Collective Communications for Scalable Programming

Sang Boem Lim[1], Bryan Carpenter[2], Geoffrey Fox[3]
and Han-Ku Lee[4⋆]

[1] Korea Institute of Science and Technology Information (KISTI)
Daejeon, Korea
slim@kisti.re.kr
[2] OMII, University of Southampton
Southampton SO17 1BJ, UK
dbc@ecs.soton.ac.uk
[3] Pervasive Technology Labs at Indiana University
Bloomington, IN 47404-3730
gcf@indiana.edu
[4] School of Internet and Multimedia Engineering, Konkuk University
Seoul, Korea
hlee@konkuk.ac.kr

Abstract. HPJava is an environment for scientific and parallel programming using Java. It is based on an extended version of the Java language. One feature that HPJava adds to Java is a multi-dimensional array, or multiarray, with properties similar to the arrays of Fortran. We are using Adlib as our high-level collective communication library. Adlib was originally developed using C++ by the Parallel Compiler Runtime Consortium (PCRC). Many functionalities of this high-level communication library is following its predecessor. However, many design issues are reconsidered and re-implemented according to Java environment. Detailed functionalities and implementation issues of this collective library will be described.

1 Introduction

The basic features of HPJava [10] [11] [12] have been described in several earlier publications. In this paper we will jump straight into a discussion of the implementation of some collective communications in HPJava.

The main characteristic change from Java to HPJava is to add a concept of multi-dimensional arrays, called "*multiarrays*". And to support parallel programming, HPJava creates "multiarrays" by extending multiarrays. These "multiarrays" are very closely modeled on the arrays of High Performance Fortran (HPF). The new distributed data structures are cleanly integrated into the syntax of the language. In other word, new distributed data structure doesn't interfere with

⋆ Correspondence to: Han-ku Lee, School of Internet and Multimedia Engineering, Konkuk University, Seoul, Korea

the existing syntax and semantics of Java-for example ordinary Java arrays are left unaffected.

New syntaxes in the source HPJava program is translated to an intermediate standard Java file and this Java file is compiled using ordinary Java compiler. The preprocessor that performs this task is reasonably sophisticated. During the preprocessor phase, it performs a complete static semantic check of the source program, following rules that include all the static rules of the Java Language Specification [9]. So it should not normally happen that a program accepted by the HPJava preprocessor would be rejected by the backend Java compiler. The translation scheme depends on type information, so we were essentially forced to do a complete type analysis for HPJava (which is a superset of standard Java). Moreover we wanted to produce a practical tool, and we felt users would not accept a simpler preprocessor that did not do full checking.

The current version of the preprocessor also works hard to preserve line-numbering in the conversion from HPJava to Java. This means that the line numbers in run-time exception messages accurately refer back to the HPJava source. Clearly this is very important for easy debugging.

A translated and compiled HPJava program is a standard Java class file, ready for execution on a distributed collection of JIT-enabled Java Virtual Machines. All externally visible attributes of an HPJava class can be transparently reconstructed from Java signatures stored in the class file. This makes it possible to build libraries operating on distributed arrays, while maintaining the usual portability and compatibility features of Java. The libraries themselves can be implemented in HPJava, or in standard Java, or as JNI interfaces to other languages. The HPJava language specification documents the mapping between distributed arrays and the standard-Java components they translate to.

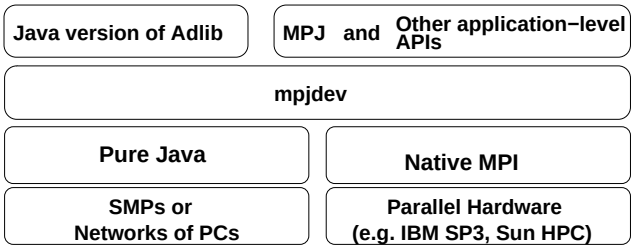

Fig. 1. An HPJava communication stack

Currently HPJava is supplied with one library for parallel computing-a Java version of the *Adlib* library of collective operations on distributed arrays [14]. A version of the *mpiJava* [1] binding of MPI can also be called directly from HPJava programs. Figure 1 summarizes an HPJava communication libraries stack. This figure shows how high-level collective libraries and low-level device library are working together.

