A Java Fork-Join Framework-based Parallelizing Compiler for Dealing with Divide-and-conquer Algorithm

Abdourahmane SENGHOR, Karim KONATE
Department of Mathematics and Informatics
Cheikh Anta Diop University of Dakar
Dakar, Senegal
asenghor@sicap.sn, kkonate911@yahoo.fr

ABSTRACT: *Before Java Fork-Join framework implementation, the only way to parallelize recursive application in java was done in a naïve way, which consists of creating as many threads as existing tasks. However, javar introduced the cut-depth parameter that consists of limiting the creations of the number of threads, but does not allow benefiting from fine-grain parallelism.*

The purpose of this article is to design and implement a performing compiler for parallelizing Java application with divide-and-conquer algorithm. The compiler includes directives and environment variables. It is built around Java ForkJoin framework, which is directly integrated within Java 1.7 version and imported as archive library in Java 1.6 and 1.5 versions. This compiler tends to make easier and less error-prone the parallelization of recursive applications. Although in Java ForkJoin Framework there are two user-level performance parameters , which are the number of threads and the threshold, our compiler introduces another user-level performance parameter which is the MaxDepth corresponding to the maximum of depth before which parallel execution is enforced and after which sequential execution is enforced. This allows balancing between fine-grain and coarse-grain parallelisms. Experimental results are presented and discussed.

Keywords: Parallelizing Compiler, Divive-and-conquer Algorithm, ForkJoin Framework, Fine-grain Parallelism, Coase-grain Parallelism

Received: 21 November 2012, Revised 26 December 2012, Accepted 31 December 2012

1. Introduction

Over many years, the only way in Java to deal with divide-and-conquer algorithm was to do it in a naïve way by using the low level threads[7]. The naïve way method consists of creating as many threads as tasks. Knowing the cost of creating and destructing threads, this method rapidly reached its limits in terms of performance. That makes the divide-and-conquer applications relatively yielded poor performances. In 2000 Doug Lea introduces Java Fork-Join [5] framework which deals with divide-and-conquer algorithm and which implements high level threads. The framework is based on a pool of worker threads, a pool of tasks considered as lightweight processes and a strategy of task scheduling. Tasks are spawned and results are joined. The work-stealing strategy is used to manage the queuing. The performance is determined by a parameter named Threshold. However, parallelizing a recursive application by using the framework is done by hand. Java Concurrencer [3] is an effort to make easier the utilization of the framework. It is a graphic-level source-to-source parallelizing tool that transforms a sequential recursive application into parallel one. However, that tool is only utilizable with eclipse IDE and only uses Threshold as performance parameter.

Before Java Fork-Join framework implementation, Java suffers from a lack of performing parallelizing compiler dealing with recursive applications. The rare existing compilers yield poor performances. Jomp [2] is a source-to-source parallelizing compiler which deals with loop parallelism. However, recursive parallelism can be dealt with Jomp by using section directive. But, it is not a good way because, whenever the recursive method is invoked, a thread is created. Javar [1] is a source-to-source compiler that allows dealing with divide-and-conquer algorithm [9] [4]. However, the performance is a trade-off between coarse-grain parallelism and fine-grain parallelism at the spent of recurrent thread creation and destruction.

We propose to design and implement a parallelizing compiler, FJComp, which is easier to use, less error-prone and performing. This compiler tends to achieve a fine-grain parallelism while reducing the overhead of the thread and task management. The performance depends on three user-level parameters that are the number of worker threads, the threshold and the maximum of depth. The remainder of the paper is organized as follows: Section 2 is the design of the compiler, Section 3 corresponds to the implementation, Section 4 evaluates the performance of the compiler, Section 5 describes similar works. Finally Section 6 concludes.

2. Design

ForkJoin Framework [5] hides the complexity of the recursive parallel programming. Thus, both parameters the programmer must specify are the threshold and the number of worker threads. We design our compiler with respect to these specifications. However, we add another user-level performance parameter, MaxDepth, which consists of stopping the parallel execution and resuming a sequential execution after a maximum of depth of recursion is reached. The threshold and the maximum of depth are both parameters which limit the parallel recursive execution. Threshold specification depends on the type of the tasks to be divided up. For instance, in the Fibonacci algorithm, the threshold conditional expression is: `if(n<THRESHOLD`. In the Sorting algorithm such as Quicksort or Mergesort, the threshold conditional expression is: `if((right-left)<THRESHOLD)`. This means that threshold specification is bound to the type of the tasks to be subdivided and requires well understanding the algorithm. However, MaxDepth is more intuitive than Threshold because it is based on the number of recursions or the depth in the tree context. However, it is strongly recommended to not use both specifications at the same time. If both clauses are specified by the programmer, then MaxDepth is accounted for and Threshold Expression is ignored. If none of them is specified, the MaxDepth default value is used. As root of the directives, we use `//taskq`. This is followed by clauses which are optional.

```
//taskq [nthreads = <int>]
[if(threshold_exp)][MaxDepth =
<int>]
```

`nthreads=<int>` corresponds to the number of worker threads. The default value is given by `Runtime.getRuntime().availableProcessors()` and corresponds to the available number of processors of the underlying system.

