This paper has been submitted to HPCS'98, the 12th Annual Int. Symposium on High Performance Computing Systems and Applications, May 1998, Edmonton, Alberta, Canada.

Data Parallelism in Java

Michael Philippsen
Computer Science Dept., University of Karlsruhe, PO-Box 6980, 76128 Karlsruhe, Germany
Tel: +49/721/608-4067, Fax: +49/721/7343, eMail: phlipp@ira.uka.de

Abstract

Java supports threads and remote method invocation but it does neither support data parallel nor distributed programming. This paper discusses Java's shortcomings with respect to data parallel programming. It then presents countermeasures that allow for data parallel programming in Java. The technical contributions of this paper are twofold: a source-to-source transformation is presented that maps forall statements into efficient multi-threaded Java code. In addition, an optimization strategy is presented that achieves a minimal number of synchronization barriers.

The transformation, the optimization, and a distributed runtime system have been implemented in the JavaParty environment. In JavaParty, code compiled with current just-in-time compilers runs merely a factor of two to three slower than Fortran, on both a shared-memory parallel machine (IBM SP/2). Furthermore, better compiler technology is on the horizon, which will narrow the performance gap.

1 Introduction

Java is adopted by application programmers so quickly because of, among other reasons, its portability. Similarly, Java's portability is one of the key issues that might make Java one of the main languages for scientific programming as well. Unfortunately, Java has neither been designed for parallel programming nor for distributed programming. Hence, it does not offer adequate support for either of them, as we will show below.[1]

HPF is a procedural language and still an descendent of Fortran. From the software engineering point of view, Java is preferable to HPF [10], since it allows for clearly designed, re-usable, and maintainable code. But since HPF combines the collective knowledge on data parallel programming it must be considered when thinking about data parallelism in Java.

[1] Java stands for Java as defined in the JDK 1.1.5. We do not see any plans at Sun suggesting that future releases of the JDK will contradict the basic assumptions used here.

HPF offers forall statements, directives to distribute array data in a distributed memory environment, and statements for collective communication. Java does not offer language support for any of these. But there are at least three different approaches to add HPF expressiveness to Java.

A) Dual-Language Approach. To use both Java and HPF at the same time, libraries and wrapper functions are added to Java. The libraries communicate array data between Java and HPF. The wrappers allow to invoke HPF subroutines.

This approach aims at a gradual move from HPF programs to Java programs that use HPF functionality; it helps to guard investments made into HPF programs but does not prevent well designed object-oriented programs from taking over in future.

Mixing Java and HPF causes several problems. Since different HPF compilers behave differently, the portability gained by using Java is lost in the HPF subroutines. At the Java-to-HPF interface, problems are caused by both the different sets of primary types in each language, in particular by Java's lack of complex numbers, and by Java's inability to access array substructures, like dimensions or their subsections, efficiently. Finally, since Java's floating point operations do not necessarily adhere to IEEE floating point standards, different sections of a dual-language program will compute incompatible results.

B) Language and ByteCode extensions. The syntactic elements of HPF are added to the Java language; compilers that implement Java plus HPF semantics are constructed.

All HPF features can be covered with this approach. In a compiler for the new language, costly array bound checks can be avoided, the format used for floating point I/O can be chosen in a way that suits state-of-the-art parallel computers best, other Java features can be omitted to avoid difficult interactions with HPF semantics, e.g., one could re-evaluate Java's design decisions to use a ByteCode as an intermediate representation, to implement arrays and their sub-dimensions as objects, and to hide the memory

layout of array data from the user.

The disadvantage is that the special purpose compiler cannot be as portable as the JDK. For instance, since array accesses are explicit in Java ByteCode, the decision to avoid them requires a special form of ByteCode. If access to sub-sections of arrays shall be efficient, the memory layout must be defined, the gargabe collector must understand these sub-arrays, etc.

Therefore, one of the key advantages of Java, namely portability, is lost in that approach.

C) Pre-Compiler and Library. A third approach is to extend the expressive power by means of libraries for data distribution, collective communication, and forall statements. This approach has been taken for various parallel extensions of C++, e.g. [3, 6]. However, since Java does neither support templates nor macros, some restrictions apply.

Both data distribution and collective communication are best handled by libraries that handle and partition arrays on a distributed parallel machine. Since Java does not support parameterized classes, i.e., replicated versions of these libraries must exist for all primary types and for objects. For objects, type casts must be used to down-cast elements to the proper type by the programmer. One of Java's strengths, the strong static type checking, is lost.

The forall statement is *not* a candidate for macro or template processing (see section 2 for details). Therefore, the forall semantics must either be offered by means of another library or it can be implemented by a pre-compiler's source-to-source transformation that accepts forall statements in an extended Java and generates multi-threaded Java equivalents for them.

