Journal of Information Processing Vol.25 426–437 (June 2017)

[DOI: 10.2197/ipsjjip.25.426]

Regular Paper

A Task-driven Parallel Code Generation Scheme for Coarse Grain Parallelization on Android Platform

Akimasa Yoshida[1,2,a)] Akira Kamiyama[3] Hiroki Oka[1]

Received: October 24, 2016, Revised: January 5, 2017,
Accepted: February 7, 2017

Abstract: Thanks to high performance and low power consumption on Android mobile devices such as smartphones and tablet computers, the use of Android platform has been increasing significantly. Android platforms almost consist of ARM-based multicores and most of the applications have been developed in Java language. Recent Android OS introduces the Java runtime environment called ART which enables use of Fork/Join framework. The Fork/Join framework provides the scheduling mechanism with work-stealing and it is mainly used for programs to implement the divide-and-conquer algorithm or the recursive algorithm. However, in the case of ordinary programs, it is difficult to implement a coarse-grain parallel code based on Fork/Join framework considering data-dependency. To cope with such a problem, this paper proposes a coarse-grain parallel code generation scheme using a developed compiler which converts a Java source program with directives into a task-driven parallel code based on Fork/Join framework. In the performance evaluation using four programs from Java Grande Forum Benchmark Suite, the execution on Samsung Galaxy S6 with heterogeneous eight cores could achieve 2.77–5.12 times speedup versus sequential processing and the execution on NVIDIA Shield Tablet with four cores could also achieve 2.34–3.94 times speedup. Consequently, effectiveness of the proposed scheme was confirmed.

Keywords: Android, Java Fork/Join, task-driven, coarse-grain parallelization, multicore, compiler

1. Introduction

Modern computers such as supercomputers, servers, smartphones and embedded systems have used the parallel processing techniques with multicores to improve their computation performance. Recently, since mobile devices including IoT devices are more popular, it is expected to further enhance performance of Android mobile devices.

Coarse grain parallelization [1], [2], [3], [4] as well as loop parallelization [5] is expected to provide high performance on multicore processors. Particularly, in order to enhance coarse grain parallelism extraction, the layer-unified execution control scheme [6], [7] has been proposed, which can utilize coarse grain task parallelism among different layers.

Though C and Fortran languages are conventionally used as target languages for parallel computing, interest in Java for high performance computing is recently increasing [8]. 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. In addition, to improve Android device performance, Android 5.0 later adopts the ART runtime that is based on Ahead-Of-Time (AOT) compiler techniques instead of the previous Dalvik runtime.

With respect to the parallel code implementation using Java, the Thread class or the Runnable interface is conventionally used. On the contrary, Java SE 7 and Android 5.0 later adopts Fork/Join framework [9], which can parallelize small-grain tasks with low overhead and is promising as a popular parallel platform. The parallel code implementation based on Fork/Join framework is generally performed for the divide-and-conquer or recursive algorithm, but is difficult to use for exploiting coarse grain parallelism considering data-dependency from various kinds of scientific programs in Fork/Join framework environment.

Therefore, this paper proposes a task-driven execution scheme to realize the coarse grain parallel processing with the layer-unified execution control [6] in Fork/Join framework environment, and develops a parallelizing compiler to generate its parallel code. The compiler-generated parallel code enables coarse grain parallel processing in a task-driven manner on Fork/Join framework. This paper also describes performance evaluation on a smartphone Samsung Galaxy S6, a tablet NVIDIA Shield Tablet and a server equipped with Intel Xeon E5-2680.

The rest of this paper is organized as follows. Section 2 describes an overview of the proposed scheme. Section 3 describes a task-driven execution for coarse grain parallel processing. Section 4 describes a task-driven parallel code based on Fork/Join framework. Section 5 describes a parallelizing compiler to generate a task-driven parallel code. Section 6 evaluates the performance of the coarse grain parallel processing with task-driven execution on multicore systems. Section 7 describes related works. Finally, Section 8 presents our conclusions.

[1] Meiji University, School of Interdisciplinary Mathematical Sciences, Nakano, Tokyo 164–8525, Japan
[2] Waseda University, GCS Research Organization, Shinjuku, Tokyo 162–0042, Japan
[3] SoftBank Corp., Minato, Tokyo 105–0021, Japan
[a)] akimasay@meiji.ac.jp

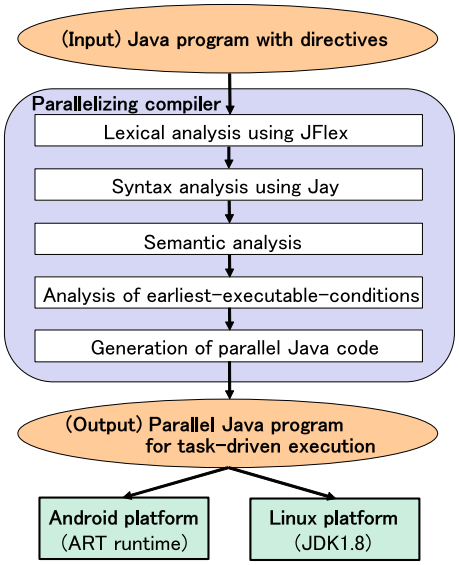


Fig. 1 Flow of task-driven coarse grain parallel processing.

2. Overview of Proposed Scheme

The proposed scheme in this paper consists of (1) Task-driven coarse grain parallel processing scheme which is also called Task-driven execution below, (2) Fork/Join-based implementation of the task-driven execution and (3) Parallelizing compiler for the task-driven execution.

This section briefly explains how to apply task-driven execution to ordinary programs. As shown in **Fig. 1**, a Java program with directives is input to the developed parallelizing compiler, and the compiler generates a parallel Java program for task-driven execution. Namely, the compiler works as a source-to-source translator.

Next, the generated parallel Java program is executed on Android platform with ART runtime or on Linux platform with JVM (Java virtual machine). In other words, the generated parallel Java program can be adapted to several platforms. In addition, the class files corresponding to the generated parallel Java program may be archived into a library like a JAR file. Therefore, this proposed scheme is considered to be useful for application developers, library developers and general programmers.

3. Task-driven Execution for Coarse Grain Parallel Processing

This section describes the task-driven execution scheme to realize coarse grain parallel processing in Fork/Join framework environment.

3.1 Concept of Task-driven Execution

In this subsection, we explain the concept of the proposed task-driven execution by using a sample program as shown in **Fig. 2**. To realize the task-driven execution, the sample program is converted to a macro-task-graph (MTG) depicted in **Fig. 3** by the parallelizing compiler. The MTG represents the earliest executable conditions of coarse-grain tasks (called macrotasks) as indicated in **Table 1**.

As seen from **Fig. 4**, an execution image on four cores, namely

```
class Other {
  public static void func() {
    /*mt fork*/ {                          MTG2
    macrotask code;                  //MT21
    }
    /*mt fork*/ {
    macrotask code;                  //MT22
    }
  }
}

public class Main {
  public static void main(String[] args) {
    /*mt fork*/ {                          MTG0
    macrotask code;                  //MT1
    }
    ...
    /*mt fork inner*/ {
    for (int i=0; i<2; i++) {        //MT6
      /*mt fork*/ {                        MTG1
      macrotask code;                //MT11
      }
      /*mt fork*/ {
      macrotask code;                //MT12
      }
    }
    }
    ...
    /*mt fork inner*/ {
    Other.func();                    //MT8
    }
    ...
  }
}
```

Fig. 2 A source program with parallelization directives.