`if(threshold_exp):` In case the conditional expression evaluates to false, parallel execution is done and tasks are forked and joined. Otherwise, parallel execution is stopped and sequential execution is enforced.

`MaxDepth = <int>` corresponds to the depth after which, sequential execution is enforced. Before reaching the specified maximum of depth, tasks are forked, computed in parallel and results are joined. The default value is 1.

`task:` identifies the invoked method as a recursive task. It is used inside the block of code following //taskq.

Example1: Fibonacci Computation

```
public class Fibonnacci {
   public long fibonacci(int n)
     { if(n==0) return 0;
if (n==1) return 1; long x, y;
     //taskq nthreads=4 MaxDepth=4
     {
        //task
        x=fibonacci(n-1);
       //task
        y=fibonacci(n-2);
     }
        return x+y ;
    }
}
```

Example 2. Quicksort Sorting

```
public class Quicksort {
   public void Qq ( int[]a,intleft,int right )
   {
     int pivotIndex=partition(a,left,right ) ;
     //taskq nthreads=8 if((left-right)<100)
     {
       if ( left < pivotIndex )
         //task
         Qs(a,left,pivotIndex ) ;
       if ( pivotIndex + 1 < right )
         //task
          Qs(a,pivotIndex+1,right) ;
      }
    }
}
```

3. Implementation

Building our parallelizing compiler is made possible thank to the utilization of the JavaCC [6] compiler generator. JavaCC includes a `jjt` extension file that contains tokens and syntactic expressions. It also includes a `jj` extension file that implements the abstract syntactic tree generated from the syntactic expressions. For instance `//taskq` is registered as a token and also corresponds to a syntactic expression and then represents a node within the abstract syntactic tree. Each node is a java extension file and contains a method that accepts the node be visited. The encountered constructs are unparsed and then corresponding instructions are written within a method of `AddAcceptVisitor` class. A class implementing a symbol table is added. This class tracks the local variables and parameters.

The generated code is made up of the original class in which we add a private class which subclasses `RecursiveAction` and then overrides the `compute` method. In that method, the tasks are created and concurrently executed. The content of the sequential recursive method is modified and transformed into a parallel method involving `ForkJoinPool` to create the worker threads that compute the recursive tasks. When recursive method returns a value, our compiler creates an instance variable named `result` which holds the various returned values.

4. Experiments

We propose to test the performance of the programs parallelized by FJComp. The objective is 1) to compare the performances following the JVM versions (Java 6 and Java 7) 2) to compare MaxDepth and Threshold parameters.

4.1 Environment of test

Our programs are performed in an IBM X 3650 with 8 -cores Quad-Core Intel Xeon processors E5420, 8 Gigabytes of RAM running on Linux Fedora 11 with Java 1.6.0 and Java 1.7.0 runtime environment.

Our experiments cover Fibonacci, Quicksort, Mergesort, Integrate and Matrix Multiplication. Fibonnaci is a computational algorithm summing natural integer. QuickSort is a divide-and-conquer algorithm which partitions list around a pivot. The elements of the left-side list are inferior to the pivot while the elements of the right-side list are superior to the pivot. Mergesort is a divide-and-conquer algorithm which divides the list into two equal sub lists until the sub list is reduced to two elements. A reverse traverse is used to sort back the sub-lists and merge them until getting the original list, which is then sorted. Integrate is a mathematical computing corresponding to the recursive Gaussian quadrature summing over odd values from 1 to 5 and integrating from -47 to 48. We have two versions of Matrix Multiplication, an iterative version and an recursive version. The recursive version is O(n^3) while the recursive version using Strassen's algorithm is O($n^{2.8}$) of complexity.

4.2 Results

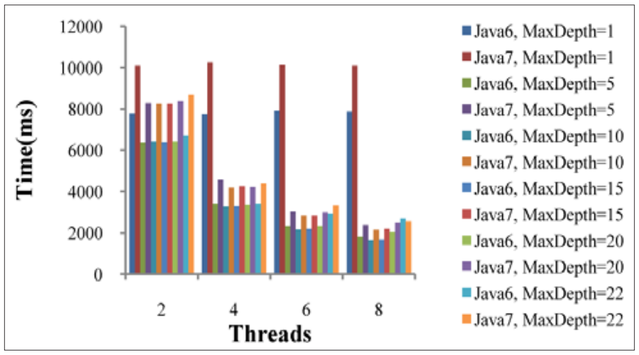

Figure 1. Comparing performances of Fibonacci dealt with FJComp over Java 6 and Java 7 by using the MaxDepth parameter

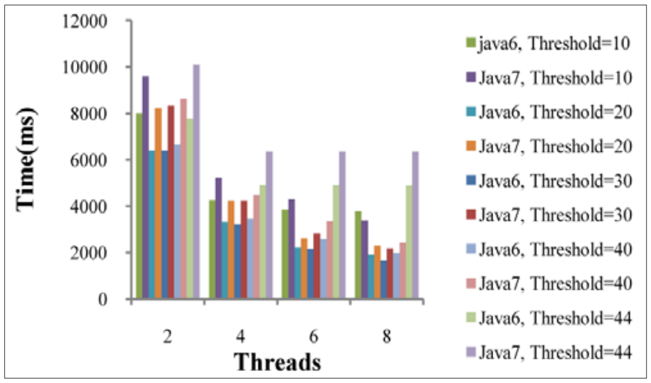

Figure 2. Comparing performances of Fibonacci dealt with FJComp over Java 6 and Java 7 by using the Threshold parameter

