution trace of Turb3d program with selective static data structure by eight threads on Xeon E5-2660 without Hotspot optimization.

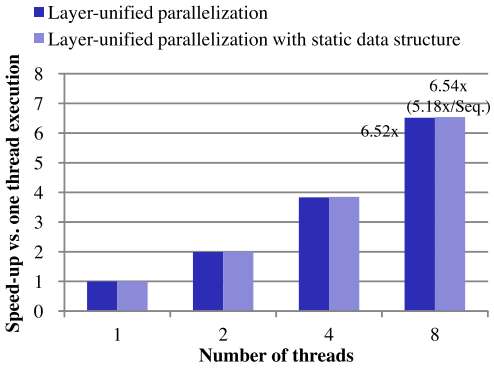

Fig. 16 Crypt program in layer-unified coarse grain task parallel processing on Xeon E5-2660 with Hotspot optimization.

speed-up versus sequential execution). This execution can reduce the parallel execution time by 13.9% compared to ordinary layer-unified coarse grain task parallel processing.

Thirdly, in the case of JVM execution with Hotspot optimization as shown in **Fig. 14**, the proposed scheme on eight threads achieved 6.49 times faster speed-up versus one thread execution (i.e., 3.21 times faster speed-up versus sequential execution). Note that the sequential execution time is listed in Table 3. Although this speed-up ratio is suppressed by the Just-In-Time compilation overhead owing to the long code of Turb3d program, the layer-unified coarse grain task parallel processing gave us such a speed-up.

Finally, we show the execution trace of layer-unified coarse grain task parallel processing with selective static data structure on eight threads (eight cores) without Hotspot optimization as **Fig. 15**. The trace represents the execution of macrotasks within the first iteration of an outermost loop whose execution time is 3.193 [s]. Here, the horizontal axis means the elapsed time; a white part represents execution of a macrotask; a black part represents idle state of the core; the idle state of the core results from either the synchronization wait due to inter-macrotask dependence or the processing of dynamic scheduling. As you can see from this trace, inter-macrotask parallelism is exploited sufficiently and the overhead is small. Note that this trace includes to a small degree the overhead to measure the start time and the finish time of each macrotask.

As a result, layer-unified coarse grain task parallel processing using the proposed compiler-generated parallel Java code can utilize enough coarse grain parallelism in addition to loop parallelism. Furthermore, the adoption of selective static data structure could reduce data access overhead remarkably. Therefore, the effectiveness of the proposed scheme was confirmed.

5.4 Evaluation Using Java Grande Forum Benchmark

This section presents the parallel execution results obtained by using the Crypt program and the MolDyn program from the Java grande forum benchmark suite version 2.0 [15] as shown in Table 3.

First, the Crypt program performs IDEA (International Data Encryption Algorithm) encryption and decryption on an array of N bytes. In this evaluation, we use the class C (large data set, N = 50,000,000). For this program, we apply program restructuring methods [1] such as the inline expansion and the constant propagation; we merged several program files into a Java file. After that, we inserted four parallelization directives, where the number of loop decomposition is 16.

As shown in **Fig. 16**, in ordinary layer-unified coarse grain task parallel processing with dynamic data structure on eight threads (eight cores) with Hotspot optimization, the execution time is 6.52 times faster than ordinary one thread execution with dynamic data structure. Additionally, when the selective static data structure is applied, the execution time is 6.54 times faster than one thread execution (i.e., 5.18 times faster than sequential execution). The effect of the selective static data structure is small. This slight improvement is performed by the reduction of the dynamic scheduling overhead due to static data structure, because

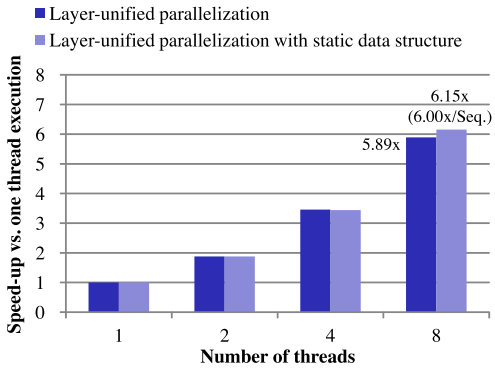

Fig. 17 MolDyn program in layer-unified coarse grain task parallel processing on Xeon E5-2660 with Hotspot optimization.

this program has no access to shared variables as shown in Table 4.

Secondly, the MolDyn program is an N-body code modeling particles interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. The number of particles is given by N. In this evaluation, we use the class B (large data set, N = 8,788). For this program, we apply program restructuring methods such as the node splitting, the scalar expansion, the loop distribution and the constant propagation; we merged several program files into a Java file. After that, we inserted four parallelization directives, where the number of loop decomposition is 26.