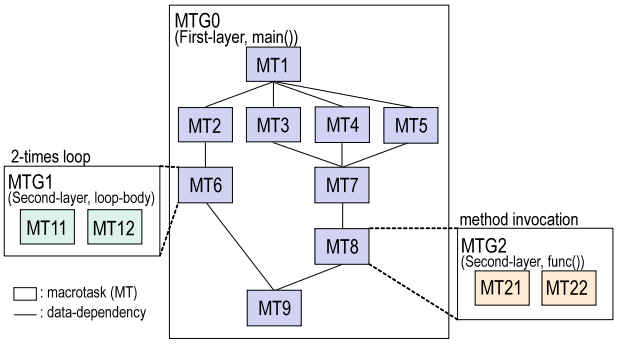


Fig. 3 Macro-task-graph (MTG).

four worker-threads on Fork/Join framework, shows that macrotasks are executed on four cores effectively. When MT1 in Fig. 4 finishes its execution on Core0, a worker-thread binding to Core0 tries to check the earliest executable conditions for the task-driven succeeding macrotask-candidates such as MT2, MT3, MT4 and MT5 in Table 1, and forks the executable macrotasks to its own worker-queue. Then, each worker-thread binding to a core starts to execute a macrotask in Fork/Join framework with work-stealing.

In code generation of the task-driven execution, the first phase detects the inter-macrotask parallelism according to the layer-unified coarse grain parallelization [6], [7], which defines macrotasks hierarchically, analyzes the earliest executable conditions in Table 1 and represents them as a macro-task-graph (MTG) [6], [10]. The second phase generates the task-driven parallel code that can execute the macrotasks on multicores in a dynamic scheduling manner. Here, the management of the macrotask state such as Finish or Branch is performed by the task-driven parallel code generated by the compiler, but Fork/Join framework is responsible for scheduling to take an executable macrotask

Table 1 Earliest executable conditions and task-driven succeeding macrotask-candidates.

MTG	MT	Earliest Executable Condition	Finish/ Branch Notification	Task-driven Succeeding MT-candidates
0	1	true	1	2,3,4,5
	2	1	2	6
	3	1	3	7
	4	1	4	7
	5	1	5	7
	6†	2	6S	10
	7	3∧4∧5	7	8
	8†	7	8S	21,22
	9	6∧8	9	End
	End‡	9	—	—
1	10‡(Loop)	6S	10	11,12
	11	10	11	13
	12	10	12	13
	13‡(Ctrl)	11∧12	13_{14}, 13_{15}	14,15
	14‡(Repeat)	13_{14}	14	10
	15‡(Exit)	13_{15}	6	9
2	21	8S	21	23
	22	8S	22	23
	23‡(Exit)	21∧22	8	9

†: layer-start macrotask
‡: dummy-macrotasks for management
13_{14}: MT13 finishes and branches to MT14
13_{15}: MT13 finishes and branches to MT15

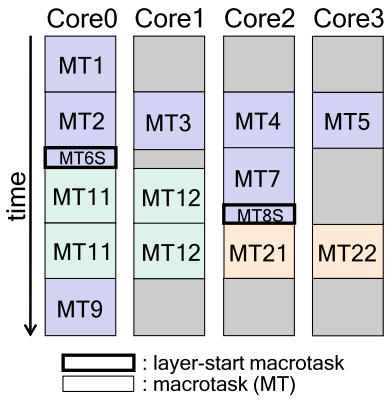


Fig. 4 Task-driven coarse grain parallel processing on 4 cores.

from a worker-queue and execute it on a worker-thread.

3.2 Definition of Macrotask

In coarse grain parallelization [1], [6], an entire program is firstly 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 method). Next, when a first-layer macrotask contains several sub-macrotasks, the sub-macrotasks are defined as second-layer macrotasks. Similarly, macrotasks can be defined hierarchically, but if an upper-layer macrotask has enough parallelism to exhaust given cores, then lower-layer macrotasks within the upper-layer macrotask need not be defined as sub-macrotasks.

In addition, the layer-start macrotask [6] is introduced to uniformly control macrotasks of all layers. As shown in Fig. 3, in the case of a repetition block MT6 that contains MT11 and MT12, MT6 is treated as a layer-start macrotask MT6S in Fig. 4. Also, in the case of a subroutine block MT8 that invokes a method composed of MT21 and MT22, MT8 is treated as a layer-start macrotask MT8S in Fig. 4.

3.3 Earliest Executable Condition and Task-driven Succeeding Macrotask-candidate

After definition of macrotasks, in order to extract inter-macrotask parallelism taking control dependency and data dependency into consideration, the compiler analyzes the earliest executable condition [1], [6]. For example, the earliest executable condition of MT7 in Fig. 3 is expressed by a logical expression 3∧4∧5 in Table 1, which means that MT7 will be executable after MT3, MT4, and MT5 finish.

In the case of a repetition block MT6 in Fig. 3, when MT6 completes the role of a layer-start macrotask MT6S, MT6 issues the Finish notification represented as 6S in Table 1. The notification 6S enables executing MT10 (dummy-macrotask for Loop) within MTG1 in Table 1. After execution of MT10, MT10 issues the Finish notification represented as 10 in Table 1 and its succeeding macrotask-candidates such as MT11 and MT12 will be executable.

MT13 (dummy-macrotask for Ctrl) in Table 1 determines whether to repeat the loop body of MT6 including MT11 and MT12 as follows: MT13 compares a loop induction variable with the loop's upper limit; it issues the Branch notification to MT14 (dummy-macrotask for Repeat) like 13_{14} or the Branch notification to MT15 (dummy-macrotask for Exit) like 13_{15}; if this Branch notification can satisfy the earliest executable condition MT14 or MT15, then MT14 or MT15 will be executable. The earliest executable condition of macrotasks is also represented by a macro-task-graph (MTG) [1] in Fig. 3 where dummy-macrotasks are omitted.

In the case of the method invocation MT8 that contains MT21 and MT22, the macrotask MT8 is treated as a layer-start macrotask. Here, MT23 (dummy-macrotask for Exit) in Table 1 corresponds to an exit node of the method.

3.4 Scheduling for Task-driven Execution

Regarding behavior of macrotasks in the proposed task-driven execution, when a certain macrotask executing on a worker-thread finishes, the worker-thread checks whether to satisfy the earliest executable condition for task-driven succeeding macrotask-candidates in Table 1. If a task-driven succeeding macrotask-candidate is executable, the worker-thread forks the macrotask-candidate and then executes a macrotask that is popped from its worker-queue.

To take a simple example composed of four macrotasks in **Fig. 5** (a), the execution process in Fig. 5 (b) is explained as follows: (1) The worker-thread0 executing the macrotask MTa after MTa finishes, pushes its task-driven succeeding macrotask-candidates such as MTb and MTc to the worker-queue0; (2) The worker-thread0 pops the macrotask MTb from the worker-queue0 and executes it; (3) The worker-thread1 takes the macrotask MTc from the worker-queue0 in work-stealing manner and executes it; (4) Even if the finish time of MTc is later than that of MTb, the MTc forks the task-driven succeeding macrotask-candidate MTd and pushes it to the worker-queue1; (5) The worker-thread1 pops the macrotask MTd from the worker-queue1 and executes it.