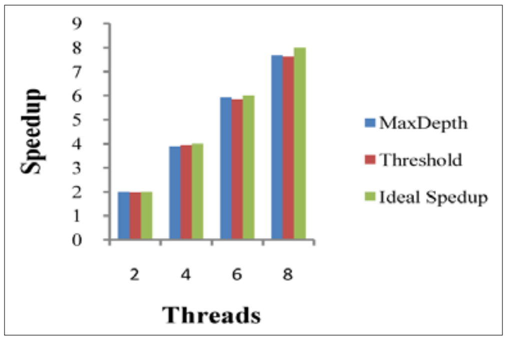

Figure 3. Comparing speedups of Fibonacci by using Threshold and MaxDepth parameters

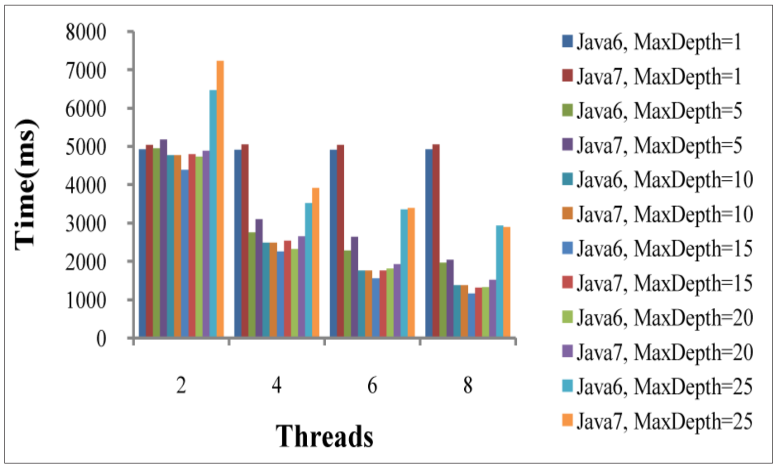

Figure 4. Comparing performances of Integrate dealt with FJComp over Java 6 and Java 7 by using the MaxDepth parameter

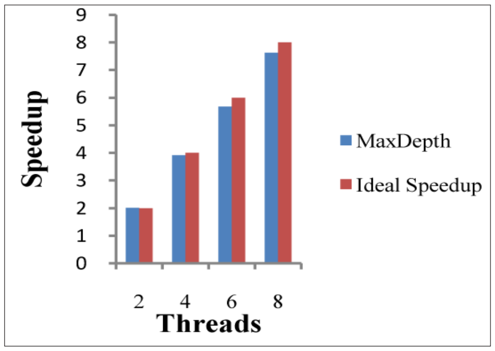

Figure 5. Speedups of Integrate by using MaxDepth parameters

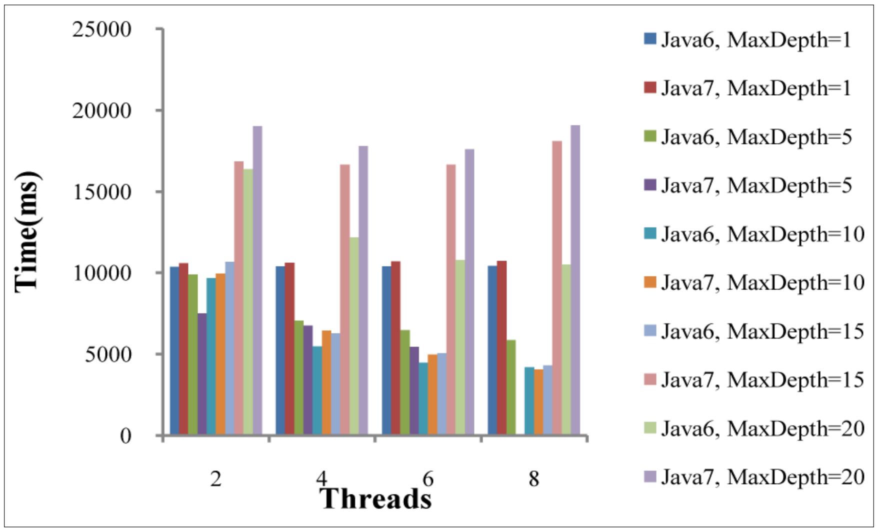

Figure 6. Comparing performances of Quicksort dealt with FJComp over Java 6 and Java 7 by using the MaxDepth parameter

4.3 Discussion

4.3.1 Comparison of performances of FJComp running over Java 7 and Java 6

Figure 1, Figure 2, Figure 4, Figure 6, Figure 7 corresponding to the execution of Fibonacci, Integrate and Quicksort show that

