Back to the Futures

André Nicolau[1] and Catarina Gamboa[1]

Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa
{fc47880,fc49535}@alunos.fc.ul.pt

Abstract. Current computer architectures have the capacity of executing multiple processes at the same time allowing the execution of parallel tasks. However, most of the developed code, nowadays, is not written to take advantage of this architecture, once that would require the writing of concurrent programs by the developers which is an error-prone task that most developers don't master. In this study we purpose a source-to-source compiler that automatically generates a parallel version of a given recursive sequential program. Our study focuses on problems that implement a divide-and-conquer algorithm using recursion and are written in Java, but the base algorithm of the compiler can be used for almost any language. The evaluation of the parallel programs generated by the compiler, showed a reduction of the execution time of the programs in relation to their sequential version, as expected. This approach allows the developers to create a parallel, more efficient, version of their programs with minimum effort.

Keywords: Automatic Parallelization · Futures · Recursion.

1 Introduction

Nowadays all the processors in market have a very different architecture from the processors of twenty years ago. The main differences consist on the use of hyperthreading and multicore architectures that allow instructions to run in parallel. Hyperthreading consists in the execution of two or more threads inside a single CPU, while multicore is related to having two or more actual CPUs in one chip [11].
To have software as efficient as possible, we need to create applications that take the most advantage of the underlaying architecture of the system. This means that we need to write concurrent programs. However, developing parallel programs is not the simplest task to programmers, some are not used to the concurrent programming mental model, and to the ones that are it still is a very error-prone task.
To address this issue we created a source-to-source compiler (transpiler) that automatically parallelizes recursive procedures of sequential programs using futures. This approach minimizes the programmer's effort, that only needs to develop a sequential recursive version of the program. The programs that take advantage of this technique are the ones that employ a divide-and-conquer algorithm. In this type of algorithms, the problem is decomposed in several independent sub-problems that call the original procedure and, in the end, their results

are combined to reveal the solution to the original problem. We focused on this type of programs because they can have a simple parallel implementation once all the subprograms are independent [9].
We used futures to implement the parallelization of the recursive calls [5], once they offer a simple model of parallelism. A future acts as a placeholder for the result of an asynchronous computation [12]. By using the get primitive on the future object we retrieve the value of the computation when it terminates, which means that get can be a blocking operation. Having several futures running at the same time means that several computations are being processed in parallel.
The goals of this project were:

– To create an algorithm for the source-to-source compiler to parallelize a given sequential recursive code;
– Implement the previous algorithm;
– Make an experimental evaluation with known problems to understand the improvements of the parallel version against the sequential version.

2 Related Work

Many authors have studied different ways to automatically transform sequential code in parallel one, ensuring the same results in both versions. The idea of automatically parallelize programs, has gained popularity because programmers/developers have the possibility of creating more efficient programs by taking advantage of the processor's architecture. However, developing that code can be very expensive in terms of effort and many developers don't have the enough knowledge to do it.
Much of these studies result in new languages and frameworks. This languages have the capacity to improve, for instance, *for* cycles taking the advantage of the processor to improve the performance. One example of these languages is Chapel [2].
In terms of frameworks, their objective is to make easier the parallelization to the developers without using a specific language. However, different languages have different limitations in terms of use of threads. There are many examples of these frameworks, for instance Æminium framework [1], which analyses the sequential code and parallelizes everything that is possible and, based on the instructions, creates signatures that have information about dependencies and control flow. This approach makes possible the organization of the code in a task-based structure. One other framework that is important in Java, is the ForkJoin framework [7], which creates tasks that take the advantages of a Work-Stealing scheduler.
Specifically in Java, one of the most used programming languages, there are many studies about automatic parallelization. There are studies about runtime support for automatic parallelization[3], where the compile time and the runtime analysis is combined to make possible the detection of dependencies and exploit the parallelism among the threads in use. Other example is a contribution about parallelizing Java sequential code in terms of dependencies[4]. The

solution of this example is to infer dependencies between instructions based on extracted instruction signatures and to create tasks to be executed in parallel.
Also related to this article, there are many studies about recursion and the use of futures. Recursion is a very useful case to be parallelized because, in most of the cases, the same tasks are made many times. However, the dependencies of each task can be a problem. Gupta, Mukhopadhyay, and Sinha developed a method that detects the independence of multiple recursive calls inside a procedure, which is a useful information when we want to parallelize algorithms that follow a divide-and-conquer approach.
Futures [5] are a simple way to parallelize recursive algorithms. AutoFutures is a study that simplifies the manual parallelization process based on async calls, and its objective is to reduce the runtime execution time [6]. One study related to the automatic generation of futures [10], analyses the benefits of parallelizing using futures and if the results of this analyses are positive, then the future parallelized code will be generated from the sequential version.

