JPT: a Java Parallelization Tool

Kristof Beyls[1], Erik D'Hollander[2], and Yijun Yu[3]

[1] Kristof.Beyls@rug.ac.be
[2] Erik.DHollander@elis.rug.ac.be
[3] Yijun.Yu@elis.rug.ac.be
University of Ghent,
Department of Electrical Engineering
Parallel Information Systems
Sint-Pietersnieuwstraat 41
B-9000 Gent, Belgium

Abstract. PVM is a succesfull programming environment for distributed computing in the languages C and Fortran. Recently several implementations of PVM for Java have been added, making PVM programming accessible to the Java community.
With PVM for Java however, the user still needs to partition the problem, calculate the data partitioning and program the message passing and synchronization. In this paper, JPT is introduced, a parallelization tool which generates PVM code from a serial Java program. JPT automatically detects parallel loops and generates master and slave PVM programs.

1 Introduction

The importance of Java as a coherent, platform independent, object-oriented and network-minded language is widely recognized. With 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[10], Spar[8], JAVAR[1].

Most of these projects study various ways to achieve a faster execution of Java programs by efficiently expressing the parallelism in the language. 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 we focus on the automatic parallelization and efficient code generation of Java programs. Rather than reimplementing a parallelizing compiler, the kernel parallelization algorithms and the internal syntax tree of an existing compiler, FPT[3], are reused to automatically detect parallelism in Java loops. As a result, the dependence analysis needed to reveal the parallelism in a Java program is executed by the FPT-analyzer. Once the parallel loops in the program are detected, they are transformed into an explicit parallel form by JPT. Currently, JPT generates code for 2 Java parallel platforms: parallel Java threads and jPVM.

A description of the existing parallel virtual machines for Java is given in sect. 2. The automatic parallelization of loops in the FPT parallelizer is discussed in section 3. An operational overview of JPT is given in sect. 4. The code generation is explained in sect. 5. Finally, experiments and their speedup are presented sect. 6, after which a conclusion is formulated in sect. 7.

2 Parallel Virtual Machines in Java

In the literature, there are two approaches to develop a Java based Parallel Virtual Machine: either rewrite PVM in Java[4], or write a Java-interface to the existing PVM API[7] .

1. jPVM[7] is layered upon the standard distribution of PVM and makes use of Java's capability to call functions written in other languages using the Java Native Interface[5]. jPVM programs use wrappers contained in the class jPVM to call the *native* PVM functions, which are written in C.
2. JPVM[4] on the other hand is entirely implemented in Java and uses none of the original PVM code. JPVM provides an interface similar to the C interface provided by PVM, but with a syntax and semantics adapted to Java threads and the Java programming style. Unlike jPVM, JPVM is not inter-operable with standard PVM. JPVM provides a Java implementation of the PVM daemon and a communications library. In addition, both tasks and threads are supported as basic units of parallelism.

In [9] different benchmarks were executed to test the communication performance of JPVM, jPVM and PVM. This showed jPVM to be faster than JPVM and C/PVM to be faster than jPVM. Our JPT is able to generate parallel code for the jPVM platform.

3 Loop Parallelization

3.1 Data Dependence Analysis

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], the backbone of JPT. FPT uses techniques derived from Banerjee, Wolfe and the GCD tests[6], loop boundary calculation and unimodular transformations[11].

By design, the inner data structures and the abstract syntax tree (AST) of FPT are 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 offers tools to detect, annotate and retrieve the parallelism.

3.2 Loop Scheduling

PVM code[11] for the outermost of a nest of parallel loops is obtained by generating slave programs, which each execute a group of the n iterations as one task. If there are p slaves, then n/p iterations are assigned to each slave. Besides initializing the PVM-system and contacting the number of cooperating processors, for each parallel loop the following code is generated:

- In the *prologue*, the input data for all the parallel loop is gathered and put into a single message, which is broadcast to all slave computers. Next the number of iterations to be executed by each slave is calculated and included in the message.
- In the *execution* phase, each slave program unpacks the message and executes his band of the loop. During this phase there is no communication, because the inner loop iterations are independent.
- In the *epilogue*, each slave sends back the results. The master will restore the received data in the proper locations.