2 Related Works

UC Berkeley is developing Titanium [3] to add a comprehensive set of parallel extensions to the Java language. Support for a shared address space and compile-time analysis of patterns of synchronization is supported.

The Timber [2] project is developed from Delft University of Technology. It extends Java with the Spar primitives for scientific programming, which include multidimensional arrays and tuples. It also adds task parallel constructs like a foreach construct.

Jade [8] from University of Illinois at Urbana-Champaign focuses on message-driven parallelism extracted from interactions between a special kind of distributed object called a Chare. It introduces a kind of parallel array called a ChareArray. Jade also supports code migration.

HPJava differs from these projects in emphasizing a lower-level (MPI-like) approach to parallelism and communication, and by importing HPF-like distribution formats for arrays. Another significant difference between HPJava and the other systems mentioned above is that HPJava translates to Java byte codes, relying on clusters of conventional JVMs for execution. The systems mentioned above typically translate to C or C++. While HPJava may pay some price in performance for this approach, it tends to be more fully compliant with the standard Java platform.

3 High-level Collective Communications

A C++ library Adlib [6] was completed in the Parallel Compiler Runtime Consortium (PCRC) [7] project. It was a high-level runtime library designed to support translation of data-parallel languages. It incorporated a built-in representation of a distributed array, and a library of communication and arithmetic operations acting on these arrays. The array model supported general HPF-like distribution formats, and arbitrary regular sections.

The Adlib series of libraries support *collective operations* on distributed arrays. All members of some active process group, which may or may not be the entire set of processes executing the program, must invoke a call to a collective operation simultaneously. Communication patterns supported include HPF/Fortran 90 intrinsic such as **cshift**. More importantly they include the regular-section copy operation, **remap**, which copies elements between shape-conforming array sections regardless of source and destination mapping. Another function, **writeHalo**, updates ghost areas of a distributed array. Various collective **gather** and **scatter** operations allow irregular patterns of access. The library also provides essentially all Fortran 90 arithmetic transformational functions on distributed arrays and various additional HPF library functions.

Figure 2 shows how collective communication is used in HPJava. It creates a general purpose matrix multiplication routine that works for arrays with any distributed format. This program takes arrays which may be distributed in both their dimensions, and copies into the temporary array with a special distribution

```
public class Comm {
  public void matmul(float [[-,-]] c, float [[-,-]] a, float [[-,-]] b) {

    Group2 p = c.grp();
    Range  x = c.rng(0);
    Range  y = c.rng(1);

    int N = a.rng(1).size();

    float [[-,*]] ta = new float [[x, N]] on p;
    float [[*,-]] tb = new float [[N, y]] on p;

    Adlib.remap(ta, a);
    Adlib.remap(tb, b);

    on(p)
      overall(i = x for : )
        overall(j = y for : ) {

          float sum = 0;
          for(int k = 0; k < N ; k++)
            sum += ta [i, k] * tb [k, j];

          c[i, j] = sum;
        }
  }
```

Fig. 2. A general Matrix multiplication in HPJava.

format for better performance. A collective communication schedule **remap()** is used to copy the elements of one distributed array to another. From the viewpoint of this paper, the most important part of this code is communication method. One of the most characteristic and important communication library methods, **remap()**, takes two arrays as arguments and copies the elements of the source array to the destination array, regardless of the distribution format of the two arrays.

3.1 Implementation of Collectives

By using a characteristic example of collective communication, we will discuss implementation of the Java Adlib collectives. For illustration we concentrate on the important **remap** operation. Although it is a powerful and general operation, it is actually one of the more simple collectives to implement in the HPJava framework.