3 Approach

We started by verifying some conditions on the recursive method to ensure that it could be parallelized without compromising the final result of the method. If any of the verifications fail then the program will not be parallelized, otherwise we will move to the modification of the original code to create its parallelized version. The following pseudo-code represents the trace lines of the used algorithm.

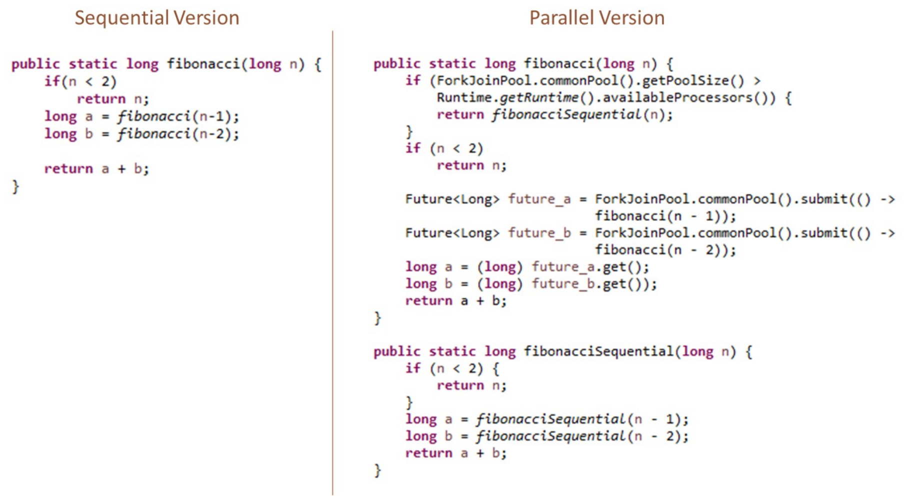
```
if arguments passed to the recursive function are
   of mutable types and there are writes in those arguments
   before the recursion call:
        return
if writes in any global variable:
        return
Create method with sequential version
Create futures on recursive method
Create get of futures , as far as possible of their creation
```

In the modification of the code, we first insert a new method that contains the sequential version of the recursive procedure, this version will be called when there are no more threads available to execute, which in large problems will certainly happen after some time. After this, we begin to create the code that parallelizes the problem. We change the recursion calls to be executed inside the futures and then look for the best place to retrieve those futures. To increase the parallelism of the program, the creation of the future task must be as far as possible from the corresponding get() operation, so that the process that called the futures will not block for a long time waiting for the future result.

Given this, we decided to execute the get operation just before the first time the value returned by the future is necessary. If the future does not return a value, we decided to execute the get operations immediately after the creation of all futures.

4 Implementation Details

We decided to implement the compiler in Java and focused on the compilation of Java programs. One of the biggest challenges was to analyse the source code and then modify it to create a correspondent parallel version. This task was accomplished with the usage of the Spoon library [8]. This library allows the developers access to the Java AST (Abstract Syntax Tree) to be possible to analyse and edit the source code (removing or/and adding nodes to the tree).
Inside Spoon we used templates to perform the necessary code transformations. The templates are simple Java classes that can be taken as input by Spoon to perform a transformation on the code.
To add futures to the result code, we used the Future Java interface materialized as ForkJoinTasks that execute inside a ForkJoinPool. The ForkJoinPool is used with the goal of getting all the available processing power to enhance the performance of the program.



```
public static long fibonacci(long n) {
    if(n < 2)
        return n;
    long a = fibonacci(n-1);
    long b = fibonacci(n-2);

    return a + b;
}
```

```
public static long fibonacci(long n) {
    if (ForkJoinPool.commonPool().getPoolSize() >
        Runtime.getRuntime().availableProcessors()) {
        return fibonacciSequential(n);
    }
    if (n < 2)
        return n;

    Future<Long> future_a = ForkJoinPool.commonPool().submit(() ->
                            fibonacci(n - 1));
    Future<Long> future_b = ForkJoinPool.commonPool().submit(() ->
                            fibonacci(n - 2));
    long a = (long) future_a.get();
    long b = (long) future_b.get());
    return a + b;
}

public static long fibonacciSequential(long n) {
    if (n < 2) {
        return n;
    }
    long a = fibonacciSequential(n - 1);
    long b = fibonacciSequential(n - 2);
    return a + b;
}
```