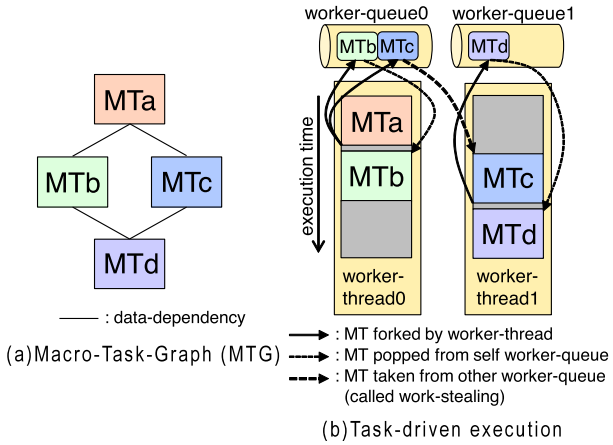


Fig. 5 Task-driven execution using Fork/Join framework.

3.5 Java Fork/Join Framework

This section describes the Java Fork/Join framework [9] which is used for scheduling of the proposed task-driven execution. The Fork/Join framework is a parallel processing framework to implement the ExecutorService interface, which is introduced to Java SE 7, Android 5.0 later. This framework has been designed for small-grain task parallel processing and can cope with the lock-contention problem for large-scale systems by means of the work-stealing mechanism.

Firstly, the Fork/Join framework creates a thread pool including user-specified number of worker-threads. Each worker-thread has a corresponding worker-queue which is maintained as a double-ended queue and it pushes a forked task to a front of the worker-queue in default setting.

When a worker-thread finishes its executing task, it executes a new task which is popped from a front of its worker-queue. Even if the worker-queue is empty, the worker-thread executes a new task which is taken from a back of a worker-queue owned by the other worker-thread in work-stealing manner.

To implement the parallel code, an abstract class `RecursiveAction` or `RecursiveTask` can be utilized. The implementation of the proposed parallel code adopts a `RecursiveAction` that does not treat the return-value because the proposed task-driven parallel code can manage the task state such as Finish or Branch by itself. An abstract method `compute()` within `RecursiveAction` is overridden after class extension.

4. Task-driven Parallel Code Based on Fork/Join Framework

This section describes a generation scheme of task-driven parallel code based on Fork/Join framework for coarse grain parallelization.

4.1 Structure of Task-driven Parallel Code

The task-driven parallel code based on Fork/Join framework can be generated automatically by the parallelizing compiler mentioned below. In general, the form of the parallel code is shown in **Fig. 6** (a). Here, the ForkTemplate-style class is implemented as an inner class and corresponds to a method within a source program. It contains macrotask codes called self-manageable macrotask codes as shown in Fig. 6 (b). The self-manageable macrotask code makes it possible to execute the macrotask, manage the states of the macrotask, and try to fork succeeding macrotask-candidates.

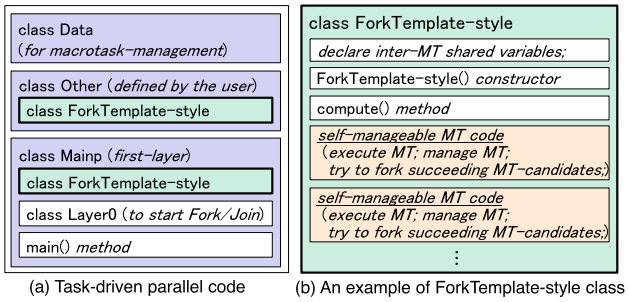


Fig. 6 The structure of the task-driven parallel code.

```
01: class Data  { //for macrotask-management
02:     static ArrayList<ArrayList<RecursiveAction>> forkmt
03:      = new ArrayList<ArrayList<RecursiveAction>>();
04:     static Map<Integer, ArrayList<Integer>> successor
05:      = new HashMap<Integer, ArrayList<Integer>>();
06:     declare macrotask-management-table;
07:     static void initSuccessor() {
08:         set succeeding MT-candidate to successor;
09:     }
10:     static boolean checkEEC(int mt) {
11:         return whether to satisfy Earliest-Executable-Condition;
12:     }
13: }
```

Fig. 7 Data class within a task-driven parallel code.

```
01: class Other { //defined by the user
02:     public static class ForkTemplate_func extends RecursiveAction {
03:         declare shared variables among MTs;
04:         ForkTemplate_func(MT's identifier) {
05:             set MT's identifier to field variables;
06:         }
07:         protected void compute() {
08:             execute corresponding MT-method;
09:         }
10:         public void mt21() { self-manageable MT-code; }
11:         public void mt22() { self-manageable MT-code; }
12:         public void mt23() { //Exit
13:             manage MT;
14:             try to fork succeeding MT-candidates;
15:         }
16:     }
17: }
```

Fig. 8 Other class within a task-driven parallel code.

For example, in the parallel code corresponding to the macrotask-graph shown in Fig. 3, three classes exists as follows: (1) Data class for macrotask-management in **Fig. 7**, (2) Other class that expresses a user-defined class in **Fig. 8** and (3) Mainp class corresponding to first-layer macro-task-graph in **Fig. 9**.

4.1.1 Data Class for Macrotask-management

Data class treats data for macrotask-management. In the code shown in Fig. 7, a variable `forkmt` (at lines 2–3) is used to preserve information of forked macrotasks. A variable `successor` (at lines 4–5) is used to preserve information of succeeding macrotasks. The macrotask-management-table (at line 6) is declared to manage the macrotask state such as Finish or Branch. `initSuccessor()` (at lines 7–9) registers the succeeding macrotask-candidates to the variable `successor`. `checkEEC()` (at lines 10–12) checks whether a macrotask `mt` satisfies its earliest executable condition and returns the result as a boolean-type value.

4.1.2 Other Class Defined by the User

If a source program has user-defined classes including paral-

```
01: class Mainp { //first-layer
02:     public static class ForkTemplate_main extends RecursiveAction {
03:         declare inter-MT shared variables;
04:         ForkTemplate_main(MT's identifier) { //constructor
05:             set MT's identifier to field variables;
06:         }
07:         protected void compute() {
08:             execute corresponding MT-method;
09:         }
10:         public void mtStart() { //mtStart
11:             manage MT; try to fork MT1;
12:         }
13:         public void mt1()  { self-manageable MT-code; }
14:         public void mt2()  { self-manageable MT-code; }
15:         ...
16:         public void mt6()  { //layer-start MT
17:             manage MT; try to fork mt10;
18:         }
19:         public void mt7()  { self-manageable MT-code; }
20:         public void mt8()  { //layer-start MT
21:             manage MT; try to fork mt21, mt22;
22:         }
23:         public void mt9()  { self-manageable MT-code; }
24:         public void mtEnd(){ execute MT; manage MT;  } //End
25:         public void mt10() { //Loop
26:             manage MT; try to fork mt11, mt12;
27:         }
28:         public void mt11() { self-manageable MT-code; }
29:         public void mt12() { self-manageable MT-code; }
30:         public void mt13() { //Ctrl
31:             manage MT; try to fork mt14 or mt15;
32:         }
33:         public void mt14() { //Repeat
34:             manage MT; try to fork mt10;
35:         }
36:         public void mt15() { //Exit
37:             manage MT; try to fork mt9;
38:         }
39:     }
40:     static class Layer0 extends RecursiveAction {
41:         Layer0() { //constructor
42:             initialize field variables within Data;
43:         }
44:         protected void compute() {
45:             fork mtStart;
46:             helpQuiesce() //start task processing
47:             join;
48:         }
49:     }
50:     public static void main(String[] args) {
51:         ForkJoinPool pool = new ForkJoinPool(number of threads);
52:         Layer0 layer0 = new Layer0();
53:         pool.invoke(layer0); //start Fork/Join
54:     }
55: }
```

Fig. 9 Mainp class within a task-driven parallel code.

lelization directives (e.g., Other in Fig. 2), the name of each user-defined class is preserved as shown in Fig. 8. Each method within the user-defined class is converted into a ForkTemplate-style class like ForkTemplate_func in Fig. 8.