3.3 Data Partitioning

After the dependence analysis, FPT determines the data to be exchanged between processors by looking for the data references to the left and the right side of assignment statements in the loop body. However, using this technique without optimization could resolve in unnecessary communication overhead. FPT uses 2 techniques to reduce the overhead:

1. If consecutive elements of an array have to be sent, then FPT will create a single *pk*-call to pack the data, instead of creating a loop in which the elements are packed one at a time.
2. The references to data in the loop might overlap. When array subscripts are of the form $ai + c$, i being the loop index and a and c are constants, FPT identifies overlapping areas and sends them only once.

When a parallel loop is part of a surrounding sequential loop, then the communication can be further optimized by *array privatization.* This technique is not yet implemented, but should work as follows. A parallel loop nest which is nested inside a sequential loop nest can be formally expressed as in fig. 1. Following sets must be calculated to create the messages between master and slaves:

- $W(i,j)$ is the set of all writes in $H(i,j)$.
- $R(i,j)$ is the set of all reads in $H(i,j)$.
- $WR(i,j)$ is the set of all reads that read a value created by a write in the same iteration.
- $E(i,j) = R(i,j) - WR(i,j)$ contains all variables and array elements that can influence the results of the parallel `j` loop.

```
DO i=...
  DOALL j=...
    H(i,j)
  ENDDO
ENDDO
```

Fig. 1. A parallel loop inside a sequential loop

The data to be sent between iteration i and $i+1$ of the sequential loop from task p_1 to task p_2 would then be $S(i \rightarrow i+1, p_1, p_2) = W_{p_1}(i) \cap E_{p_2}(i+1)$, with $W_{p_1}(i) = \bigcup_{j \in I(p_1)} W(i,j)$ and $E_{p_2}(i) = \bigcup_{j \in I(p_2)} E(i,j)$ where $I(p)$ is the set of iterations to be executed by task p.

4 JPT Operational Overview

The conversion of a Java program into an FPT AST and the parallelism extraction occurs in four steps (see Fig.2):

1. The Java source is *parsed* using the GNU compiler `guavac` into a complete Java-based abstract syntax tree (in this paper further called a Guavac AST). Since FPT was developed for Fortran, obviously some Java language 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 transformed into an FPT AST.
2. JPT *translates* the parts in the Guavac AST that are expressible in FPT and feeds them one by one into the *parallelizer* of FPT.
3. 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. The FPT parallelizer also generates the messages to be sent between master and slaves.
4. explicit *parallel code is generated* from the annotated Guavac AST. Currently JPT generates parallel code based on
   (a) Java threads,
   (b) jPVM.

5 Code Generation

JPT transforms the original program into an explicitly parallel PVM program by replacing each parallel loop by a master loop which calls a number of slaves and fetches the results.

A separate class is inserted to contain the slave code. This class extends the `JPT.JptPvmSlave` class (see Fig. 3(a)) which implements the job scheduling code common to all slaves. A `run` method is created in the new class. The loop specific slave code generated in the next steps will be inserted, in this `run` method. The interaction between the master and slaves is as follows:

1. The *master* spawns the slaves using the method `spawn_slaves` (see Fig. 3(b)). The `spawn_slaves` method spawns a new Java Virtual Machine for each slave, after which it sends the job information to the newly spawned slaves.