JavaParty, consisting of a pre-compiler and a distributed runtime system, offers data parallel programming by taking the third approach.

In the remainder of this paper, we discuss the design ideas of JavaParty with respect to data parallel programming. In section 2 it is argued that in general forall statements cannot be handled by macros or templates. Then the key issues of JavaParty's source-to-source transformation are presented in sections 3 to 5. Section 6 presents the runtime system that allows for transparent distribution of array components and for execution of remote threads. Benchmark results are presented in section 7 and compared to HPF and Fortran90 implementations.

2 Forall macros do *not* work

Java's threads express control parallelism. Threads are objects inheriting from a standard Thread class. This class offers a `run` method that returns immediately to the caller but is in fact executed concurrently. There is no other concept of concurrency available in Java, in particular, data parallelism cannot be expressed directly. Control parallelism is for concurrent applications that have a small number of concurrent activities, for example, graphical user interfaces and applets, and that run on a machine with one or only a few processors.

A too Simplistic Macro

The Java documentation supports the belief that threads can easily be used for data parallelism as well. Consider the following forall statement:

```
forall (int i = 0; i < 100; i++)
  bar(i);
```

With inner classes,[2] the JDK documentation suggests a multi-threaded equivalent as follows [17]:

```
 1  Thread[] worker = new Thread[100];
 2  for (int i = 0; i < 100; i++) {
 3    final int ii = i;
 4    worker[i] = new Thread() {
 5      public void run() {
 6        bar(ii);
 7      }
 8    };
 9    worker[i].start();
10  }
11  try {
12    for (int i = 0; i < 100; i++)
13      worker[i].join();
14  } catch (InterruptedException e) {
15    //some complicated cleanup code
16  }
```

Although the above code fragment is hard to read at first, one can get used to that pattern. In the body of the first `for` loop, an anonymous inner class is declared that inherits from Thread (lines 4–8). The `run` method of class Thread is overwritten by a specific `run` method that calls `bar(ii)` (lines 5–7). Since Java does not allow regular local variables to be used in inner classes one has to declare, initialize, and use a final helper variable instead (`ii`, lines 3 & 6). To execute `bar(ii)` concurrently, an object of that anonymous class is created and stored in an array of workers before its `start` method is called (lines 4 & 9). `Start` calls `run` which concurrently executes `bar(ii)`.

The main program can only continue after all concurrent invocations of `bar(i)` have been completed. This is implemented in lines 11–16. For each worker in the array, `join` is called (line 13). Since one of the threads might have been interrupted, the whole synchronization needs to be in a `try–catch` construct.

[2] Inner classes have been introduced into Java with version 1.1 of the JDK.

Although the above code appears to be a macro, transforming foralls statements into mutli-threaded Java, we discuss its incompleteness, superficiality, and inefficiency with respect to the demands of data parallel programs. Forall transformations require semantic knowledge and are much more complex.

Lack of Semantic Knowledge

Independent of the approach –there are alternatives to the inner class approach– the body of the forall is moved during the transformation and the surrounding state as defined by the contents of local variables must be captured at runtime and conveyed to the body's new position. Since Java's semantics distinguish between primary and reference types and between local and instance variables, capturing is type dependent. Hence, forall transformation schemes depend on the types of surrounding variables.

The above simplistic macro is incomplete if local variables are involved. Different transformations are needed depending on whether local variables or instance variables are used in the original forall. Assume a forall statement that uses local variables for the "iterations" to store results.[3] The transformation of such a forall requires that additional helper instance variables need to be declared at class level and are used instead of the local variables in the transformed body. After the end of the forall, the contents of the helpers must be assigned back to the local variables.

The transformation is thus non-local. It needs semantic knowledge about the types of variables involved. Since macros and templates operate locally and without semantic knowledge, the transformation cannot be achieved by them.

Although the macro approach cannot work, it is useful for understanding the simplistic macros's other disadvantages discussed below.

Lack of Generality

The simplistic macro suffers from limited applicability and generality.

Instead of a simple function call, there can be statements that alter the flow of control: the enclosing method can be left by a `return` statement, an exception can be thrown, a surrounding loop can be the target of a `break` or `continue` statement, etc. The semantics of these statements inside of a forall are similar to their semantic when used inside of a regular `for` statement.

[3]For example, imagine a parallel search for a target value in a two dimensional array. Each "iteration" of the forall searches for a different row of the array. If it is known that the target value either appears once or not at all, it is sufficient if the finder, if any, write the target's coordinates into two index variables. No additional synchronization is needed.

All these constructs cannot easily be used inside of the inner class. A generally applicable transformation scheme needs to be much more complicated.