A ForkTemplate-style class corresponds to a method defined by the user, and several ForkTemplate-style classes may exist. In the ForkTemplate_func class shown in Fig. 8, shared variables among macrotasks are declared at line 3. The constructor at lines 4–6 is used when forking an instance of ForkTemplate_func. Arguments of the constructor include a macrotask identifier which can identify macrotasks to be executed within ForkTemplate_func. `compute()` at lines 7–9 executes a corresponding self-manageable macrotask code (at lines 10–15). `mt21()` at line 10 corresponds to MT21 in Fig. 3, `mt22()` at line 11 corresponds to MT22 in Fig. 3, and `mt23()` at lines 12–15 is a dummy-macrotask corresponding to MT23(Exit) in Table 1. Note that even if the user defines several classes, the compiler generates respective corresponding classes.

4.1.3 Manip Class Corresponding to First-layer MTG

When executing the task-driven parallel code, firstly a method `main()` within Mainp class (at lines 50–54 in Fig. 9) is executed. This `main()` method creates a thread pool including worker-threads. Then, parallel processing based on Fork/Join framework starts by the invocation of an instance of Layer0 class.

For the 0th-layer macrotask that means an entire program, Layer0 class (at lines 40–49 in Fig. 9) are prepared and its instance executes a method `compute()` within it. This leads to forking a macrotask mtStart in ForkTemplate_main within Mainp (at lines 10–12 in Fig. 9), where `compute()` executes a corresponding self-manageable macrotask-code (at lines 13–38 in Fig. 9) within ForkTemplate_main.

4.2 ForkTemplate-style Class

The ForkTemplate-style class shown in Fig. 6 (b) is generated for each method including parallelization directives within a source program. The execution of macrotasks within the method is realized by forking an instance of the ForkTemplate-style class.

For example, ForkTemplate_main class (at lines 2–39 in Fig. 9) corresponds to a method `main()` of a source program shown in Fig. 2. Similarly, ForkTemplate_func class (at lines 2–16 in Fig. 8) corresponds to the method `func()` within Other class defined by the user.

A ForkTemplate-style class consists of self-manageable macrotask codes, each of which is processed as follows: (1) To execute a macrotask; (2) To renew a macrotask-management table preserving states such as Finish or Branch; (3) To try to fork succeeding macrotask-candidates by checking their earliest-executable conditions, or namely to push the executable succeeding macrotask-candidates to a worker-queue.

5. Parallelizing Compiler

The task-driven parallel code based on Fork/Join framework is advantageous in terms of use of parallelism. However, it is not easy for users to generate such a parallel code manually. Therefore, we have developed a parallelizing compiler as a prototype.

As shown in Fig. 1, this compiler reads a Java program with directives as a source program, and it generates a parallel Java program for task-driven execution. Namely, the compiler works as a source-to-source translator.

5.1 Specification of Parallelizing Compiler

The developed parallelizing compiler reads a Java program including parallelization directives listed in **Table 2**, analyzes the earliest executable conditions [6] of the macrotasks and generates a task-driven parallel code based on Fork/Join framework.

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. They are called nonparallel-targeted code as mentioned later. Here, if a parallel-target code may invoke a method within nonparallel-targeted codes, the method is assumed to be converted to a thread-safe method in advance.

5.2 Parallelization Directives

To perform the task-driven execution for coarse grain parallelization, it is necessary to insert the parallelization directives in Table 2 into a target Java program. Then the proposed paralleliz-

Table 2 Parallelization directives.

Denotation	Definition
/*mt fork*/	a macrotask
/*mt fork inner*/	parallelization within a macrotask
/*premt*/	a preprocessing part
/*postmt*/	a postprocessing part
/*mt fork decomp=*value*/	the number of loop-decomposition
reduction(+:*variable*)	reduction variable as a clause of the directive decomp
private(*variable*)	private variable as a clause of the directive decomp
/*mt fork *logical-expression*/	earliest executable condition (EEC)†

†: e.g., The EEC of MT7 inside MTG0 in Table 1 is denoted by /*mt fork (0 3)&(0 4)&(0 5)*/.

ing compiler generates a task-driven parallel code.

To define a macrotask as mentioned in Section 3.2, the parallelization directive /*mt fork*/ must be used as shown in Fig. 2. With regard to a macrotasks having sub-macrotasks, e.g., a repetition block or a class method, the macrotask needs to be annotated by /*mt fork inner*/ and its sub-macrotasks also need to be annotated by /*mt fork*/. A sequential-execution region for preprocessing or postprocessing should be denoted by parallelization directives such as /*premt*/ and /*postmt*/.

Optionally, several parallelization directives to enhance performance are prepared as follows. A directive /*mt decomp=*value*/ specifies the number of decomposition of a parallelizable loop. Also, reduction(+:*variable*), private(*variable*) can be used as a directive clause. A directive /*mt fork *logical-expression*/ can explicitly indicate an earliest executable condition of a macrotask, where the *logical-expression* is written by a pair of '(*MTG-number macrotask-number*)', the symbol '&' meaning AND, the symbol '|' meaning OR and the symbol '*' meaning branch. The example is indicated in the footnote of Table 2. Note that such an earliest executable condition can be analyzed within a method automatically, but the directive is effective for exploiting parallelism among methods.

5.3 Implementation of Parallelizing Compiler

The parallelizing compiler has been developed by Java language. The structure of the compiler is shown in Fig. 1, where the lexical analysis and the syntax analysis are implemented by the Jay/JFlex parser generator that deals with the LALR(1) grammar. In this compiler, a source program with directives is converted into the abstract-syntax-tree (AST) as intermediate representation.

The compiler recognizes macrotasks denoted by parallelization directives in Table 2, analyzes both control dependency and data dependency, analyzes the earliest executable condition as shown in Table 1, and generates a task-driven parallel code based on Fork/Join framework. Here, the compiler analyzes task-driven succeeding macrotasks from earliest executable conditions of the macrotasks and reflects them to the task-driven parallel code.

In this prototype compiler, the data dependency related to the primitive type can be analyzed automatically, but the data dependency related to the reference type is preserved in consideration of the alias problem. Thus, to effectively extract inter-method parallelism, users can partially denote the earliest executable condition as a parallelization directive.

6. Performance Evaluation of Task-driven Parallel Code on Multicore Systems

This section presents performance evaluation results by means of the task-driven parallel code on multicore systems. This evaluation consists of the overall evaluation using Java Grande Forum Benchmark Suite Version 2.0 [11] and the overhead evaluation using the integral program. The execution time measured in this paper is expressed as an average of the middle three data within five measurements.

6.1 How to Evaluate on Android Platform

This subsection uses four programs from Java Grande Forum Benchmark Suite Version 2.0 whose property is indicated in **Table 3**. For these programs, we apply the program restructuring techniques such as inline expansion and constant propagation, and merged several program files into a Java file. After that, we inserted parallelization directives listed in Table 2. In this evaluation, the developed parallelizing compiler reads a source program with parallelization directives as an input file, and generates a task-driven parallel code.