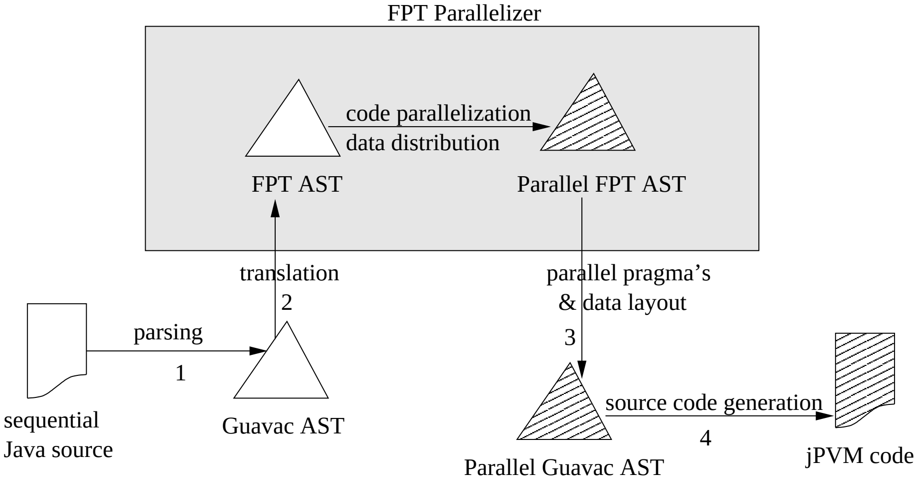


Fig. 2. JPT Parsing, Parallelization and Code Generation. The source file is parsed into an Abstract Syntax Tree by Guavac. The loop nests are forwarded to FPT. After parallelization by FPT, the parallel loops are annotated in the Guavac AST. Finally, the code can be generated for different parallel Java platforms.

2. Each JVM will execute one *slave*. The spawned JVM knows which slave to execute because the name of the slave class is passed as a command line argument. The spawned JVM will execute the `main` method of the `JptPvmSlave` class (see Fig. 3(a)). The `main` method creates an object of the slave class, and lets the slave execute in a separate thread. When the slave thread stops, the JVM ends executing. Creating an object of the slave class forces the constructor (starting at `/* 2 */` in Fig. 3(a)) to be executed. This constructor receives the job information sent in the `spawn_slave` method at `/* 1 */`.
   Each task is run in a separate JVM because PVM and therefore jPVM are not thread-safe (see section 2).
3. After spawning the slaves, the *master* code packs the sizes of all the arrays in the parallel loop as well as the input data for all iterations of the loop into a single message and multicasts it to all slaves.
   The slave code (which is put in the `run()` method) receives this message, creates the arrays, and executes his band of the loop.
4. The *slave* sends the output data for that band to the master.

6 Results

The performance of the code generated by JPT was measured using a matrix multiplication and a Gauss-Jordan linear system solver. The tests were performed on 4 Pentium-II machines interconnected with a 100 Mb/s Ethernet running Linux as operating system and using JDK 1.1.7 as the Java Virtual Machine.

```
package JPT;
import jPVM;

public abstract class JptPvmSlave
  implements Runnable
{
  public int mytid=-1;
  public int parent_tid=-1;
  public int nslaves=-1;
  public int[] tids;
  public int slavenum=-1;
  public int band=0;

  /* 2: job information is received */
  // constructor initializes job
  public JptPvmSlave()
    {
      mytid=jPVM.mytid();
      parent_tid=jPVM.parent();

      // PVM -> Java data receive
      jPVM.recv(parent_tid,STARTUP_MESG);
      int[] nslaves_band=new int[2];
      jPVM.upkint(nslaves_band,2,1);
      nslaves=nslaves_band[0];
      band=nslaves_band[1];
      tids=new int[nslaves];
      jPVM.upkint(tids,nslaves,1);
      for(int i=0; i<tids.length; i++)
        if (tids[i]==mytid) {
            slavenum=i;
            break;
          }
    }

  // slave entry point
  public static void main(String[] args)
  throws ClassNotFoundException,
         InstantiationException,
         IllegalAccessException,
         InterruptedException
    {
      Class thisClass =
        Class.forName(args[0]);
      JptPvmSlave thisClassInstance =
        (JptPvmSlave)
        thisClass.newInstance();
      Thread t = new
        Thread(thisClassInstance);
      t.start();
      t.join();
    }
}
```