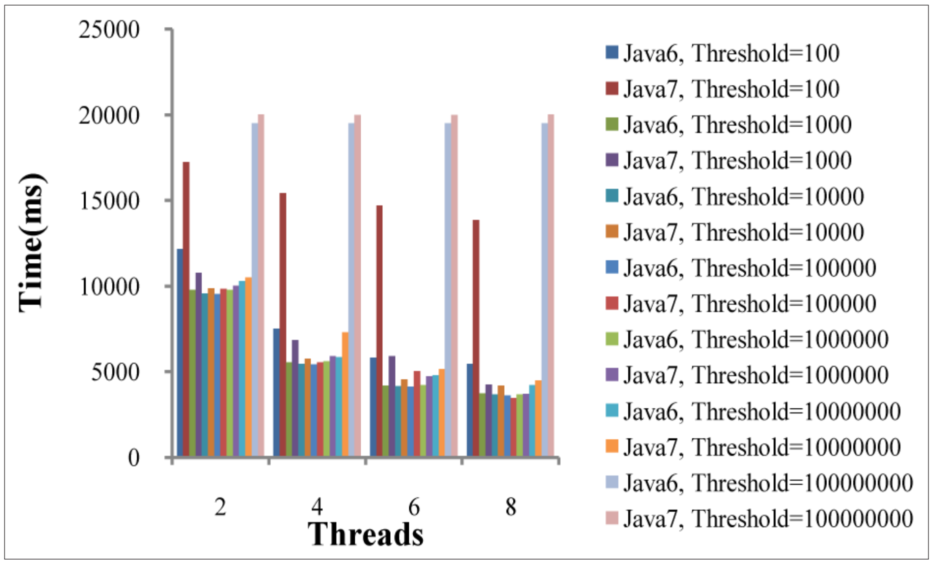

Figure 7. Comparing performances of Quicksort dealt with FJComp over Java 6 and Java 7 by using the Threshold parameter

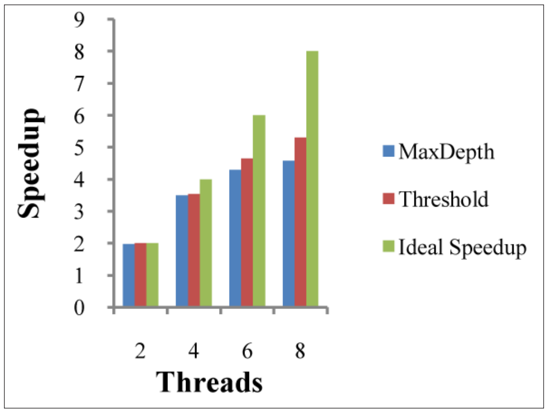

Figure 8. Comparing speedups of Quicksort by using Threshold and MaxDepth parameters

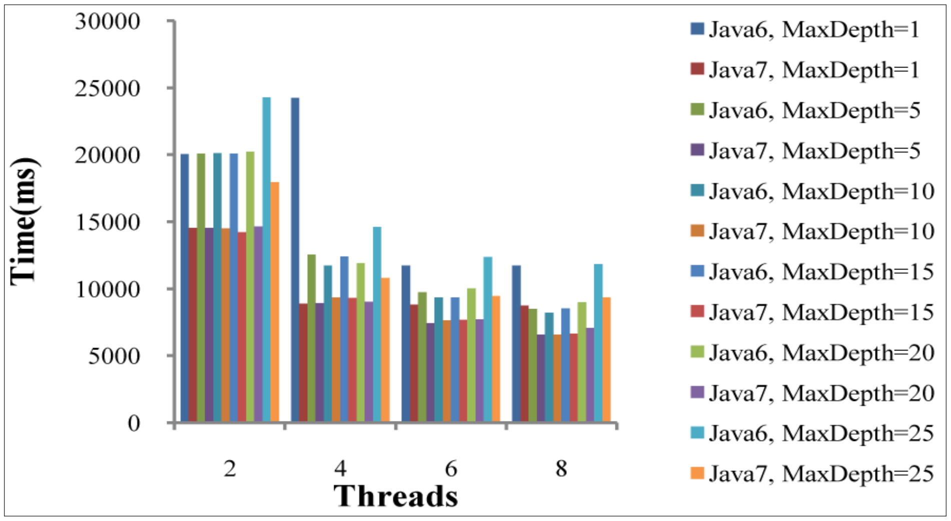

Figure 9. Comparing performances of Mergesort dealt with FJComp over Java 6 and Java 7 by using the MaxDepth parameter