Lack of Efficiency

• **Object creation cost.** Object creation in Java is expensive since each object needs some memory that must be initialized and registered for garbage collection, and an additional lock object must either be created by the Java virtual machine or is provided by the operating system by a costly kernel entry. Hence, for performance reasons numerous or frequent creation of thread objects must be avoided.

Unfortunately, the simplistic macro creates threads numerously and frequently. A thread object is started for every single "iteration" of the original forall. Moreover, for each forall, a new set of worker threads is created, started, and joined. After joining, the threads are useless and disposable by the garbage collector. They are useless because subsequent foralls have different bodies and require different `run` methods and because Java threads cannot be restarted.

For recursion and nested parallelism the above transformation scheme is not just costly but useless for it easily results in a number of thread objects that overwhelms the capabilities of either the Java virtual machine or the virtual memory system.

To avoid that sort of inefficiency and to make a transformation work for nested parallelism as well, threads must be re-used and standard virtualization loop techniques [16] must used. Both will increase the complexity of the transformation, since re-use of threads is difficult to achieve in Java, and since code for index set splitting and additional boundary checks for the first/last thread need to be added.

• **Fan-out and fan-in restrictions.** In general, data parallel programs have too many and too small foralls to accept the bottleneck caused by sequential start and join of threads.

When threads are recycled instead of being created over and over, `start` and `join` can no longer be used. But without them, tree structured re-start and barriers are even more difficult to implement. The reason is that Java's weak memory consistency model –there is no sequential consistency [11]– is coupled to unsuitable or too costly synchronization operations.[4]

Monitors and critical sections are too heavy weight to implement re-starts and barriers since they often cause slow kernel entries in current JVMs. Other mechanisms are awkward to use: `wait` and `notify`

[4]Simplified, is is not guaranteed that a thread sees the effects of write operations performed by different threads, unless synchronization operations have been executed before.

suffer from race conditions since `notify` operations are not buffered but are lost if no thread is waiting at an object; they are difficult to use for (distributed) load-balancing, since the JVM or the operating system may decide at will which out of several waiting threads is continued upon notification.

An efficient transformation template thus needs a tree structure for start-up and barriers. It must be capable of dealing with the specific characteristics of Java's synchronization primitives. Both requirements add further complexity.

Lack of Barrier Support

Pure SIMD semantics demand that all "iterations" proceed in perfect unison, i.e., no "iteration" starts to process an expression before the previous expression is completed by all "iterations".

Because of polymorphism, dynamic dispatch, and aliasing, there is no way a macro can transform all methods that are called in the body of a forall into a form that is amenable for completely synchronous execution. Moreover, it is not even promising to use compile time data dependence analysis techniques, e.g. [13, 18], for that task, since the same reasons render those techniques either too costly or too weak.

The only way out is the programmer must detect potential data dependencies and split the forall into parts so that the sources and targets of all dependencies are located in different forall statements.

```
forall (int i = 0; i < 100; i++) {
  bar(i);
  sync;
  gee(i);
}
```

Above, the necessity of a synchronization barrier is indicated by `sync`. Below are the splitted foralls:

```
forall (int i = 0; i < 100; i++)
  bar(i);
forall (int i = 0; i < 100; i++)
  gee(i);
```

Since, in general, a macro cannot determine the proper placements of barriers, and since a (pre-)compiler can only determine them in very special cases, this task is left to the programmer. However, macros and compilers can offer support in splitting foralls; compilers can even apply optimization techniques to minimize the number of barriers to improve performance.

3 Pre-compiling foralls does work

A pre-compiler can avoid the disadvantages of the macro approach. It has the necessary semantic knowledge and can modify given classes at a whole without distorting the clearness of the original code. A pre-compiler can use complex transformations, i.e., it can do its job completely and still achieve efficiency. In addition, it can help in splitting foralls.

This is the design idea of the forall treatment implemented in JavaParty: An additional data structure is created that can hold a pool of threads. This pool is filled with new thread objects, but only once per program. Instead of creating, starting, joining, and discarding new threads for every single data parallel section of the code, thread objects from that pool alternate between being blocked and executing: they are blocked and wait for work to arrive, execute the work, and return to a blocked state again. To allow for non-sequential start-up and join, the work packets are in a form so that the worker can decide whether to execute the packet (= virtualization loop) entirely or to split it up and work only on a part. The worker's strategy takes into account the current load, the number of available threads, and the size of the packet.

Based on this design idea, section 4 presents the details of JavaParty's source-to-source transformation and of the required libraries.

4 Forall Implementation in JavaParty

The JavaParty pre-compiler accepts Java plus forall statements and generates efficient multi-threaded pure Java code. Instead of a back-end that emits Java source code, a regular ByteCode back-end can be used to speed up compilation by avoiding additional file I/O and repeated semantic analysis.