(a) The base class JptPvmSlave in Java package JPT

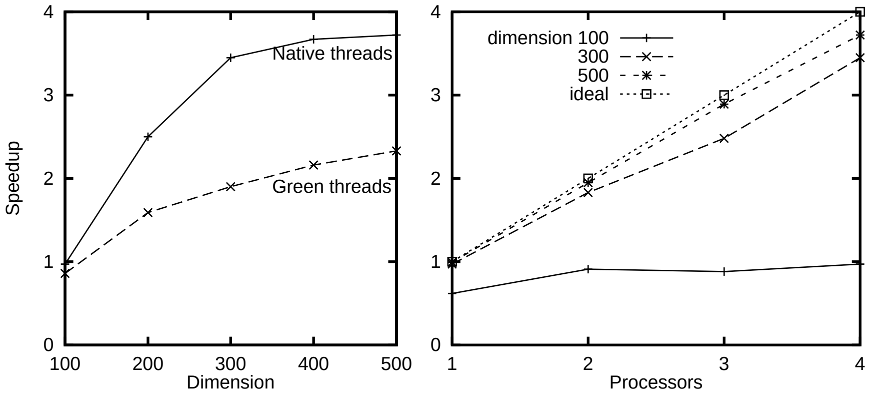
```
public static int spawn_slaves
  (int[] tids, String slavename, int band)
  {
    final int nrequested = tids.length;
    int mytid = jPVM.mytid();

    String[] args=new String[2];
    args[0]=slavename;
    args[1]=slavename;
    // spawn slaves
    if (tids.length >
        jPVM.spawn("java", args,
                   jPVM.PvmDataDefault,
                   "",tids))
      {
        System.err.println(
         "in JPT.JptPvmSlave.spawn_slaves"
         +"(int[] tids): couldn't spawn "
         +"the number of tasks requested"
       );
       jPVM.exit();
       System.exit(-2);
      }

    /* 1: Job information is sent */
    // Java -> PVM data send
    jPVM.initsend(jPVM.PvmDataDefault);
    int[] ntasks_band=new int[2];
    ntasks_band[0]=tids.length;
    ntasks_band[1]=band;
    jPVM.pkint(ntasks_band,2,1);
    jPVM.pkint(tids,tids.length,1);
    jPVM.mcast(tids,STARTUP_MESG);
    return mytid;
  }
```

(b) The `spawn_slaves` method

Fig. 3. The JPT package



(a) Comparison between green and native threads

(b) speedup vs. processors

Fig. 4. Speedups of the matrix multiplication algorithm

First the parallelized matrix multiplication algorithm was executed using "green" threads, i.e. threads scheduled within the Java Virtual machine. In this implementation the operating system executes the JVM as a whole, and has no grip on load balancing the individual threads. As a result, some long idle periods during the inter process communication were observed, affecting the overall speedup. In a second experiment, native threads where used, i.e. the Java threads are visible as individual threads to the operating system. This resulted in a better scheduling and good communication and speedup figures. Figure 4(a) depicts the speedup difference between native and green threads. Using native threads, the speedup has been measured for 1 to 4 processors, yielding a fairly linear speedup for a dimension > 300. (see fig. 4(b)).

We further tested the results on the Gauss-Jordan Elimination algorithm. We found that the data distribution as calculated by FPT creates excessive data communication. When this algorithm is transformed using array privatization, and the data communication is done as resulted form the proposed technique in sect. 3.3, good speedups were found, as seen in fig. 5.

7 Conclusion

The Java parallelizer JPT facilitates the use of Java in a PVM environment. The results show that loops can be parallelized using standard techniques embedded in the open compiler FPT, including data distribution and message coding. Further work is aimed at reducing the communication overhead by a sophisticated array privatization analysis. The speedups obtained indicate that Java is a viable language for parallel computing in a platform independent PVM-network.

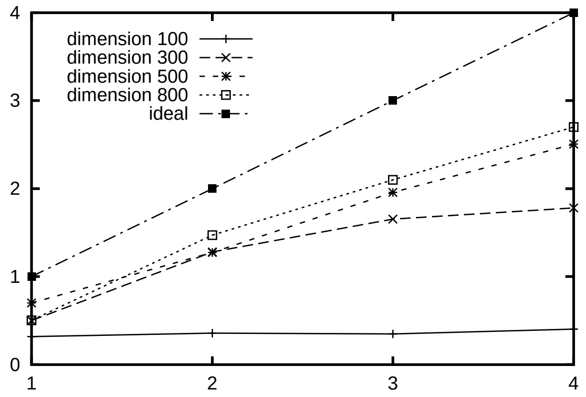


Fig. 5. Speedup of the optimized Gauss-Jordan linear system solver.

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] JavaSoft. Java native interface specification, Nov. 1996. Release 1.1.

[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] D. Thurman. jPVM. http://www.isye.gatech.edu/chmsr/jPVM/.

[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] N. Yalamanchilli and W. Cohen. Communication performance of java based parallel virtual machines. *Concurrency: Practice and Experience.*

[10] 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.

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