Next, in the evaluation compared to conventional schemes, we use a parallel code using Runnable interface and a parallel code using Fork/Join framework without our task-driven execution. A parallel code using Runnable interface is generated manually in following manner: decomposing each parallelizable loop into N_d parts of loop where N_d is the number of loop-decomposition indicated in Table 3 and generating the threaded code which executes the decomposed partial loop on each thread. Meanwhile, a parallel code using Fork/Join framework without our task-driven execution is generated manually in following manner: decomposing each parallelizable loop into N_d parts of loop and generating the Fork/Join code which forks the decomposed partial loop on each worker thread.

In the target multicore systems such as Galaxy S6 and NVIDIA Shield Tablet indicated in **Table 4**, the parallel codes are executed by ART runtime of Android OS. We use Android Studio 2.1 to build and load the parallel code to the Android device. The Android Studio converts Java files into dex (Dalvik Executable) files via class files. Since ART runtime on Android device adopts AOT (Ahead-Of-Time) compilation policy, dex files are converted into native codes like ARM instructions when loading the parallel codes to the Android device. This type of method allows reducing overhead due to JIT (Just-In-Time) compilation of Dalvik runtime.

Table 3 Performance evaluation programs from Java Grande Forum Benchmark Suite.

Property	Crypt	Series	Monte-Carlo	Ray-Tracer
Data set	C(N = 50,000,000)	B(N = 100,000)	A(N = 10,000)	B(N = 500)
The length of parallel-targeted code	308	505	553	448
The length of compiler-generated parallel code	3,251	1,155	1,349	4,071
The length of nonparallel-targeted code	—	—	2,585	998
The number of inserted parallelization directives	5	6	5	3
The number of loop-decomposition	16	16	16	50
1-thread task-driven execution time on Galaxy S6 [ms]	5,471	87,746	5,507	657,551
The number of executed macrotasks including dummy-macrotasks	37	22	21	53
Macrotask-granularity (average macrotask execution time) [ms]	148	3,988	262	12,407

Table 4 Environment of performance evaluation.

Device	Samsung Galaxy S6	NVIDIA Shield Tablet	Dell PowerEdge R730
Processor	Exynos 7420 (2.1 GHz+1.5 GHz)	Tegra K1 (2.3 GHz)	Intel Xeon E5-2680 v3 (2.5 GHz)
CPU core	4-cores Cortex-A57 + 4-cores Cortex-A53	4-cores Cortex-A15r3	12-cores
Memory	3 GB	2 GB	64 GB
OS	Android 6.0.1	Android 6.0	CentOS 6.7
Java environment	ART (API level 23)	ART (API level 23)	JDK1.8 (with HotSpot Optimization)

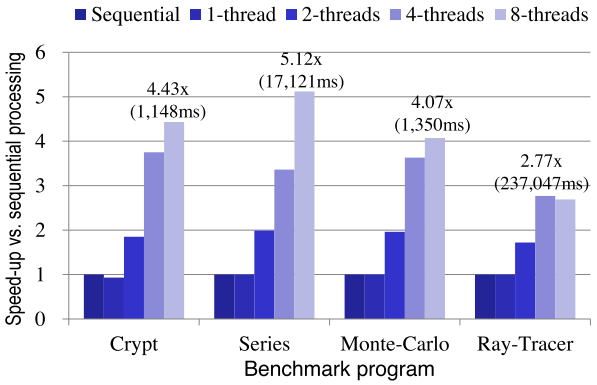


Fig. 10 Task-driven coarse grain parallel processing on Galaxy S6.

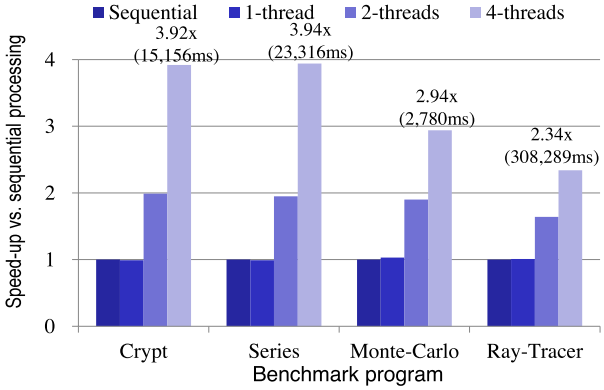


Fig. 11 Task-driven coarse grain parallel processing on Shield Tablet.

In a file MainActivity.java within project files in the Android Studio, we prepared a template code including both the onClick() method and the user interface to select the number of worker-threads in advance. The compiler-generated task-driven parallel codes are invoked from this template code respectively so as to perform the coarse grain parallel processing on the Android device.

6.2 Performance Evaluation of Task-driven Parallel Execution for Benchmark Programs

This subsection describes a performance evaluation for using four programs from Java Grande Forum Benchmark Suite Version 2.0 [11] on Android platform and Linux platform.

6.2.1 Evaluation on Android Platform

In this evaluation, the task-driven parallel codes are generated by our compiler and are executed on Android platform.

Crypt on Android Crypt is a program to encrypt and decrypt by using IDEA (International Data Encryption Algorithm) and its code length is 308 lines. Our parallelizing compiler generates 3,251-lines of task-driven parallel code from a source code with parallelization directives. According to the execution result of the parallel code on Galaxy S6 shown in **Fig. 10** (or **Table A·1**), eight-threads execution gave us 4.43 times faster speedup compared to the original sequential processing without parallelization. As for the execution result on Shield Tablet shown in **Fig. 11** (or **Table A·2**), four-threads execution gave us 3.92 times faster speedup compared to the original sequential processing.

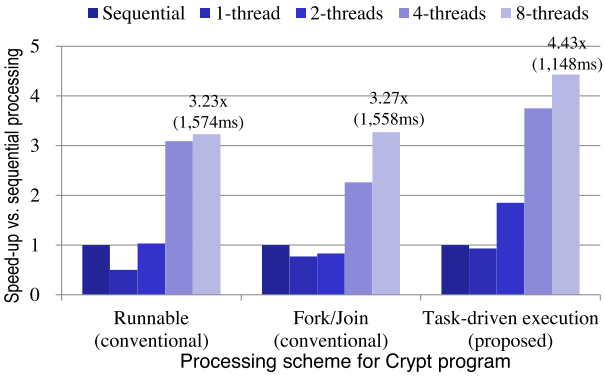


Fig. 12 Comparison of parallel processing schemes on Galaxy S6.

Note that Galaxy S6 is composed of heterogeneous multi-cores indicated in Table 4, namely four-threads execution can use high-speed four cores, but eight-threads execution must use both high-speed four cores and low-speed/low-power four cores. Thus the performance on heterogeneous eight cores cannot attain an ideal speedup.

In comparison with conventional schemes, we prepare a parallel code using Runnable interface and a parallel code using Fork/Join framework without task-driven execution manually. As shown in **Fig. 12** (or **Table A·3**), the Runnable execution with eight-threads of Galaxy S6 gave us 3.23

Table 5 Single core execution and Math-class-library ratio in Crypt and Series programs.

Program	Execution	Time[ms] on Galaxy S6	Time[ms] on Xeon	
			JVM with HotSpot	JVM without HotSpot
Crypt	Sequential	5,087	1,933	23,665
	(Math-class-library ratio)†	(0%)	(0%)	(0%)
	1-thread task-driven	5,471	3,086	21,939
Series	Sequential	87,575	46,154	73,674
	(Math-class-library ratio)†	(79%)	(84%)	(65%)
	1-thread task-driven	87,746	30,788	74,372

†: Ratio of Math-class-library execution time against Sequential processing time

times faster speedup versus sequential processing, and the Fork/Join execution without task-driven execution gave us 3.27 times faster speedup versus sequential processing. On the contrary, the proposed task-driven execution scheme could achieve 4.43 times faster speedup as mentioned above. That is to say, the proposed scheme is superior to the Runnable execution by 27% and to the Fork/Join execution without task-driven execution by 26% in terms of execution time.