For now, we deliberately restrict the presentation to a shared memory architecture, i.e., the presentation is based on threads and a common address space as provided by Java. Section 5 presents the optimized implementation of synchronization barriers. Section 6 focuses on object and thread distribution in a distributed memory parallel computer.

ForallThread and WorkPile

A central component in JavaParty's forall library is the `ForallThread`. Upon program start, a fixed number of these threads are started. Their `run` method is an endless loop that gets work from a work pile, executes that work, indicates completion and starts over.

JavaParty uses Pizza's closures [15] to capture state and to construct functions.[5] In the code fragment below, `wp.getWork` returns a function from the work pile

[5] Pizza uses a source-to-source transformation as well. Although we could have used the final Java code (no closures), we discuss the intermediate Pizza representation (with closures), because it is easier to understand.

that neither takes arguments nor returns a result. In Pizza terminology, this function is of type `(->void)`. More details on closure implementation and the closure type system of Pizza can be found in [12].

```
class ForallThread extends Thread {
  private WorkPile wp;
  ...
  public void run() {
    while(true) {
      (->void) fkt = wp.getWork();
      fkt();
      wp.doneWork();
    }
  }
}
```

The work pile is designed according to several design patterns [5]. There is only a single work pile in the system ("Singleton"), initialized upon program start. The design pattern "Strategy" is used for the `SplitStrategy` that is explained later on.

The method `doWork` puts new work onto work pile. If the pile is full and `AddWork` returns `false` the current thread executes the work rather than postponing it for another `ForallThread` to do the work. The `SplitStrategy` implements the semantics of `full`. For example, the work pile is considered to be full, if all `ForallThreads` are busy and a certain number of packets are on pile, etc.

```
public class WorkPile {
  private static WorkPile wp = new WorkPile();
  private static SplitStrategy splstrat =
    new DefaultSplitStrategy();
  ...
  public static void doWork(ForallController fac,
                            (->void) fkt) {
    Thread.currentThread().yield();
    if (!wp.AddWork(fac, fkt))
      fkt();
  }
```

In addition to `doWork`, there is the synchronized method `AddWork` that either returns `false` if the work pile is not full, or the work is added to the pile. An arbitrary `ForallThread` that is blocked in `getWork` is notified and resumes. It is irrelevant which `ForallThread` resumes, hence, Java's awkward notification mechanisms does not interfere.

```
synchronized boolean AddWork(ForallController fac,
                             (->void) fkt) {
  if (full()) return false;
  else {
    totalWorkInQ++;
    fac.addWork();
    enqueue(new Work(fkt));
    notify();
    return true;
  }
}
```

A `ForallThread` waits inside of the `getWork` method if no work is on the pile. Otherwise the work is dequeued and returned to the thread. Since a `while` loop is used to implement the blocking a lost notification does not hurt. Since the method is synchronized, no other `ForallThreads` can interfere.

```
synchronized (->void) getWork() {
  while (isEmpty()) {
    try {
        wait();
    } catch (Exception e) { /* ignore */ }
  }
  totalWorkInQ--;
  totalBusyThreads++;
  return dequeue().fkt;
}
```

Forall Transformation Pattern

We now describe the code that results from the simple forall statement from section 2.

```
forall (int i = 0; i < 100; i++)
  bar(i);
```

The part of the resulting code that textually replaces the given forall is discussed first. The body of the original forall is moved to a closure declaration (discussed below) that preceeds the following code fragment.

```
try {
  ForallController fac = new ForallController();
  WorkPile.doWork(fac, makeCl(0, 100, 1, fac));
  fac.finalBarrier();
} catch (RuntimeException _exc) {
  throw _exc;
} catch (Error _exc) {
  throw _exc;
}
```

A `try-catch` block replaces the original forall. Inside the `try` block, a `ForallController` is created that is unique to an execution of the whole forall. The work pile is asked to `doWork`. It is handed the `ForallController` and the result of `makeCl`. This function returns a closure describing the forall range and containing the original body. Before we discuss the details of the closures, let us look into the rest of the `try`. The `finalBarrier` makes sure that all work packets that belong to the forall have been completed before execution continues. Since an unknown number of `ForallThreads` will each work on an unknown number of work packets, a sequential join phase no longer works. Instead, the `ForallController` keeps track of the number of packets to be completed. If that number reaches zero, execution can continue past the barrier. The `catch` clauses are necessary to pass exception objects from the body of the forall to the caller: If one of the "iterations" of the original forall

throws an exception, this exception is now thrown inside of a `ForallThread`, i.e., at a different position in the code. To simulate the intended behavior, the exception object is caught in the `ForallThread`, buffered in the `ForallController` and then re-thrown inside of `finalBarrier` The JavaParty compiler generates `catch` clauses for RuntimeExceptions and Errors since those need not be declared by the programmer. In addition (not shown in the example) a `catch` clause is generated for every named exception that can be thrown in the original body.