As shown in **Fig. 17**, in ordinary layer-unified coarse grain task parallel processing with dynamic data structure on eight threads (eight cores) with Hotspot optimization, the execution time is 5.89 times faster than ordinary one thread execution with dynamic data structure. Additionally, when the selective static data structure is applied, the execution time is 6.15 times faster than one thread execution (i.e., 6.00 times faster than sequential execution). The improvement due to the selective static data structure is performed by the reduction of dynamic scheduling overhead whose target macrotask's number is shown in Table 3, because this program has little access to shared variables as shown in Table 4.

6. Related works

With respect to parallelization for Java programs, several researchers have proposed various ideas. Javar [16] is a restructuring compiler for loops and multi-way recursive methods. A transformation technique to eliminate data dependence caused by the container is proposed for general-purpose Java programs [17]. Jrpm [18] uses parallelism among threads by means of run-time support. zJava [19] combines compile time analysis with run time support to extract parallelism among methods. Parallel Java [20] provides the same capabilities as OpenMP/MPI in a pure Java API. HPJava [21] uses an array distribution such as High Performance Fortran (HPF). MPJ Express [22] implements Message Passing Interface (MPI)-like bindings for Java languages.

Habanero-Java [12] is an extension such as complex-number-type to the original Java-based definition of the X10 language for scientific computing. The X10 language focuses on PGAS (Partitioned Global Address Space) with asynchrony and introduces `places` corresponding to processors [23]. In the description of a parallel program, the `map` function for arrays and the `async` statement to manage activities like threads are used. In such denotations, it is difficult to express inter-task parallelism among program's layers (e.g., nested levels, method invocation levels). On the other hand, our approach can easily extract inter-task parallelism among program's layers by putting the parallelization directive `/*mt*/` into a program and generating the parallel Java code automatically.

7. Conclusions

This paper has proposed a scheme of parallel Java code generation to realize layer-unified coarse grain task parallel processing. Additionally, the selective static data structure was adopted into the parallel Java code. The proposed scheme has been implemented as a prototype parallelizing compiler. This compiler could treat a Java program with parallelization directives as a source program and could generate a platform-independent parallel Java code automatically.

According to performance evaluation on eight cores of Xeon E5-2660, the parallel Java code could perform layer-unified coarse grain task parallel processing effectively. The parallel Java code for the Jacobi program achieved 7.82 times faster speed-up versus one thread execution. In the case of Java grande forum benchmark, the Crypt program achieved 6.54 times faster speed-up and the MolDyn program achieved 6.15 times faster speed-up. Therefore, the effectiveness of parallel Java code generation for layer-unified coarse grain task parallel processing was confirmed.

Acknowledgments The authors would like to express their gratitude to Prof. Hironori Kasahara of Waseda University for his valuable advice to this paper.

References

[1] Wolfe, M.: *High performance compilers for parallel computing*, Addison-Wesley Publishing Company (1996).
[2] Eigenmann, R., Hoeflinger, J. and Padua, D.: On the automatic parallelization of the Perfect benchmarks, *IEEE Trans. Parallel and Distributed System*, Vol.9, No.1, pp.5–23 (1998).
[3] Kasahara, H., Obata, M. and Ishizaka, K.: Automatic coarse grain task parallel processing on SMP using OpenMP, *Proc. 13th International Workshop on Languages and Compilers for Parallel Computing* (2000).
[4] Mase, M., Onozaki, Y., Kimura, K. and Kasahara, H.: Parallelizable C and Its Performance on Low Power High Performance Multicore Processors, *Proc. 15th Workshop on Compilers for Parallel Computing* (2010).
[5] Yoshida, A.: An Overlapping Task Assignment Scheme for Hierarchical Coarse Grain Task Parallel Processing, *Journal Concurrency and Computation: Practice and Experience*, Vol.18, No.11, pp.1335–1351 (2006).
[6] Thies, W., Chandrasekhar, V. and Amarasinghe, S.: A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs, *Proc. IEEE/ACM Int. Symposium on Microarchitecture*, pp.356–368 (2007).
[7] Martorell, X., Ayguade, E., Navarro, N., Corbalan, J., Gonzalez, M. and Labarta, J.: Thread Fork/Join techniques for multi-level parallelism exploitation in NUMA multi-processors, *Proc. Int. Conference on Supercomputing*, pp.294–301 (1999).
[8] Brownhill, C.J., Nicolau, A., Novack, S. and Polychronopoulos, C.D.: Achieving multi-level parallellization, *Proc. ISHPC'97*, pp.183–194 (1997).
[9] Hayashi, A., Shimaoka, M., Mikami, H., Mase, M., Wada, Y., Shirako, J., Kimura, K. and Kasahara, H.: OSCAR Parallelizing Compiler and API for Real-time Low Power Heterogeneous Multicores, *Proc. 16th Workshop on Compilers for Parallel Computing* (*CPC 2012*) (2012).
[10] Yoshida, A.: Layer-unified Execution Control Scheme for Coarse