Series on Android Series is a program composed of 505-lines of source code to calculate Fourier coefficients. Our parallelizing compiler generates 1,155-lines of task-driven parallel code. The execution result on eight-threads of Galaxy S6 in Fig. 10 (or Table A·1) gave us 5.12 times faster speedup versus sequential processing. The execution result on four-threads of Shield Tablet in Fig. 11 (or Table A·2) shows gave us 3.94 times faster speedup versus sequential processing.

Such an ideal speedup of Series results from the properties of its program. The partial execution time for Math-class-library is as much as 79% of the sequential processing time on Galaxy S6 in **Table 5**. In addition, the memory access cost of Math-class-library seems to be less than that of ordinary Java codes written by users. Thus, Series can achieve better performance than the other benchmark programs.

Monte-Carlo on Android Monte-Carlo is a program of a financial simulation using Monte-Carlo techniques to price derived products. This source program consists of both the 553-lines of parallel-targeted code where our parallelization directives are inserted and the 2,585-lines of nonparallel-targeted code where our parallelization directives are not inserted. Our parallelizing compiler reads the 553-lines of parallel-targeted code and generates the 1,349-lines of task-driven parallel code. In the execution on the Android device, both the 1,349-lines of task-driven parallel code and the 2,585-lines of nonparallel-targeted code are loaded to the Android device.

The eight-threads execution on Galaxy S6 in Fig. 10 (or Table A·1) shows 4.07 times faster speedup versus sequential processing. The four-threads execution on Shield Tablet in Fig. 11 (or Table A·2) shows 2.94 times faster speedup versus sequential processing.

Ray-Tracer on Android Ray-Tracer is a program to measure the performance of a 3D ray tracer. The scene including 64 spheres is rendered. This source program consists of both the 448-lines of parallel-targeted code and the 998-lines of nonparallel-targeted code. Our parallelizing compiler reads the 448-lines of parallel-targeted code and generates the 4,071-lines of task-driven parallel code. Considering data size N = 500, the number of loop-decomposition is determined as 50. In the execution, both the task-driven parallel code and the nonparallel-targeted code are loaded to the Android device.

The eight-threads execution on Galaxy S6 in Fig. 10 (or Table A·1) shows 2.77 times faster speedup versus sequential processing. The four-threads execution on Shield Tablet in Fig. 11 (or Table A·2) shows 2.34 times faster speedup versus sequential processing. These speedup ratio is not enough performance, but it is dependent on using frequent instance creations on the Android platform. On the other hand, the execution result on the Linux platform presents better scalability as mentioned in 6.2.2, and parallelism seems to be extracted within this program.

These performance evaluations with four programs on Android platforms such as Galaxy S6 and Shield Tablet confirmed that coarse grain parallelization using the task-driven parallel code can extract parallelism efficiently and can drastically reduce the execution time. Moreover, it means that the proposed task-driven execution policy can treat both the parallel-targeted code and the nonparallel-targeted code. This is beneficial for parallelization of large-scale applications.

6.2.2 Evaluation on Linux Platform

As the proposed task-driven parallel code is platform-free, the parallel code enables performing coarse grain task parallel processing on not only the Android platform but also the Linux platform. This subsection performs the evaluation on DELL PowerEdge R730 equipped with Intel Xeon E5-2680 v3 (12cores, 2.5 GHz) specified in Table 4. Linux platform adopts JDK1.8 as a Java execution environment where HotSpot optimization is applied by default. The execution result on Xeon E5-2680 is shown in **Fig. 13** (or **Table A·4**) and **Fig. 14** (or **Table A·5**).

Crypt on Linux Crypt program on 12-threads gave us 7.19 times faster speedup versus 1-thread task-driven execution, or 4.51 times faster speedup versus sequential processing in Fig. 13. These results show that parallelism within this program is extracted efficiently, but 1-thread task-driven execution is slower than sequential processing on JVM with HotSpot optimization (Just-In-Time compilation). As can been seen from Table 5, unless HotSpot optimization is applied, 1-thread task-driven execution time is slightly shorter than sequential processing time. However, provided that HotSpot optimization is applied, 1-thread task-driven execution time is longer than sequential processing time. This is

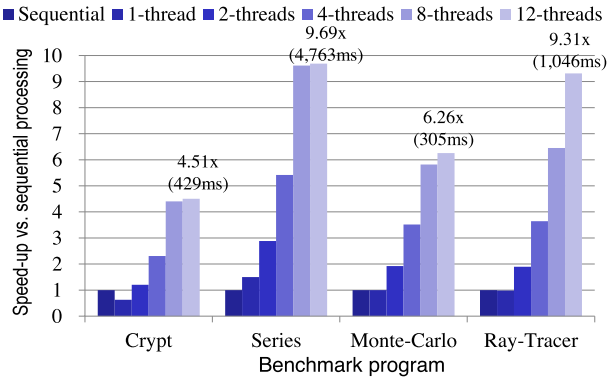


Fig. 13 Task-driven coarse grain parallel processing on Xeon E5-2680.

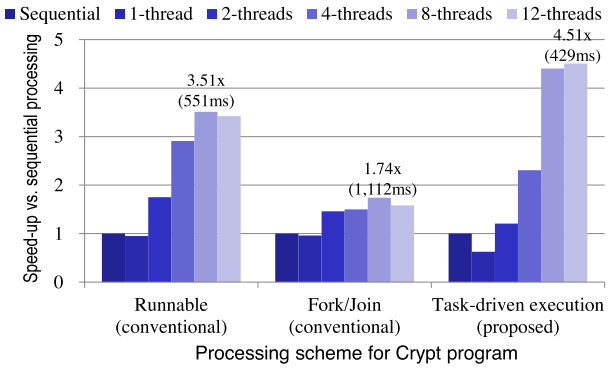


Fig. 14 Comparison of parallel processing schemes on Xeon E5-2680.

because the length of compiler-generated parallel code with the loop-decomposition is 10.6 times as long as that of the sequential processing code in Table 3. Namely, such a long code degrades the performance of HotSpot optimization.

Meanwhile, Fig. 14 shows comparison with conventional processing schemes for Crypt program. The Runnable implementation on eight-threads gave us 3.51 times speedup versus sequential processing, and the Fork/Join implementation on eight-threads without task-driven execution gave us 1.74 times speedup versus sequential processing. Therefore, the proposed task-driven execution can respectively reduce execution time by 20% versus the Runnable implementation and by 61% versus the Fork/Join implementation without task-driven execution.

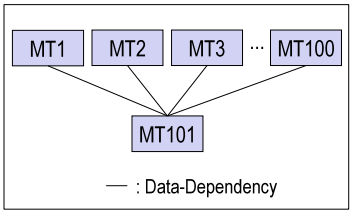
Series on Linux Series program on 12-threads achieves 9.69 times speedup versus sequential processing as shown in Fig. 13. In this program, let us consider the effect due to the HotSpot optimization of JVM on Xeon. As shown in Table 5, unless HotSpot optimization is applied, the 1-thread task-driven execution time of Series is almost the same as the sequential processing time. But, if HotSpot optimization is applied, the 1-thread task-driven execution time is remarkably shorter than the sequential processing time. This is because HotSpot optimization is more effective for the loops decomposed by the directive `/*mt fork decomp=16*/` in 1-thread task-driven execution.