The function `makeCl` is a closure that has four parameters and returns a void function without parameters. That function is called `forallCl`; it is defined in lines 4–21 below. A `ForallThread` will later execute `forallCl`. The forall closure first checks that no other thread has encountered a `break` (line 5). The call of the split strategy that checks whether the work packet must be split into parts before being executed (lines 6–7) implements a high fan-out start. If no split is necessary, the work is performed in the virtualization loop (lines 8–17).[6] In each iteration, the loop checks the break flag again and executes `bar(i)`. Any potential exceptions is caught and registered with the `ForallController`. To work on Java virtual machines without pre-emptive scheduling as well,[7] the current thread is asked to yield. The thread indicates completion at the `ForallController` so that the `finalBarrier` can work as expected.

```
 1 void() (int, int, int, ForallController)
 2 makeCl = fun(int lb, int ub, int step,
 3              ForallController fac) {
 4   void() forallCl =  fun() {
 5     if (! fac.breakFlag) {
 6       if (! WorkPile.split(fac, makeCl,
 7                            lb, ub, step)){
 8         for (int i = lb; i <= ub; i += step) {
 9           if (fac.breakFlag) break;
10           try {
11             bar(i);
12           } catch (Throwable e) {
13             fac.caughtException = e;
14             fac.breakFlag = true;
15           };
16           Thread.currentThread().yield();
17         };
18       };
19     };
20     fac.workDone();
21   };
22   return forallCl;
23 };
```

When splitting work packets, `forallCl` calls `makeCl`. This is why two nested closures are required.

[6] The upper bound check is simplified for readability; the correct test depends on the sign of `step`.

[7] Several releases of the JDK implemented so called green threads without pre-emptive scheduling.

ForallController

The `ForallController` now works as expected. It has a `counter` to keep track of the number of work packets that are being processed or in the pile. If that counter reaches zero, all waiting threads are notified to continue. An exception that might have been registered with the `ForallController` during the execution of the body is thrown to the caller at the original position in the code.

```
public class ForallController {
  public boolean breakFlag;
  public Throwable caughtException;
  int counter;
  ...
  public synchronized void workDone() {
    if ((--counter)<=0)
      notifyAll();
  }
  public synchronized void finalBarrier()
  throws Throwable {
    try {
      while (counter > 0)
        wait();
    } catch (Exception e) { /* ignore */ }
    if (caughtException != null)
      throw caughtException;
  }
}
```

Split Strategy

The strategy can access the ForallController, can call the closure `makeCl` that makes a forall closure, and knows the range of the virtualization loop in a given work packet. Moreover the strategy knows about the number of active and blocked threads and the number of packets in the work pile.

```
  public static boolean
  split(ForallController fac,
        (int, int, int,
         ForallController -> (->void)) makeCl,
        int lwb, int upb, int stp) {
    ...
  }
```

Several strategies can be implemented and selected dynamically. The JavaParty environment provides some default strategies that work well with nested forall statements and do not cause any deadlocks. These strategies can be refined by the user.

5 Optimal Barriers in JavaParty

In the implementation of foralls presented so far, the programmer is in charge of splitting forall statements if data dependencies demand synchronization. However, as has been mentioned above, the compiler can assist the programmer with the splitting, especially if control flow statements are to be split. This section

gives a motivating example first, before presenting the central idea of the restructuring that results in a minimal number of barriers.

Example of Barrier Elimination

Consider the following situation; the two necessary synchronizations are indicated by `sync`. The first `sync` refers to those conceptual threads that entered the `if` branch; it does not affect the other threads. (And vice versa for the second `sync`.)

```
forall (int i = 1; i < 100; i++) {
  if (cond(i)) {
    bar(i);
    sync;
    gee(i);
  } else {
    bar(i/2);
    sync;
    gee(i/2);
  }
}
```

One synchronization barrier is sufficient for both the synchronization of the `if` branch and the synchronization of the `else` branch. An indication of which branch has been selected is stored temporarily in helper arrays `tmp_if` and `tmp_else` of booleans.[8]

```
boolean tmp_if[]   = new boolean[100]; //all false
boolean tmp_else[] = new boolean[100]; //all false
forall (int i = 1; i < 100; i++) {
  if (cond(i)) {
    bar(i);
    tmp_if[i] = true;
  } else {
    bar(i/2)
    tmp_else[i] = true;
  }
}
forall (int i = 1; i < 100; i++) {
  if (tmp_if[i])   gee(i)
  if (tmp_else[i]) gee(i/2)
}
```

The straightforward approach is to use a single helper array to store the evaluation of `cond(i)`, and to use that helper array in the second forall to decide for every thread which branch to enter. But that does not work in the general case of arbitraty control flow statements, especially if for an individual thread k the flow of control leaves early in the first forall, e.g., by a `break` statement. In such cases, the use of a single helper would result in a unintended execution of a "iteration" k in the second forall. Thus, the general solution is to use a branch specific helper variable to register the fact that a thread k is supposed to continue in a later forall immediately before k leaves that branch in the first forall.