General algorithms for this primitive have been described by other authors. For example it is essentially equivalent to the operation called **Regular_Section_Copy_Sched**

```
public abstract class BlockMessSchedule {

  BlockMessSchedule(int rank, int elementLen,boolean isObject) { ... }

  void sendReq(int offset, int[] strs, int[] exts, int dstId) { ... }
  void recvReq(int offset, int[] strs, int[] exts, int srcId) { ... }

  void build()   { ... }
  void gather()  { ... }
  void scatter() { ... }
  ...
}
```

Fig. 3. API of the class `BlockMessSchedule`

in [4]. In this section we want to illustrate how this kind of operation can be implemented in terms of the particular **Range** and **Group** hierarchies of HPJava (complemented by a suitable set of messaging primitives).

Constructor and public method of the remap schedule for distributed arrays of float element can be described as follows:

```
class RemapFloat extends Remap {
  public RemapFloat (float # dst, float # src) {...}

  public void execute() {...}
  . . .
}
```

The remap schedule combines two functionalities: it reorganizes data in the way indicated by the distribution formats of source and destination array. Also, if the destination array has a replicated distribution format, it broadcasts data to all copies of the destination. Here we will concentrate on the former aspect, which is handled by an object of class **RemapSkeleton** contained in every **Remap** object.

During construction of a **RemapSkeleton** schedule, all send messages, receive messages, and internal copy operations implied by execution of the schedule are enumerated and stored in light-weight data structures. These messages have to be sorted before sending, for possible message agglomeration, and to ensure a deadlock-free communication schedule. These algorithms, and maintenance of the associated data structures, are dealt with in a base class of **RemapSkeleton** called **BlockMessSchedule**. The API for the super class is outlined in Figure 3. To set-up such a low-level schedule, one makes a series of calls to **sendReq** and **recvReq** to define the required messages. Messages are characterized by an offset in some local array segment, and a set of strides and extents parameterizing a multi-dimensional patch of the flat Java array. Finally the **build()** operation does any necessary processing of the message lists. The schedule is executed in a "forward" or "backward" direction by invoking **gather()** or **scatter()**.

The implementation details of **BlockMessSchedule** will not be discussed in greater detail here because they are not particularly specific to our HPJava system, and the principles are fairly well known (see for example [4]).

However we do wish to describe in a little more detail the implementation of the higher-level **RemapSkeleton** schedule on top of **BlockMessSchedule**. This provides some insight into the structure HPJava distributed arrays, and the underlying role of the special **Range** and **Group** classes.

To produce an implementation of the **RemapSkeleton** class that works independently of the detailed distribution format of the arrays we rely on virtual functions of the **Range** class to enumerate the blocks of index values held by each process. These virtual functions, implemented differently for different distribution formats, encode all-important information about those formats. To a large extent the communication code itself is distribution format independent.

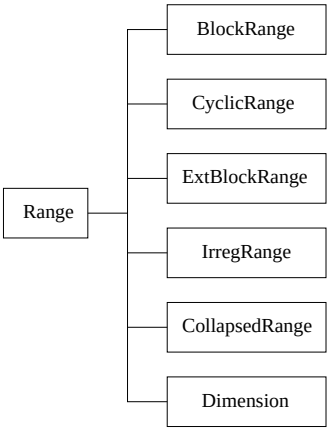

Fig. 4. The HPJava Range hierarchy

The range hierarchy of HPJava is illustrated in Figure 4 and some of the relevant virtual functions are displayed in the API of Figure 5. The most relevant methods optionally take arguments that allow one to specify a contiguous or striped subrange of interest. The **Triplet** and **Block** classes represent simple struck–like objects holding a few **int** fields describing respectively a "triplet" interval, and the strided interval of "global" and "local" subscripts that the distribution format maps to a particular process. In the examples here **Triplet** is used only to describe a range of *process coordinates* that a range or subrange is distributed over.