Monte-Carlo on Linux Monte-Carlo program on 12-threads achieves 6.26 times speedup versus sequential processing in Fig. 13.

Ray-Tracer on Linux Ray-Tracer program on 12-threads achieves 9.31 times speedup versus sequential processing in Fig. 13.

```
/*mt fork decomp=100
  reduction(+:sum)
  private(x1, x2, y1, y2)*/
{
  for(i=0;i<N;i++){
    x1=i*step;
    x2=(i+1)*step;
    y1=4.0/(1.0+x1*x1);
    y2=4.0/(1.0+x2*x2);
    sum+=(y1+y2)*step/2.0;
  }
}
```

(a)Integral program with directives



(b)Macro-Task-Graph of Integral program (The number of loop-decomposition is 100.)

Fig. 15 Integral program with directives and its MTG.

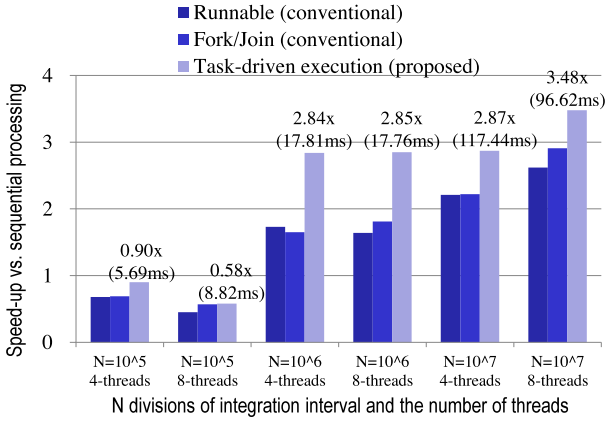


Fig. 16 Overhead evaluation using Integral program on Galaxy S6.

As mentioned above, the task-driven parallel code can attain good speedup on the Linux platform and is effective for efficiently extracting parallelism.

6.3 Overhead Evaluation of Task-driven Parallel Code

This subsection evaluates overhead of the proposed task-driven parallel code. The evaluation program is the Integral program that solves Pi by using the trapezoidal rule as shown in **Fig. 15** (a). In a loop to calculate the definite integral, a parallelization directive `/*mt fork*/` is inserted with directive clauses such as `decomp=100`, `reduction(+:sum)` and `private(x1,x2,y1,y2)`. This directive directs the compiler to decompose a loop into 100 partial loops, namely MT1–MT100, and to recognize them as macrotasks as shown in Fig. 15 (b), where MT101 calculates a total value from partial sums. The number of divisions of the integration interval is N. The value N varies $N = 10^5, 10^6, 10^7$ as shown in **Fig. 16**, which signifies adjusting the macrotask's grain or the macrotask's execution time. Macrotasks like MT1–MT100 calculate N/100 parts of trapezoidal areas respectively. Next, for comparison with conventional schemes, we generate a Runnable implementation code and a Fork/Join implementation code without the task-driven execution in manual.

Figure 16 presents parallel execution times in three executions on the condition that $N = 10^5, 10^6, 10^7$ and the number of threads is four or eight. These results indicate that coarse grain parallel processing with the proposed task-driven execution can demonstrate better performance than conventional implementations such as Runnable or Fork/Join without task-driven execu-

Table 6 Macrotask-granularity of task-driven execution for Integral program on Galaxy S6.

Property	The number of divisions for integral		
	$N = 10^5$	$N = 10^6$	$N = 10^7$
1-thread task-driven execution time[ms]	8.043	56.072	338.932
The number of executed macrotasks including dummy-macrotasks	104	104	104
Macrotask-granularity (average macrotask execution time) [ms]	0.077	0.539	3.259

tion in any case.

In more detail, the conventional Runnable implementation code creates the same number of threads as cores, where MT1–MT100 are assigned to respective threads uniformly. Therefore, it is difficult for a heterogeneous multicore system to attain best performance owing to differences in core performance. Next, in the conventional Fork/Join implementation code, a class that extends RecursiveTask is prepared to perform calculation corresponding to MT1–MT100, and the overridden method `compute()` executes a partial region directed by a parameter. For this class, 100 instances are forked and joined in Fork/Join framework where the number of worker-threads is equal to the number of cores. On the other hand, since the proposed task-driven implementation which adopts the layer-unified execution control [6] can cope with the dynamic macrotask scheduling based on the earliest-executable-conditions, we can obtain better load-balancing and less scheduling overhead than conventional implementations.

As a result, in coarse grain parallel processing with the task-driven execution, $N = 10^6$ or $N = 10^7$ (where each macrotask calculates N/100 parts of areas) is suitable as the macrotask-granularity on Galaxy S6. The execution of $N = 10^7$ on eight-threads gave us 3.48 times speedup versus sequential processing. **Table 6** shows the macrotask-granularity (or the average macrotask execution time). The macrotask-granularity even if $N = 10^7$ is very small compared to that of ordinary programs. Therefore, it is found that the macrotask-granularity of benchmark programs in Table 3 is sufficient for effectively extracting parallelism.

7. Related Works

Research on parallelization of Java programs has proposed methods including the following: HPJava [12] to adopt array distribution like HPF; Parallel Java [13] to present API for shared memory/cluster parallel programming like OpenMP/MPI; zJava [14] to use parallelism among asynchronous threads by runtime support; Jrpm [15] to perform speculative execution by using hardware profile and so on. However, these schemes differ from our approach that utilizes coarse grain parallelism by layer-unified execution control [6].

On the other hand, the parallel code generation scheme for the layer-unified coarse grain parallelization has been proposed [7]. This scheme can generate a parallel Java code with the original scheduler implemented by Runnable interface, but the generated parallel code does not adopt the task-driven execution style and can not cope with Java Fork/Join framework.

The scheduler with work-stealing is widely introduced to not only Java Fork/Join framework but also Cilk [16], Intel TBB (Threading Building Blocks) [17], X10 [18] and so on. This is because it is effective for scheduling small-grain tasks with low-overhead. Habanero-Java [19] proposed the runtime extension of work-stealing to support async-finish parallelism by the compiler. For fine grain task scheduling on embedded systems, the lightweight Fork-Join framework ARTM [20] is proposed. Since these researches focus on improvement of work-stealing mechanism, they are different from our proposed scheme that performs coarse grain parallel processing with task-driven execution based on Fork/Join framework.

8. Conclusions

This paper proposes a task-driven parallel code generation scheme based on Fork/Join framework, which enables performing the coarse grain parallel processing with dynamic scheduling on Android platform and for which we developed a prototype parallelizing compiler. This parallelizing compiler as a source-to-source translator can read a Java program with parallelization directives and can generate a task-driven parallel code automatically. The task-driven parallel code is expressed by ForkTemplate-style class, which manages macrotask's execution and performs coarse grain parallel processing by effectively using the Fork/Join framework.

In the performance evaluation, four programs with parallelization directives from Java Grande Forum Benchmark Suite Version 2.0 are compiled into task-driven parallel codes by the developed parallelizing compiler. When the parallel codes are executed on Samsung Galaxy S6 equipped with heterogeneous multicores and NVIDIA Shield Tablet equipped with a homogeneous multicore, the execution results for these four programs could attain 2.77–5.12 times speedup on Galaxy S6 and 2.34–3.94 times speedup on Shield Tablet versus sequential processing. Also, the compiler-generated task-driven parallel code is executable for Linux platform based on Intel Xeon E5-2680 and 12-threads execution could achieve 4.51–9.69 times speedup versus sequential processing.