[8] The actual implementation in JavaParty uses bit vectors for performance reasons.

Forall Restructuring Pattern

JavaParty provides automatic and efficient forall splitting and thus simplifies the programmer's task.

The central idea of the restructuring algorithm is to attribute individual statements in the body of a forall with *synchronization ranks* and to use a topological sort based on these ranks. Two statements in the body of a forall that are separated by `sync` have synchronization ranks that are at least one apart. In the example, the condition and both occurrences of `bar` have synchronization rank 1, the two occurrences of `gee` must be separated from the preceding invocations of `bar` by a barrier, hence they have synchronization rank 2. Statements that alter the flow of control, like the `if` statement, are attributed with an interval of synchronization ranks defined by the smallest and the largest synchronization rank of any statement in their bodies. The `if` statement of the example has the synchronization interval [1:2] since in both branches there are statements with ranks 1 and 2.

The synchronization ranks define an order that is used during the restructuring: for each rank, a separate forall statement is constructed. In the body of the forall of rank x, all those statements are placed that either have synchronization rank x, or have an interval of synchronization ranks including x. Control flow statements can thus re-appear a few times, albeit with changing conditions; in the example, there are `if` statements in both foralls.

Before the first forall is generated that belongs to the synchronization interval of a control flow statement, as many helpers are created as there are branches in the control flow statement. Since the `if` of the example has two branches, two helpers are created. For loops, a single helper array is sufficient. The default initialization (`false`) is sufficient, except for `while` loops, since they use an entry-condition. For all loop constructs, there is the additional problem that the transformation has to pull the loop out of the forall statement.

```
forall (int i = 1; i < 100; i++) {
  while (cond(i)) {
    bar(i);
    sync;
    gee(i);
  }
}
```

This forall loop runs until the `while` loop has been terminated for each value of `i`. The transformation result is shown below. In a first forall statement, the helper array is initialized according to `cond(i)`. The `while` loop runs until no element of the helper array is true. In the body of the `while` loop there are two

foralls, one for each synchronization rank. At the end of the last forall, the helper variable might be set to `false` if for a certain index the original `while` loop would terminate.

```
forall (int i = 1; i < 100; i++)
  tmp_while[i] = cond(i);
while (atLeastOneTrue(tmp_while)) {
  forall (int i = 1; i < 100; i++) {
    if (tmp_while[i])
      bar(i);
  }
  forall (int i = 1; i < 100; i++) {
    if (tmp_while[i]) {
      gee(i);
      if (!cond(i)) tmp_while[i] = false;
    }
  }
}
```

The JavaParty implementation includes transformation rules for all control flow constructs. In particular, JavaParty can transform nested forall statements with arbitrary many `sync` points into a sequence of nested forall statements without any internal `sync`. The details of the other transformation rules are omitted here for reasons of brevity.

6 Object Distribution in JavaParty

At this point it has been sketched how JavaParty restructures forall statements with `sync` into ones without, and we know how JavaParty transforms the latter into efficient multi-threaded equivalents. The threads still live in a single address space of a Java virtual machine that is at best executed on a multi-processor SMP. This section sketches ways to distribute objects in a network, and since threads are objects, thread distribution is covered as well.

Java provides socket communication and RMI, the remote method invocation interface, for programming of distributed memory environments. Unfortunately, both are not appropriate. We have shown in [14] that the number of lines that have to be added to or changed in a given multi-threaded program to make it work in a distributed environment is of the same order of magnitude as the original program size. This doubling of code size happens both for socket based communication and for RMI based approaches. Moreover, Brose et al. argue in [1] that RMI's object model that distinguishes between remote and local objects causes severe misunderstandings and bugs when programming with that model.

We therefore have designed an additional transformation phase, that turns Java objects into remote Java objects that reside in a distributed setting. These remote objects are much closer to Java's object semantics than RMI objects. Remote objects can migrate across the machine, especially, they can be made local to speed up access. There is no semantic difference between remote objects that are accessed from a remote host and remote objects that are accessed locally. Migration and object placement can be done by the runtime system (runtime and compiler techniques to enhance locality are available) or objects can be placed manually.

We do not recapitulate the transformation techniques here, since the details can be found in [14].

