JPT: a Java Parallelization Tool

Kristof Beyls

1st March 1999

Abstract

A Java parallelization tool, JPT is presented. In contrast with other approaches, which focus on adding user-defined parallelization directives to a Java program, JPT detects parallelism automatically. The development is based on accessing the internal parallelization kernel of the open compiler, FPT. As a result, dependencies in loops are detected and parallel loops are converted into multithreaded Java code. The results illustrate the feasibility of this approach and indicate that good speedups can be obtained.

1 Introduction

The importance of Java as a coherent, platform independent, object-oriented and network-minded language is not disputed. Given these features, it is not surprising that Java has also entered the high performance computing area with projects such as HPJava[2], jPVM[4], Java/DSM[9], Spar[8], JAVAR[1], ...

These extensions/libraries all add the possibility to express parallelism in an elegant way. Some of these tools generate in 100% standard Java, while others use platform dependent libraries, or even specific Java Virtual Machines [7].

From these papers it appears that there are many ways to add parallel directives to the Java programming language. However, only a few authors[1] investigate the way to *automatically* detect parallel executable regions in a Java program, and to generate parallel code from this analysis. This can be attributed to the complexity of parallelization, and to the fact that parallelization tools were mainly developed for other languages [3]. In this paper, the kernel parallelization algorithms and internal syntax tree of FPT[10] is applied to detect parallelism in Java loops. This work was simplified because of the open structure of the FPT with regard to new algorithms and languages. As a result, a rigorous dependence analysis is performed on the Java program and the parallelism is revealed by the FPT-analyzer. In a second phase, multi-threaded Java code is generated, based on the back-end code generator of JAVAR[1]. The resulting code is able to execute in parallel on a multiprocessor which executes the Java threads in parallel, e.g. on the Sparc, Windows-NT and Linux platforms running the JDK1.7 development kit.

`def_ref`	finds out for every node in the AST which variables are read from and written into in that part of the program.
`create_depgraph`	creates a dependency graph and stores it in the AST
`doall_converted`	parallelizes all loops in the AST, subject to the dependence analysis

Table 1: The functions that are used to identify the parallel loops in the AST.

This paper addresses the automatic generation of parallelism starting from "sequential Java" source code. Related work has been done by Aart J.C. Bik and Dennis B. Gannon, who made JAVAR [1] which generates thread based parallel Java source code from sequential source code.

2 Java loop parallelization

Loops are traditionally areas of implicit parallelism. The parallel execution of loops is subject to a non trivial analysis of the loop-carried dependencies. Dependency analysis has matured over time and the most important dependence analysis algorithms have been put into the Fortran parallelizer, FPT [11], which has been developed at the University of Ghent.

2.1 Dependence Analysis

By design[10], the inner data structures and the abstract syntax tree of FPT are quite language independent. As a consequence, the same dependence analysis can be applied to any language that can be expressed in the FPT syntax tree. Furthermore, the FPT API (application programming interface) offers the right tools to detect, annotate and retrieve the parallelism. The effort to make an interface to FPT for another language than Fortran was done previously for C, leading to CPT (C parallel transformer[5]).

The dependence analysis of FPT uses techniques such as Banerjee, Wolfe and GCD tests[6], loop boundary calculation and unimodular transformations [11].

By using functions from FPT's API, one can construct an FPT AST. The API also contains the necessary functions to detect parallelism in the loops (see Table 1).

2.2 Parsing Java

The Java source is parsed by the GNU compiler `guavac` into a complete Java-based abstract syntax tree. Since FPT was developed for Fortran, obviously some Java language