Fig. 1. Example of sequential version and produced parallel version of the program

The Figure 1 exemplifies both the changes made to the source program to achieve the parallelism, in this case we see side-by-side both the sequential and parallel versions of Fibonacci problem.
The future creation and its corresponding get operation should be as far apart

as possible to increase the parallelization, once the current process will also be executing its tasks instead of being only waiting for the answer of the future's computation. We decided to keep the creation of the future on the exact place where the recursive call was made and find a place to insert the get operation. If the recursive call stored the return value in a variable, we find the first place, after the creation of the future, where that variable is used and introduce the get operation just before. If the recursive call does not store the return value, the safer approach is taken, of calling the get operations just after all the futures creation.

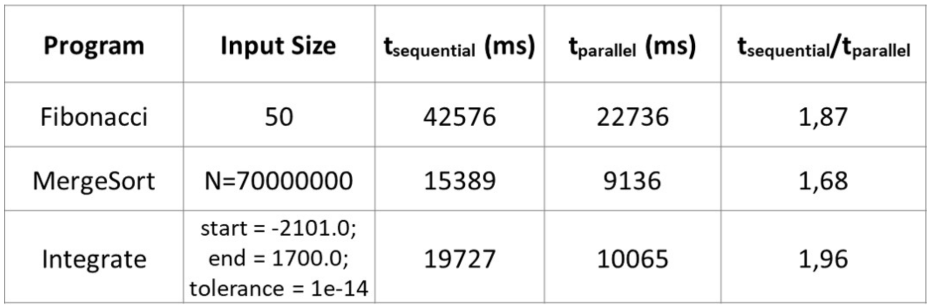
5 Evaluation

5.1 Experimental Setup

Operating System	macOS Catalina 10.15.2 64bits
Processor	2,3 GHz Quad-Core Intel Core i5 8 hyper-threaded threads
Memory	8Gb

5.2 Results

To analyse the parallelization benefits of using our source-to-source compiler, we chose some well-known problems as input and measured the execution time of their sequential and parallel versions. All the programs were executed with large inputs. If the inputs are too small, the overhead that the creation of futures brings makes the parallel version slower than the sequential version. This issue is related to the future work that we purpose on Section 6.
To measure the execution time of the different programs, we executed each version three times and registered the average time in milliseconds. For this we used the Java method System.currentTimeMillis(). The results are presented in Figure 2 and Figure 3. It is relevant to say that the computer where the measurements were made, was also executing other tasks, which means that the results are only comparable within this study.



Program	Input Size	$t_{sequential}$ (ms)	$t_{parallel}$ (ms)	$t_{sequential}/t_{parallel}$
Fibonacci	50	42576	22736	1,87
MergeSort	N=70000000	15389	9136	1,68
Integrate	start = -2101.0; end = 1700.0; tolerance = 1e-14	19727	10065	1,96

Fig. 2. Table of evaluation values

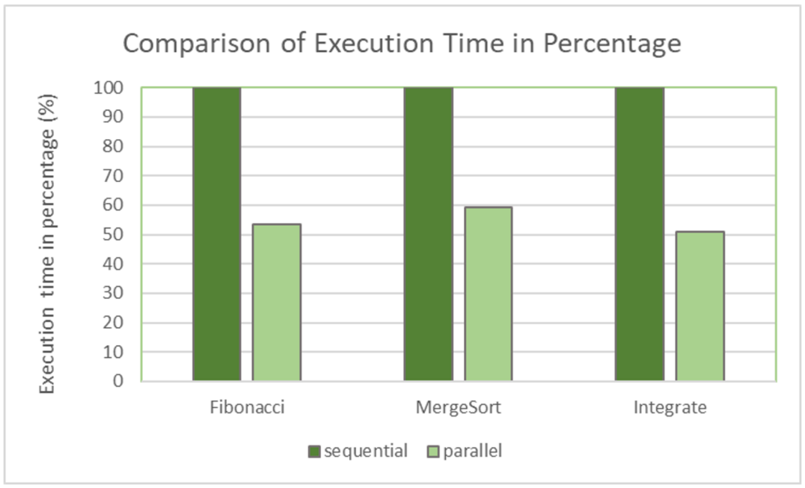


Fig. 3. Graphic with execution time comparison of sequential and parallel versions of each of the analysed problems