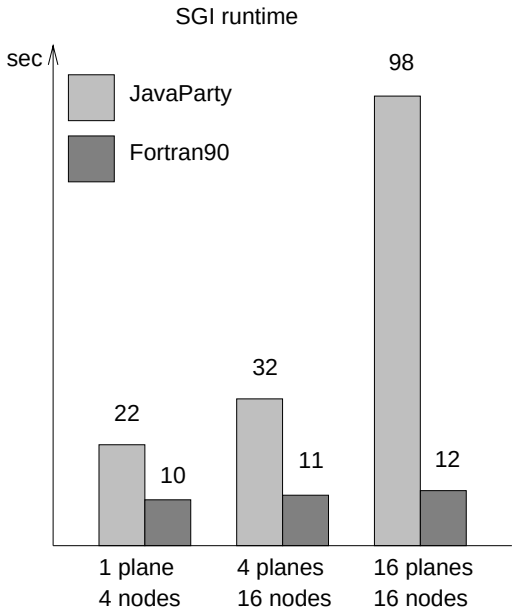
7 Results

We have studied large-scale geophysical algorithms, called Veltran velocity analysis and Kirchhoff migration to evaluate the efficiency of JavaParty's forall. These are basic algorithms used in geophysics for the analysis of the earth's sublayers by means of sound wave reflection. Since it can take terra bytes of input data to cover a reasonable area, the performance of these algorithms is crucial. The geophysics and the details of the benchmarks can be found in [9].

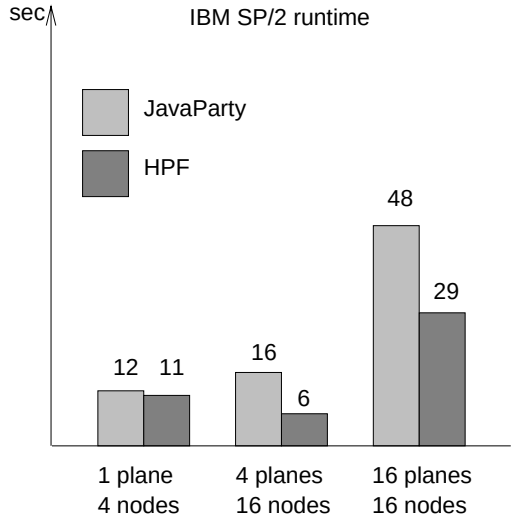
In cooperation with the Stanford Exploration Project [4], we have implemented these algorithms in JavaParty, in HPF, and in Fortran90. We then benchmarked the programs on up to 16 nodes of an IBM SP/2 distributed memory parallel computer as well as on 16 nodes of an SGI PowerChallenge shared memory machine. The JavaParty implementation on both machines is based on the JDK 1.1.2 with just-in-time compilers. On each node, a separate Java virtual machine is started.[9] On the SGI, we used SGI's standard Fortran90 compiler; on the IBM SP/2, version 2.2 of the Portland Group High Performance Fortran compiler is used.

The graph below shows the results for the SGI PowerChallenge. The JavaParty implementation faces a slowdown compared to the Fortran90 implementation of 2.2 to 8.2. The upper factor is due to the fact that the SGI currently does not support true parallelism of Java threads. We had to start a separate Java virtual machine per node which communicated through RMI instead of using the common address space. We expect the factor 8.2 to shrink significantly when truly parallel Java threads become available with a future release of SGI's Java virtual machine.

[9] Although the IBM SP/2 offers an alpha version of a High Performance Java Compiler [8] compiling to native and statically linked code, it is too early to seriously use that compiler. The HPJ compiler revealed a general speed up of 1.6 compared to just-in-time Java performance, but unfortunately HPJ's communication through RMI turned out to be slower by a factor of between 20 and 35.



On the IBM SP/2, the results are even better, as shown in the graph below. The JavaParty version is slower than the HPF version by a factor of between 1.1 and 2.7 depending on processor utilization.



On both platforms, Java's performance is not attained by compiled and optimized native code but instead relies on interpreters with just-in-time compilation features. In addition to the expected availability of native threads on the SGI, we expect significant performance increases on fine-grained computations in the near future because of two main reasons.

First, we had to use JDK 1.1.2, because later releases are not yet available for our hardware platforms. Later versions have increased performance, especially RMI performance, on Solaris and Wintel platforms and are likely to show the same effect in our environments. Future releases (JDK 1.2) are announced to speed up the JVM even further.

Second, compilers producing optimized native code like IBM's High Performance Java Compiler [8] are on the horizon. These compilers will approach Fortran performance because they can apply much more sophisticated optimization techniques than current just-in-time compilers.

8 Related Work

An advantages of JavaParty in comparison to other projects aimed at adding data parallelism to Java is portability. JavaParty programs run on workstations, on shared memory parallel computers and in truly distributed environments. Moreover, JavaParty's forall statement allows for a single program approach that is independent of the underlying topology. And finally, JavaParty programs come close the native HPF and Fortran90 performance.