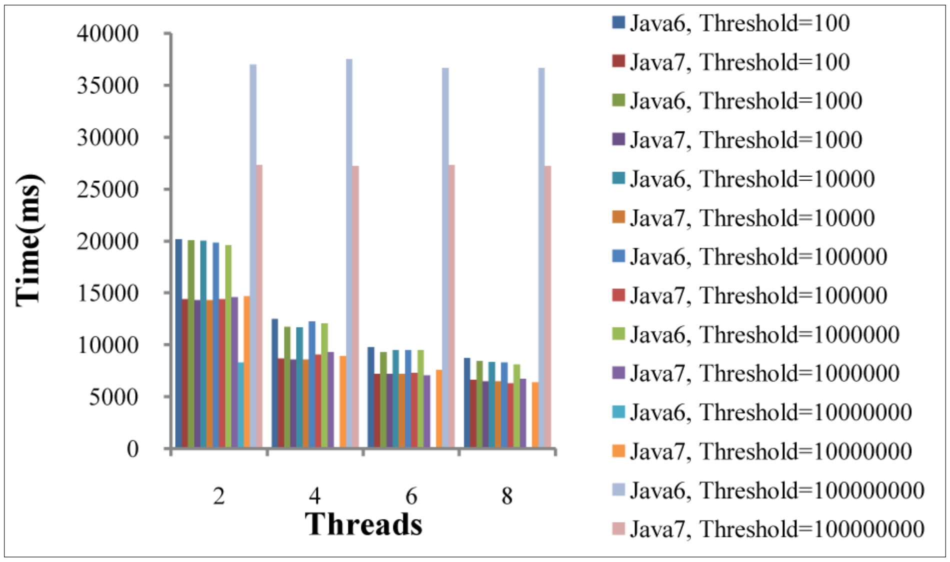

Figure 10. Comparing performances of Mergesort dealt with FJComp over Java 6 and Java 7 by using the Threshold parameter

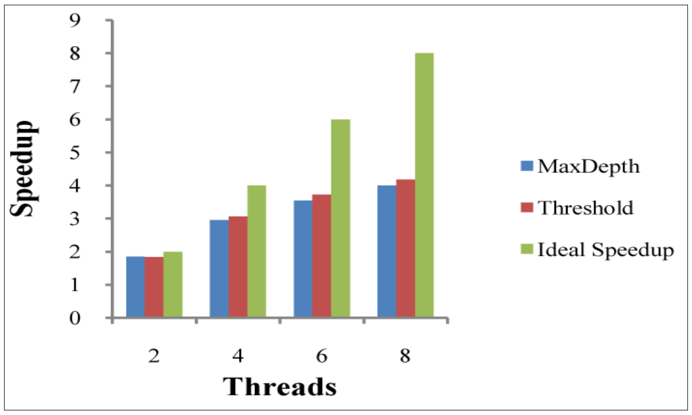

Figure 11. Comparing speedups of Mergesort by using Threshold and MaxDepth parameters

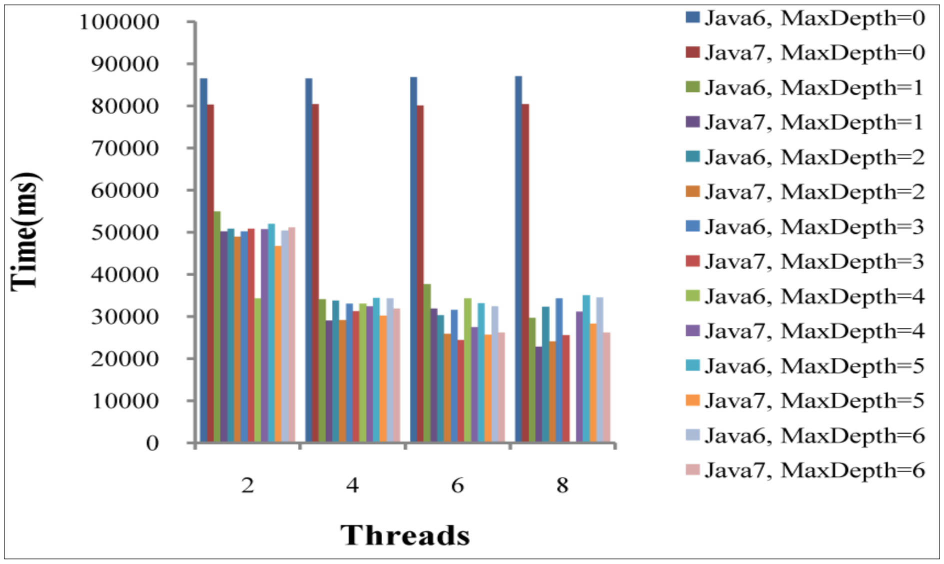

Figure 12. Comparing performances of Matrix Multiplication dealt with FJComp over Java 6 and Java 7 by using the MaxDepth parameter