Now the **RemapSkeleton** communication schedule is built by two subroutines called **sendLoop** and **recvLoop** that enumerate messages to be sent and received respectively. Figure 6 sketches the implementation of **sendLoop**. This is a recursive function-it implements a multidimensional loop over the **rank** dimensions of the arrays. It is initially called with `r = 0`. There is little point going

```
public abstract class Range {
    public int size() {...}
    public int format() {...}
    ...
    public Block localBlock() {...}
    public Block localBlock(int lo, int hi) {...}
    public Block localBlock(int lo, int hi, int stp) {...}

    public Triplet crds() {...}
    public Block block(int crd) {...}

    public Triplet crds(int lo, int hi) {...}
    public Block block(int crd, int lo, int hi) {...}

    public Triplet crds(int lo, int hi, int stp) {...}
    public Block block(int crd, int lo, int hi, int stp) {...}
    . . .
}
```

Fig. 5. Partial API of the class `Range`

into full detail of the algorithm here, but an important thing to note is how this function uses the virtual methods on the range objects of the source and destination arrays to enumerate blocks-local and remote-of relevant subranges, and enumerates the messages that must be sent. Figure 7 illustrates the significance of some of the variables in the code. When the offset and all extents and strides of a particular message have been accumulated, the **sendReq()** method of the base class is invoked. The variables **src** and **dst** represent the distributed array arguments. The inquiries **rng()** and **grp()** extract the range and group objects of these arrays.

Of the collective communication schedules currently implemented in Adlib, all except **WriteHalo** share with **Remap** this property that their implementation code does not explicitly depend on the distribution format of the arrays. All rely heavily on the methods and inquiries of the **Range** and **Group** classes, which abstract the distribution format of arrays.

3.2 Other Schedules in Adlib

We described main characteristic example of the regular communications, **remap()**. This section we will overview functionalities of all collective operations in Adlib. The Adlib has three main families of collective operation: regular communications, reduction operations, and irregular communications. We discuss usage and high-level API overview of Adlib methods.

The method **shift()** is a communication schedule for shifting the elements of a distributed array along one of its dimensions, placing the result in another

```
private void sendLoop(int offset, Group remGrp, int r){

 if(r == rank) {
   sendReq(offset, steps, exts, world.leadId(remGrp));
 } else {
   Block loc = src.rng(r).localBlock();

   int offsetElem = offset + src.str(r) * loc.sub_bas;
   int step       = src.str(r) * loc.sub_stp;

   Range rng = dst.rng(r);
   Triplet crds = rng.crds(loc.glb_lo, loc.glb_hi, loc.glb_stp);

   for (int i = 0, crd = crds.lo; i < crds.count; i++, crd += crds.stp){
     Block rem = rng.block3(crd, loc.glb_lo, loc.glb_hi, loc.glb_stp);

     exts[r]  = rem.count;
     steps[r] = step * rem.glb_stp;

     sendLoop(offsetElem + step * rem.glb_lo,
              remGrp.restrict(rng.dim(), crd), r + 1) ;
   }
 }
}
```

Fig. 6. `sendLoop` method for `Remap`

array. In general we have the signature:

```
void shift(T # destination, T # source,
           int shiftAmount, int dimension)
```

where the variable T runs over all primitive types and Object, and the notation T `#` means a multiarray of arbitrary rank, with elements of type T . The `shiftAmount` argument, which may be negative, specifies the amount and direction of the shift. In the second form the `dimension` argument is in the range $0, \ldots, R-1$ where R is the rank of the arrays: it selects the array dimension in which the shift occurs. The source and destination arrays must have the same shape, and they must also be *identically aligned*.

The function `broadcast()`, which is actually a simplified form of `remap()`. There are two signatures:

```
T broadcast(T [[]] source)
```

and

```
T broadcast(T source, Group root)
```

The first form takes rank-0 distributed array as argument and broadcasts the element value to all processes of the active process group. Typically it is used

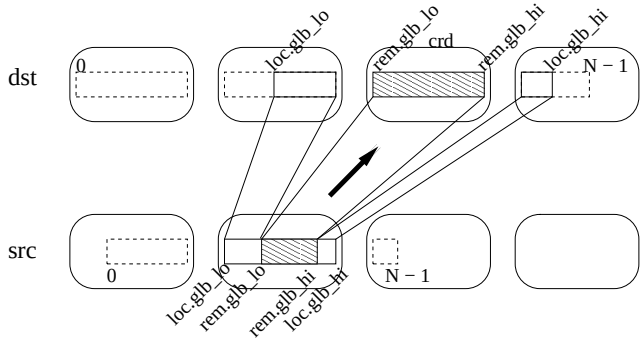

Fig. 7. Illustration of `sendLoop` operation for `remap`

with a scalar section to broadcast an element of a general array to all members of the active process group. The second form of `broadcast()` just takes an ordinary Java value as the source. This value should be defined on the process or group of processes identified by `root`. It is broadcast to all members of the active process group.