5.3 Discussion

As expected, the results of the parallel version show an improvement on the execution time for each of the programs. The speedup goes from 1,68 to almost 2 times depending on the problem executed.
In Figure 3 we show the behaviour of the several parallel programs in contrast with their sequential version, where the execution time of the parallel version is represented as a percentage of the execution time of the sequential version. In other words, the sequential version represents the maximum percentage (100%) and the result of the parallel version is calculated based on that value. In this graphic we can see that the improvements on the different problems are similar, running, in average, in 53,4% of the time of the sequential version, and so, having an average speedup of 47,6%.
The differences between the execution times of parallel versions of the evaluated problems are related to the algorithms implemented in each of the programs and the success of the parallelization procedure. The program that improved the most with the parallelization was the Integrate because each created future has a significant amount of work that is executed asynchronously, as opposed to the other two programs that contain the recursive call almost at the beginning of the method and so having a small quantity of processing to execute before introducing another asynchronous call (create future).

6 Conclusions

In this study we have purposed a source-to-source compiler that automatically parallelizes recursive programs that follow a divide-and-conquer algorithm. This compiler allows the programmer to create a parallel program with minimum effort, being only necessary to develop its sequential version. We evaluated the execution time gains of the parallel versions, created by the compiler, against their sequential version and the results show an average speedup of 1,84 times. The parallel version of these programs executes in an average of 53,4% of the time of the sequential version. These results are encouraging but we know that they can be improved if other metrics are added to the parallelization algorithm of the compiler. For future work we intend to add some granularity control mechanisms that decide if the problem is large enough to continue the parallelization or, otherwise, to continue its execution on the sequential version. Our verification methods for the safety of parallelization are also very exclusive, opting for the safer approach of aborting the parallelization if some of the verifications fail. In future work we also want to improve these verifications allowing the parallelization of more programs. This study was centred on the parallelization of recursive calls, in the future we would like to extend our studies to other code parts that can be parallelized such as for and while cicles.

References

1. A.Fonseca, B.Cabral, J.R., Correia, I.: Automatic parallelization: Executing sequential programs on a task-based parallel runtime. International Journal of Parallel Programming (2016)
2. B. L. Chamberlain, D.C., Zima, H.P.: Parallel programmability and the chapel language. The International Journal of High Performance Computing Applications **21**(3), 291–312 (2007)
3. Chan, B., T.S.Abdelrahman: Run-time support for the automatic parallelization of java programs. The Journal of Supercomputing **28**, 91–117 (2004)
4. J. Rafael, I. Correia, A.F., Cabral, B.: Dependency-based automatic parallelization of java applications. Euro-Par 2014: Parallel Processing Workshops p. 182–193 (2014)
5. J. Swaine, K. Tew, P.D.R.B.F., Flatt, M.: Back to the futures: Incremental parallelization of existing sequential runtime systems. ACM Sigplan Notices (2010)
6. K. Molitorisz, J.S., Otto, F.: Automatic parallelization using autofutures. Springer (2012)
7. Lea, D.: A java fork/join framework. Proceedings of the ACM 2000 conference on Java Grande (2000)
8. R. Pawlak, M. Monperrus, N.P.C.N., Seinturier, L.: Spoon: A library for implementing analyses and transformations of java source code. Wiley Online Library (2015)
9. Smith, D.R.: The design of divide and conquer algorithms. Science of Computer Programming p. 37–58 (1985)
10. Surendran, R., Sarkar, V.: Automatic parallelization of pure method calls via conditional future synthesis. ACM SIGPLAN Notices (2016)
11. Sutter, H.: The free lunch is over. Dr. Dobb's Journal (2005)
12. T. Cogumbreiro, R. Surendran, F.M.V.S.V.T.V., Grossman, M.: Deadlock avoidance in parallel programs with futures: Why parallel tasks should not wait for strangers. ACM Program, Article 103 (2017)