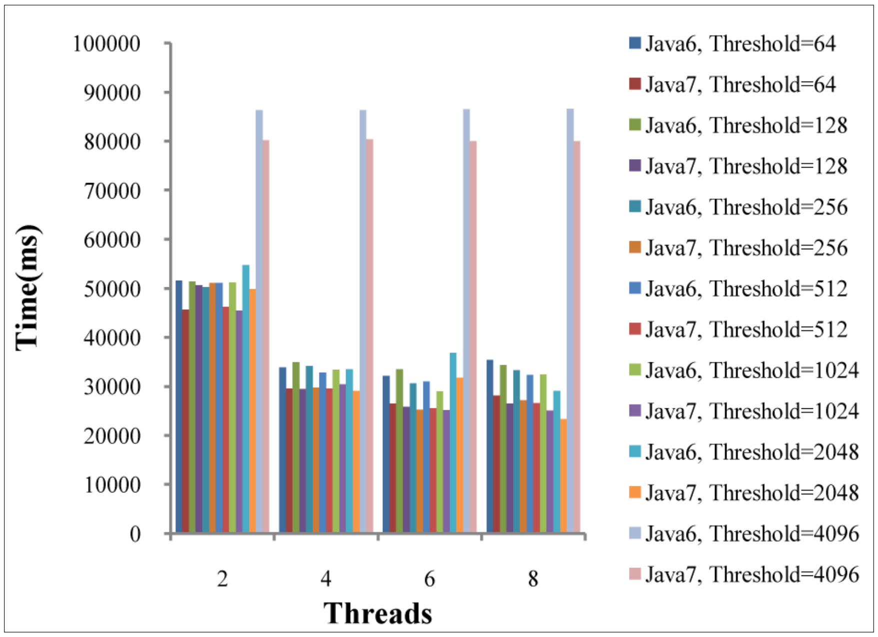

Figure 13. Comparing performances of Matrix Multiplication dealt with FJComp over Java 6 and Java 7 by using the Threshold parameter

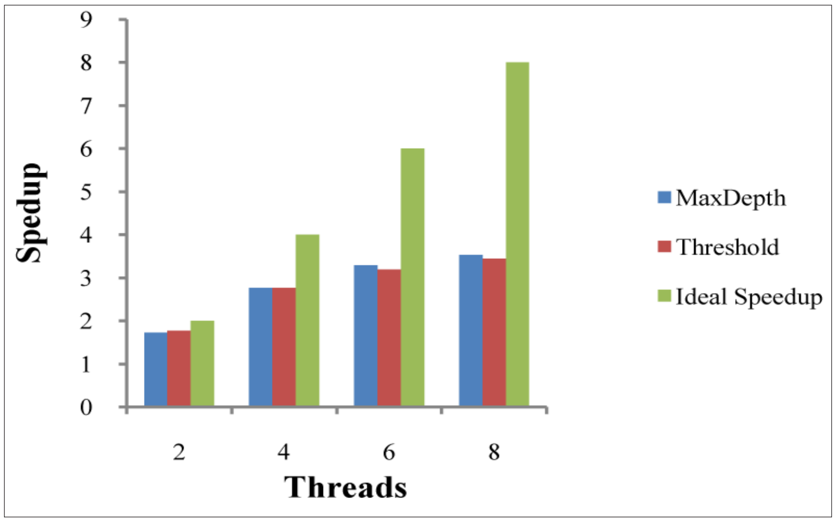

Figure 14. Comparing speedups of Matrix Multiplication by using Threshold and MaxDepth parameters

FJComp better performs over Java 6 than over Java 7 whatever the performance parameter. However, Figure 9, Figure 10 , Figure 12 and Figure 13 respectively corresponding to the execution of Mergesort and Matrix Multiplication indicate that dealing Mergesort and Matrix Multiplication with FJComp running over Java 7 gives best performances than running over Java 6.

4.3.2 MaxDepth Performance

Using MaxDepth parameter (Figure 1), Fibonacci reaches its best performances running over 2 processors with MaxDepth value equal to 5; running over 4, 6 and 8 processors with MaxDepth value equal to 10. Integrate (Figure 4) running over 2, 4, 6 and 8 processors with MaxDepth value equal to 15 give best perforemances. Quicksort (Figure 6) reaches its best performances when MaxdDepth is equal to 10 whatever the number of processors(2, 4, 6, 8). Mergesort (Figure 9) reaches its best performances when running over 2 processors with MaxDepth equal to 15; when running over 4 processors with MaxDepth equal to 1, when running over 6 and 8 processors with MaxDepth equal to 5. Matrix Muliplication(Figure 12) reaches its best performances when MaxDepth is of 5, 1, 3, 1 over respectively 2, 4, 6, 8 available processors.

4.3.3 Threshold Performance

Using Threshold parameter (Figure 2), Fibonacci reaches its best performances running over 2 processors with treshold value equal to 20; running over 4, 6 and 8 processors with Threshold value equal to 30. Quicksort (Figure 7) reaches its best performances when Threshold is equal to 100000 whatever the number of processors (2, 4, 6, 8). Mergesort (Figure 10) reaches its best performances when running over 2 and 4 processors with Threshold value is equal to 10000; when running over 6 processors with Threshold value equal to 1000000; when running over 8 processors with Threshold value equal to 100000. Matrix Multiplication (Figure 13) reaches its best performances running over 2, 4, 6, 8 processors with treshold value equal to 1024, 2048,1024, 2048 rescpectively.

4.3.4 MaxDepth versus Treshold Performances

According to our experiment results Integrate, Quicksort and Meregsort reach their best performance with Treshold parameter. Meanwhile Fibonacci and Matrix Multiplication best perform with MaxDepth parameter.

4.3.5 Speedup

Figure 3, Figure 8 and Figure 11 show that using Threshold parameter slightly gives best performances than using MaxDepth excepted Figure 14 where MaxDepth parameter slightly outperforms Treshold parameter.