Consequently coarse grain parallelization with task-driven execution is confirmed as effective for Java parallel processing on an Android platform. And the developed parallelizing compiler that achieves high performance is valuable for application developers, library developers and general programmers.

In the future, we would like to develop a preprocessor that automatically inserts parallelization directives into target programs and attains automatic parallelization for Java programs.

Acknowledgments This work was partly supported by JSPS Grant-in-Aid for Scientific Research (C) Grant Number 16K00174.

References

[1] 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).

[2] 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).
[3] 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).
[4] 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).
[5] 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).
[6] Yoshida, A.: Layer-unified Execution Control Scheme for Coarse Grain Task Parallel Processing, *IPSJ Journal*, Vol.45, No.12, pp.2732–2740 (2004).
[7] Yoshida, A., Ochi, Y. and Yamanouchi, N.: Parallel Java Code Generation for Layer-Unified Coarse Grain Task Parallel Processing, *IPSJ Transaction ACS*, Vol.7, No.4, pp.56–66 (2014).
[8] 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).
[9] Lea, D.: A Java Fork/Join Framework, *Proc. ACM conference on Java Grande, JAVA'00*, pp.36–43 (2000).
[10] 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).
[11] EPCC: The Java Grande Forum Benchmark Suite (2014), available from ⟨http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/⟩.
[12] 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).
[13] 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).
[14] Chan, B. and Abdelrahman, T.S.: Run-Time Support for the Automatic Parallelization of Java Programs, *J. Supercomputing*, Vol.28, pp.91–117 (2004).
[15] Chen, M.K. and Olukotun, K.: The Jrpm System for Dynamically Parallelizing Java Programs, *Proc. ISCA-30*, pp.434–446 (2003).
[16] Frigo, M., Leiserson, C. and Randall, K.: The Implementation of the Cilk-5 Multithreaded Language, *Proc. ACM SIGPLAN conference on Programming Language design and implementation, PLDI'98*, pp.212–223 (1998).
[17] Reinders, J.: Intel Threading Building Blocks, *O'Reilly & Associates, Inc. Sebastopol, CA, USA* (2007).
[18] Tardieu, O., Wang, H. and Lin, H.: A Work-Stealing Scheduler for X10's Task Parallelism with Suspension, *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'12* (2012).
[19] Raman, R., Zhao, J., Budimlic, Z. and Sarkar, V.: Compiler Support for Work-Stealing Parallel Runtime Systems, *Rice Technical Report TR10-02* (2010).
[20] Ojail, M., David, R., Lhuillier, Y. and Guerre, A.: ARTM: A lightweight fork-join framework for many-core embeded systems, *Proc. Design, Automation & Test in Europe Conference & Exhibition* (2013).

Appendix

Table A·1 Execution time of task-driven coarse grain parallel processing on Galaxy S6.

Execution	Crypt	Series	Monte-Carlo	Ray-Tracer
	Time [ms] (Speedup)	Time [ms] (Speedup)	Time [ms] (Speedup)	Time [ms] (Speedup)
Sequential	5,087 (1.00)	87,575 (1.00)	5,492 (1.00)	656,938 (1.00)
1-thread	5,471 (0.93)	87,746 (1.00)	5,507 (1.00)	657,551 (1.00)
2-threads	2,752 (1.85)	44,102 (1.99)	2,801 (1.96)	382,213 (1.72)
4-threads	1,358 (3.75)	26,070 (3.36)	1,511 (3.63)	237,047 (2.77)
8-threads	1,148 (4.43)	17,121 (5.12)	1,350 (4.07)	243,845 (2.69)

Table A·2 Execution time of task-driven coarse grain parallel processing on Shield Tablet.

Execution	Crypt	Series	Monte-Carlo	Ray-Tracer
	Time [ms] (Speedup)	Time [ms] (Speedup)	Time [ms] (Speedup)	Time [ms] (Speedup)
Sequential	59,442 (1.00)	91,797 (1.00)	8,183 (1.00)	722,222 (1.00)
1-thread	60,142 (0.99)	92,440 (0.99)	7,955 (1.03)	716,251 (1.01)
2-threads	29,919 (1.99)	47,095 (1.95)	4,315 (1.90)	439,139 (1.64)
4-threads	15,156 (3.92)	23,316 (3.94)	2,780 (2.94)	308,289 (2.34)

Table A·3 Execution time of Crypt for comparison of parallel processing schemes on Galaxy S6.

Execution	Runnable (conventional)		Fork/Join (conventional)		Task-driven (proposed)	
	Time [ms]	Speed up	Time [ms]	Speed up	Time [ms]	Speed up
Sequential	5,087	1.00	5,087	1.00	5,087	1.00
1-thread	10,152	0.50	6,569	0.77	5,471	0.93
2-threads	4,954	1.03	6,156	0.83	2,752	1.85
4-threads	1,645	3.09	2,247	2.26	1,358	3.75
8-threads	1,574	3.23	1,558	3.27	1,148	4.43

Table A·4 Execution time of task-driven coarse grain parallel processing on Xeon E5-2680.

Execution	Crypt	Series	Monte-Carlo	Ray-Tracer
	Time [ms] (Speedup)	Time [ms] (Speedup)	Time [ms] (Speedup)	Time [ms] (Speedup)
Sequential	1,933 (1.00)	46,154 (1.00)	1,908 (1.00)	9,739 (1.00)
1-thread	3,086 (0.63)	30,788 (1.50)	1,910 (1.00)	9,936 (0.98)
2-threads	1,604 (1.21)	16,007 (2.88)	992 (1.92)	5,143 (1.89)
4-threads	838 (2.31)	8,524 (5.41)	543 (3.51)	2,672 (3.64)
8-threads	439 (4.40)	4,800 (9.62)	328 (5.82)	1,510 (6.45)
12-threads	429 (4.51)	4,763 (9.69)	305 (6.26)	1,046 (9.31)

Table A·5 Execution time of Crypt for comparison of parallel processing schemes on Xeon E5-2680.

Execution	Runnable (conventional)		Fork/Join (conventional)		Task-driven (proposed)	
	Time [ms]	Speed up	Time [ms]	Speed up	Time [ms]	Speed up
Sequential	1,933	1.00	1,933	1.00	1,933	1.00
1-thread	2,035	0.95	2,021	0.96	3,086	0.63
2-threads	1,103	1.75	1,328	1.46	1,604	1.21
4-threads	665	2.91	1,291	1.50	838	2.31
8-threads	551	3.51	1,112	1.74	439	4.40
12-threads	565	3.42	1,227	1.58	429	4.51

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 became an associate professor at Meiji University in 2013 and has been engaged as a professor at Meiji University since 2016. He has joined the green computing system research organization as a visiting professor at Waseda University since 2014. His research interests include parallel computing, parallelizing compiler and data-locality optimization. He is a member of IPSJ, IEICE, IEEJ, IEEE and ACM.

Akira Kamiyama received his B.S. degree from Toho University in 2013 and his M.S. degree from Meiji University in 2015. His research interest is parallel computing on multicore processors. Currently, he works in SoftBank Corporation.

Hiroki Oka received his B.E. degree from Meiji University in 2017. Currently, he is a graduate student at Meiji University. His research interest is parallel computing on Android platforms. He is a member of IPSJ.