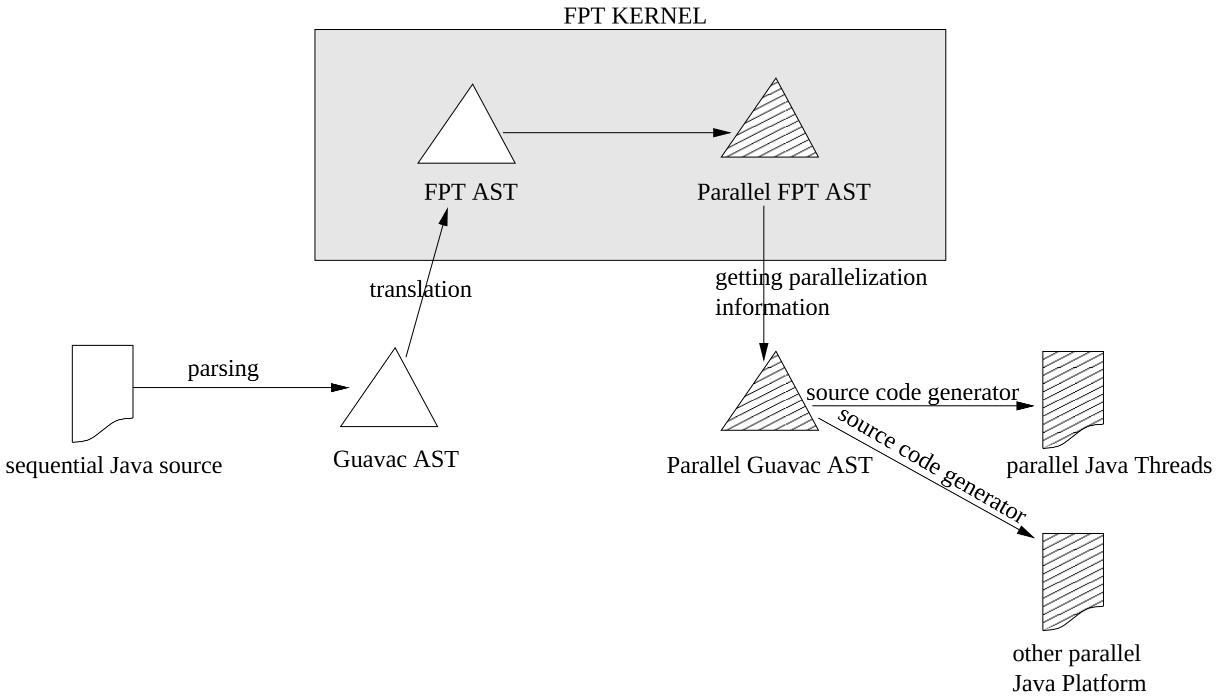

Figure 1: JPT parsing, parallelization and code generation.
The source file is parsed into an AST by Guavac. The loop nests are forwarded to FPT. After parallelization by FPT, the parallel loops are annotated in the Guavac AST. Finally, code can be generated for different parallel Java platforms.

constructs cannot be represented by the abstract syntax tree of FPT. However, the computation intensive parts, most amenable to parallelization, are represented similarly in both languages, i.e. by loops and array calculations. As a consequence, only a part of the Guavac AST is feeded into the AST of FPT.

2.3 JPT parsing, parallelization and code generation

- The source file is parsed into a Guavac AST.
- JPT translates nodes in the Guavac AST bottom-up into semantically equivalent FPT syntax trees. If a node in the Guavac AST is not translatable into an FPT AST, then all the subtrees of the Guavac AST which contain this untranslatable node are also untranslatable, after which JPT continues with starting to translate another part of the Guavac AST. As soon as an outermost for-loop that can be translated has been discovered by JPT, the translation is feed into the parallelizer of FPT (see Table 1).
- The resulting parallelized FPT AST is traversed to see which loops were parallelized

by FPT, and the corresponding loops in the Guavac AST are marked as parallelizable.

2.4 Parallel Java code generation

Currently, JPT generates parallel source code as standard Java Threads. This code production reuses the technique described in [1].

3 Results

JPT was tested on programs with non-homogeneous loops including if-tests, such as linear systems (e.g. BLAS), matrix multiplication etc.

As an example the Gauss-Jordan elimination algorithm is parallelized as follows (the directive `/* Par */` indicates that the following for-loop may be executed in parallel. On the left side is the original Java function, on the right the function parallelized by JPT):

```
void eliminate(float[][] a, int n)
{
  double f;

  for(int i=0;i<n;i++) {

    for(int j=0;j<n;j++) {
      if (i!=j) {
        f=a[j][i]/a[i][i];

        for(int k=i+1;k<n+1;k++)
          a[j][k] -= f*a[i][k];
      }
    }
  }
}
```

```
void eliminate(float[][] a, int n)
{
  double f;

  for(int i=0;i<n;i++) {
    /* Par */
    for(int j=0;j<n;j++) {
      if (i!=j) {
        f=a[j][i]/a[i][i];
        /* Par */
        for(int k=i+1;k<n+1;k++)
          a[j][k] -= f*a[i][k];
      }
    }
  }
}
```

4 Conclusion.

Java is widespread and becomes to have its impact in the High Performance community, provided it contains an easy-to-use parallelization package. This paper is an additional step in this direction.

JPT is able to find the parallelism in computation-intensive Java loops, from which efficient executable code can be generated for a number of different High Performance Java platforms.

References

[1] A. J. C. Bik, J. E. Villacis, and D. B. Gannon. javar: a prototype Java restructuring compiler. *Concurrency, Pract. Exp. (UK), Concurrency: Practice and Experience*, 9(11):1181–1191, Nov. 1997.

[2] B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen. Hpjava: Data parallel extensions to java. *Concurrency: Practice and Experience*, 10(11-13):873–877, 1998.

[3] E. D'Hollander, F. Zhang, and Q. Wang. The fortran parallel transformer and its programming environment. *Journal of Information Sciences*, 106(7):293–317, July 1998.

[4] A. J. Ferrari. JPVM: Network parallel computing in java. In *Proceedings of the ACM Workshop on Java for High-Performance Network Computing*, Mar. 1998.

[5] L.-S. Peng. Implementatie van een c-interface voor de parallelliserende compiler fpt. Master's thesis, University of Ghent, 1998.

[6] K. Psarris. The Banerjee-Wolfe and GCD tests on exact data dependence information. *Journal of Parallel and Distributed Computing*, 32(2):119–138, Feb. 1996.

[7] Sun Microsystems. *The Java Virtual Machine Specification*, 1.0 beta edition, Aug. 1995.

[8] K. van Reeuwijk, A. J. van Gemund, and H. J. Sips. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, Nov. 1997.

[9] W. M. Yu and A. L. Cox. Java/DSM: a platform for heterogeneous computing. In *Proc. of Java for Computational Science and Engineering–Simulation and Modeling Conf.*, pages 1213–1224, June 1997.

[10] F. Zhang. Development of a program information base for the FPT programming environment. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and D. Trystram, editors, *Parallel Computing: State-of-the-Art and Perspectives, Proceedings of the Conference ParCo'95, 19-22 September 1995, Ghent, Belgium*, volume 11 of *Advances in Parallel Computing*, pages 223–230, Amsterdam, Feb. 1996. Elsevier, North-Holland.

[11] F. Zhang. *The FPT Parallel Programming Environment*. PhD thesis, University of Ghent, 1996.