Hummel et al. [7] work on SPMD programming in Java and faced some of the same problems we have discussed in section 2. The main contribution are classes for efficient synchronization and for regular and irregular collective communications. Such classes are useful for JavaParty. The system runs on an IBM SP/2, but in contrast to JavaParty's 100% pure Java approach, it uses a runtime system written in C that interacts with a native MPI communication library.

In the HPJava [2] project at Syracuse, Carpenter, Fox et al. use wrappers to interface to native HPF code and library-based extensions of Java. They offer useful classes for distribution of array data and for collective communication as well. Whereas JavaParty has the capability of transforming a single program with forall statements into distributed programs that communicate by means of message passing, HPJava requires the implementation and invocation of several node programs. Such node programs know about the processor topology, the arrays to be used and their distribution across the machines. Virtualization loops get their boundaries from library calls. Since the programmer is in charge of distributing the threads (one per processor) several of the disadvantages of a macro transformation are avoided. However, HPJava's approach cannot handle nested parallelism nor does it offer any support for the efficient implementation of barriers. HPJava as well uses native routines and MPI calls to implement the communication.

9 Conclusion

This paper has discussed alternatives to add data

parallelism to Java, has reasoned that forall statements in general cannot be implemented by means of standard macro processing, has presented JavaParty's source-to-source transformation that generates efficient multi-threaded equivalents from forall statements, has sketched an optimization technique to avoid unnecessary synchronization barriers, and has demonstrated that JavaParty's data parallelism can achieve performance that comes close to HPF and Fortran90 code and is expected to improve.

The JavaParty system is freely available for non-commercial projects. For details and downloading see `http://wwwipd.ira.uka.de/JavaParty`.

Acknowledgements

I would like to thank the JavaParty group, especially Matthias Zenger, for their support of the JavaParty environment. Matthias Jacob implemented the geophysical algorithms. Christian Nester pointed out several synchronization bugs in the first version [19] of the forall transformation. Furthermore, we want to express gratitude to Maui High Performance Computing Center as well as Karlsruhe Computing Center for the granting access on the IBM SP/2.

References

[1] Gerald Brose, Klaus-Peter Löhr, and André Spiegel. Java does not distribute. Technical Report B 97-07, Freie Universität Berlin, 1997.

[2] Bryan Carpenter, Yuh-Jye Chang, Geoffrey Fox, Donald Leskiw, and Xiaoming Li. Experiments with "HPJava". *Concurrency: Practice and Experience*, June 1997.

[3] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object-oriented programming notation. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 281–313. MIT Press Cambridge, Massachusetts, London, England, 1993.

[4] J. Clearbout and B. Biondi. Geophysics in object-oriented numerics (GOON): Informal conference. In *Stanford Exploration Project Report No. 93*. October 1996. http://sepwww.stanford.edu/sep.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1994.

[6] Andrew S. Grimshaw. Easy to use object-oriented parallel programming. *IEEE Computer*, 26(5):39–51, May 1993.

[7] Susan Flynn Hummel, Ton Ngo, and Harini Srinivasan. SPMD programming in Java. *Concurrency: Practice and Experience*, June 1997.

[8] IBM. High performance compiler for Java. http://www.alphaWorks.ibm.com.

[9] Matthias Jacob. Implementing Large-Scale Geophysical Algorithms with Java: A Feasibility Study. Master's thesis, Karlsruhe Univeristy, November 1997.

[10] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press Cambridge, Massachusetts, London, England, 1994.

[11] Doug Lea. *Concurrent Programming in Java – Design Principles and Patterns*. Addison-Wesley, Reading, Mass., 1996.

[12] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[13] Michael Philippsen and Ernst A. Heinz. Automatic synchronization elimination in synchronous foralls. In *Frontiers '95:The 5th Symp. on the Frontiers of Massively Parallel Computation*, pages 350–357, Mc Lean, VA, February 6–9, 1995.

[14] Michael Philippsen and Matthias Zenger. JavaParty: Transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[15] Pizza. http://www.cis.unisa.edu.au/~pizza.

[16] Michael J. Quinn and Philip J. Hatcher. Data-parallel programming on multicomputers. *IEEE Software*, 7(5):69–76, September 1990.

[17] JavaSoft (John Rose). Inner class specification: Further example: Multi-threaded task partitioning. http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/spec/innerclasses.doc13.html.

[18] Chau-Wen Tseng. Compiler optimizations for eliminating barrier synchronization. In *5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPoPP*, pages 144–155, Santa Barbara, CA, July 19–21 1995.

[19] Matthias Winkel. Erweiterung von Java um ein FORALL. Studienarbeit, University of Karlsruhe, Department of Informatics, May 1997.