4.3.4 Scale up

Figure 3 and Figure 5 show that Fibonacci and Integrate best scale up whatever the number of processors. But, Quicksort (Figure 8) and Mergesort (Figure 11) best scale up when number of processors vary from 2 to 4 and slightly scale up from 6 to 8 processors. Figure 14 shows that Matrix Multiplication slighty scales up from 4 to 8 processors.

The following table is a summary of the results of experiments and highlights the best performance of Fibonacci, Integrate, Quicksort, Mergesort and Matrix Multiplication according to the Java runtime (Java 6 or Java 7) and the performance parameter(MaxDepth or Threshold)

	Java6	Java7	MaxDepth	Threshold
Fibonacci	✓		✓	
Integrate	✓		✓	
QuickSort	✓			✓
Mergesort		✓		✓
Matrix Multi.		✓	✓	

Table 1. Performance Comparisons

The following table determines the various criterion of performance of Fork-Join framework. For each application, the dataset and how algorithm is load balanced are determined. However, how the garbage collector interacts with the application is an implementation-dependent.

	Garbage Collection: JVM	Memory Locality: Dataset	Task Locality: Load balancing
Fibonacci	?	Scalar	Good
Integrate	?	Scalar	Good
QuickSort	?	Array	-Depends on the choice of the pivot -Not Good for big tasks
Mergesort	?	Array	-Good for small tasks -Not Good for big tasks
Matrix Multi.	?	array	Good for small tasks Not Good for big tasks

Table 2. Performance Criterions of Fork-Join framework

According to Table 1, Fibonacci, Integrate and Quicksort best perform over Java 6 whereas Mergesort and Matrix Multiplication best perform over Java 7. If we refer to [5], garbage collection, memory locality, task synchronization and task locality determine the performance. Fibonacci and Integrate have scalar dataset (Table 2), that's why they don't suffer from the memory locality. They also well load balance, thus the synchronization, if any, does not impact the performance of those applications. All those reasons make those applications scalable.

Task locality depends on the ability of the program to well load balance. For instance Quicksort is an irregular algorithm in terms of load balancing; because it depends on the choice of the pivot. Mergesort and Matrix Multiplication well load balance the small tasks, but more the tasks are big and the more chance some worker threads remain idle while others are active. So, Mergesort, Quicksort and Mayrix Multiplication, not only their suffer from the load imbalancing but their suffer from the synchronization due to the fact that the idle worker try to steal works of active threads. Quicksort most suffers from the task locality and synchronization because work-stealing strategy is sometimes used. Furthermore Quicksort Meregesort, Matrix Multiplication suffer from memory locality and bandwidth because these algorithms handle array of data (Table 2) . The garbage collection is implementation-dependent on the virtual machine. From our experiment we cannot determine how Java 6 and Java 7 garbage collectors interact with our applications. Knowing that Quicksort, Mergesort and Matrix Multiplication suffer from task locality and memory locality, what makes the difference of performance between running over Java 6 and Java7? What makes the difference between Fibonacci, Integrate, Quickort (best performance with Java 6) and Mergesort and Matrix Multiplication (best performance with Java 7)? We can notice that Fibonacci and Integrate have scalar dataset and their best performances are with Java 6. Knowing that, these two applications well load balance and do not suffer from memory locality, we can observe that Java 6 garbage collector best fit than Java 7 garbage collector when they deal with scalar data. When dealing with array dataset, Java 7 best fits with Mergesort and Matrix Multiplication and Java 6 is most indicated for Quicksort. We can observe that, Mergesort and Matrix Multiplication are similar in a way they subdivide their tasks. According to our experimental results, most of time dealing applications with our compiler over Java 7 give bets performances than running over Java 6.

Using the performance parameters Threshold and MaxDepth is a trade-off between the simplicity and the performance. According to Table 1, MaxDepth outperforms Treshold when dealing with Fibonacci, Integrate and Matrix Multiplication and Treshold outperforms MaxDepth when dealing with Quicksort and Mergesort. Using MaxDepth is much easier than using Threshold. Notice that the experiments do not cover all the possible values of MaxDepth and Threshold. Our experiments only focus on ranges of values. Thus, to really know the difference of performance between MaxDepth and Threshold, exhaustive experiments covering all possible values must be done. These parameters allow defining fine-grain and coarse-grain parallelism. For instance, Mergesort and Matrix Multiplication using MaxDepth parameter soon reach their best performances. That means Mergesort and Matrix Multiplication relatively need coarse-grain parallelism rather than fine-grain. Unlike Mergesort, Quicksort needs more fine-grain parallelism than coarse-grain parallelism because of its irregularity in terms of load balancing.

5. Related work

In [5], Doug Lea designs and implements a Java framework for supporting divide-and-conquer program. This framework is easy to use and consists of splitting the task into independent subtasks via fork operation, and then joining all subtasks via a join operation. The performances of experiment show a performance gain. The performance mainly depends on garbage collection, memory locality, task synchronization, and task locality. The framework is made up of a pool of worker threads, a fork/join Task, and queues of tasks. The worker threads are standard ("*heavy*") threads. The fork/join tasks are lightweight executable class. The queues of task are made up of dqueue (supports both LIFO and FIFO). Each worker thread generates a new task via a fork operation and each thread has its own queue. The framework also uses work-stealing algorithm which consists of, from an empty queue of a worker thread, popping a task belonging to a non-empty queue of another worker thread. Both parameters that can impact the performances are the threshold and the number of worker threads. When the subtask size is smaller than the threshold, then that subtask is executed serially. But, sometimes it is very difficult to determine the threshold.