Grain Task Parallel Processing, *IPSJ Journal*, Vol.45, No.12, pp.2732–2740 (2004).
[11] Taboada, G.L., Ramos, S., Expósito, R.R., Touriño, J. and Doallo, R.: Java in the HIgh Performance Computing Arena: Research, Practice and Experience, *Science of Computer Programming*, Vol.78, No.5, pp.425–444 (2011).
[12] Cave, V., Zhao, J., Shirako, J. and Sarkar, V.: Habanero-Java: The New Adventures of Old X10, *Proc. Int. Conference on the Principles and Practice of Programming in Java* (2011).
[13] Kasahara, H., Obata, M. and Ishizaka, K.: Coarse Grain Task Parallel Processing on a Shared Memory Multiprocessor System, *IPSJ Journal*, Vol.42, No.4, pp.910–920 (2001).
[14] Seymour, K. and Dongarra, J.: *User's Guide to f2j Version 0.8* (2007), Innovative Computing Lab., Dept. Computer Science, Univ. Tennessee.
[15] EPCC: The Java Grande Forum Benchmark Suite (2014), available from ⟨http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/⟩.
[16] Bik, A.J.C. and Gannon, D.B.: Javar a prototype Java restructuring compiler, *Concurrency: Practice and Experience*, Vol.9, No.11, pp.1181–1191 (1997).
[17] Wu, P. and Padua, D.: Container on the Parallelization of General-Purpose Java Programs, *Int. J. Parallel Programming*, Vol.28, No.6, pp.589–605 (2000).
[18] Chen, M.K. and Olukotun, K.: The Jrpm System for Dynamically Parallelizing Java Programs, *Proc. ISCA-30*, pp.434–446 (2003).
[19] Chan, B. and Abdelrahman, T.S.: Run-Time Support for the Automatic Parallelization of Java Programs, *J. Supercomputing*, Vol.28, pp.91–117 (2004).
[20] Kaminsky, A.: Parallel Java: A Unified API for Shared Memory and Cluster Parallel Programming in 100% Java, *Proc. IEEE Int. Parallel and Distributed Processing Symposium* (2007).
[21] Lim, S.B., Lee, H., Carpenter, B. and Fox, G.: Runtime support for scalable programming in Java, *J. Supercomputing*, Vol.43, pp.165–182 (2008).
[22] Shafi, A., Manzoor, J., Carpenter, B. and Baker, M.: Multicore-enabling the MPJ EXpress Messaging Library, *Proc. Int. Conference on the Principles and Practice of Programming in Java* (2010).
[23] Saraswat, V., Bloom, B., Peshansky, I., Tardieu, O. and Grove, D.: X10 Language Specification Version 2.4 (2014), available from ⟨http://x10.sourceforge.net/documentation/languagespec/x10-243.pdf⟩.

Akimasa Yoshida received his B.E., M.E. and Dr. Eng. degrees from Waseda University in 1991, 1993 and 1996, respectively. He became a JSPS research fellow (DC1) in 1993, a research associate at Waseda University in 1995, an assistant professor at Toho University in 1997, and an associate professor at Toho University in 2004. He has been engaged as an associate professor at Meiji University since 2013. He joined the green computing system research organization as a visiting associate professor at Waseda University in 2014. His research interests include parallel computing, parallelizing compiler and parallelization of application. He is a member of IEICE, IEEJ, IEEE and ACM.

Yuki Ochi received his B.S. and M.S. degrees from Toho University in 2012 and 2014, respectively. His research interest is parallel computing on multicore processors.

Nagatsugu Yamanouchi received his B.E. and M.E. degrees from the University of Tokyo in 1975 and 1977, respectively. He attended the Graduate School of Stanford University from 1978 to 1984. He joined IBM Tokyo Research Laboratory in 1984. He received Dr. Eng. degree from the University of Tokyo. He moved to Toho University in 2000 as an associate professor, and has been a professor since 2004. His research interests include parallel and distributed computing, computer network, and application of information processing. He is a member of ACM, IEEE, and JSSST.