CONCURRENCER [3] is a tool that transforms sequential Java code into parallel program. From the sequential code Int variable-type, HashMap variable-type and recursion-like algorithm are transformed into AtomicInteger, ConcurrentHasmap and ForkJoinTask respectively. However, this tool is integrated within Eclipse's refactoring engine. In other words, this tool is not useful if the program operates with an IDE other than eclipse. In the recursion conversion to ForkJoinTask, the performance parameter the programmer must specify is the threshold. However, for some divide-and-conquer it is difficult to determine the threshold. In our compiler, FJComp, we introduce another parameter performance which is the MaxDepth, when reached, sequential algorithm is performed. Specifying the MaxDepth is easier than using the Threshold.

From version 3.0, OpenMP [8] introduces the task parallelism. Programmer uses constructs to identify the tasks to be parallelized. Task constructs are enclosed within the parallel construct. Tasking, as designed in OpenMP is a good candidate to parallelize recursive algorithm. In the context of recursive application, each invoked method is identified as task. Clauses allow managing the variables. The *if* clause, when its scalar expression evaluates to false, enables sequential execution. This is very important for the performance.

Javar [1] is not designed to be a fine-grain parallelism and it is implemented in a naïve way. The cut_depth is the performance parameter. If the cut_depth is too small, there is no overhead for the thread creation but a coarse-grain parallelism is achieved. When the cut_ depth is too high, the fine-grain parallelism is achieved but there is extra overhead in terms of thread creation and destruction.

Jomp [2] is an openMp like-interface for shared memory parallel programming. However, since then OpenMP has advanced version while Jomp did not develop. Jomp was designed to deal with loop-level parallelism and sometimes exploits parallelism when a method has several independent sections. A section is considered as an enclosed block of code within a method. To parallelize dive-and-conquer algorithm, Jomp uses section construct. This causes as many parallel region creations as recursion invocations. Given that creating and destroying thread need resource invocations (cpu, memory), Jomp is not suitable to deal with parallel recursive algorithm.

6. Conclusion

We have implemented and designed FJComp, a source-to-source parallelizing compiler that deals with Java divide-and-conquer algorithm. The compiler is made up of `//taskq` directives with optional clauses which are Threshold conditional expression and MaxDepth value. Although ForkJoin framework is easy to use, the programmer may sometimes be facing the error of coding. Our compiler is designed to be easier to use and les error-prone.

Experimental results show that dealing with our compiler gives performance gain. Comparison between Java 6 and Java 7 shows that, Java 6 when dealing with scalar dataset outperforms Java 7 and Java 7 when dealing with array dataset outperforms Java6 most of time. It is obvious that how Java 6 and Java 7 garbage collectors interact really differ whether we have scalar dataset or array dataset. Comparison between Threshold and MaxDepth shows that both parameters slightly yield same performances. However, choosing one of them is a trade-off between simplicity and performance. But, it is much easier to use MaxDepth.

Several future works may be planned. The first one is to combine Jomp with FJComp in order to obtain a unified compiler and in order to get better performance when the program to be parallelized both implement divide-and-conquer algorithm and loop-level parallelism. The second one is to improve the compiler by adding the variable clauses (private, shared, firstprivate). The third one is to do more studies and benchmarks in order to find out how Java 6 and Java 7 garbage collectors really interact with Fork-Join programs.

References

[1] Bik, A. J. C., Villacis, J. E., Gannon, D. B. (1997). javar: A Prototype Java Restructuring Compiler. *Concurrency: Practice and Experience*, 9 (11) 1181–1191,

[2] Bull, J. M., Kambites, M. E. (2000). JOMP—an OpenMP-like Interface for Java. *In:* Proc. of the ACM 2000 Java Grande Conference, p. 44-53. June.

[3] Dig, D., Marrero, J., Ernst. M. D. (2009). Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. *In*: Proc. of the 31st International Conference on Software Engineering.

[4] Freisleben, B., Kielmann, T. (1995). Automated transformation of sequential divide-and-conquer algorithms into parallel programs. *Computers and Artificial Intelligence*, 14, 579–596.

[5] Lea. D. (2000). A Java fork/join framework. *In*: JAVA'00, p. 36–43.

[6] Metama Inc. JavaCC—The Java Parser Generator. www.metama.com/JavaCC

[7] Oaks, S., Wong, H. (2004). Java Threads. *Understanding and Mastering Concurrent Programming.* Third Edition. O'Reilly & Associates, Sebastopol, CA.

[8] OpenMP, available at www.openmp.org.

[9] Rugina, R., Rinard, M. C. (1999) Automatic parallelization of divide and conquer algorithms. *In*: PPoPP '99, p. 72–83.