Adlib has some support for irregular communications in the form of collective `gather()` and `scatter()` operations. The simplest form of the `gather` operation for one-dimensional arrays has prototypes

```
void gather(T [[-]] destination, T [[-]] source, int [[-]] subscripts) ;
```

The `subscripts` array should have the same shape as, and be aligned with, the `destination` array. In pseudocode, the `gather` operation is equivalent to

```
for all i in {0, ..., N − 1} in parallel do
  destination [i] = source [subscripts[i]] ;
```

where N is the size of the `destination` (and `subscripts`) array.

The basic `scatter` function has very similar prototypes, but the names `source` and `destination` are switched. Currently the HPJava version of Adlib does not support combining scatters, although these could be added in later releases.

You can find complete list of Adlib schedules in [12]. Information, API, and usage on the each schedule are described in this paper.

4 A Multigrid Application and Benchmark Results

The multigrid method [5] is a fast algorithm for solution of linear and nonlinear problems. It uses a hierarchy or stack of grids of different granularity (typically with a geometric progression of grid-spacings, increasing by a factor of two up from finest to coarsest grid). Applied to a basic relaxation method, for example, multigrid hugely accelerates elimination of the residual by restricting a smoothed

version of the error term to a coarser grid, computing a correction term on the coarse grid, then interpolating this term back to the original fine grid. Because computation of the correction term on the fine grid can itself be handled as a relaxation problem, the strategy can be applied recursively all the way up the stack of grids.

The experiments were performed on the SP3 installation at Florida State University. The system environment for SP3 runs were as follows:

- System: IBM SP3 supercomputing system with AIX 4.3.3 operating system and 42 nodes.
- CPU: A node has Four processors (Power3 375 MHz) and 2 gigabytes of shared memory.
- Network MPI Settings: Shared "css0" adapter with User Space(US) communication mode.
- Java VM: IBM 's JIT
- Java Compiler: IBM J2RE 1.3.1

For best performance, all sequential and parallel Fortran and Java codes were compiled using -O5 or -O3 with -qhot or -O (i.e. maximum optimization) flag.

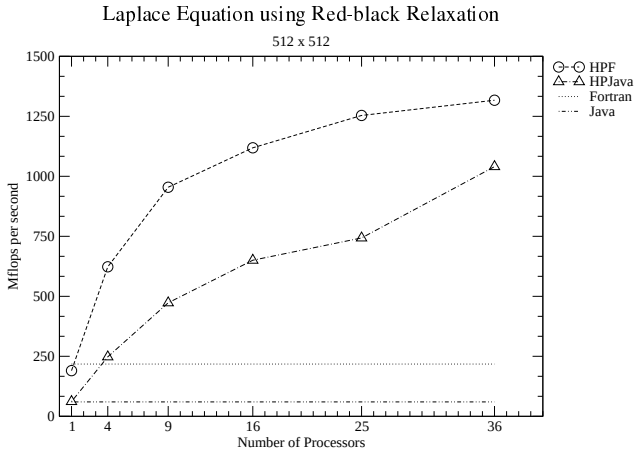

Fig. 8. Laplace Equation with Size of 512^2.

First we present some results for the computational kernel of the multigrid code, namely unaccelerated red-black relaxation algorithm. Figure 8 gives our results for this kernel on a 512 by 512 matrix. The results are encouraging. The HPJava version scales well, and eventually comes quite close to the HPF code (absolute megaflop performances are modest, but this feature was observed for all our codes, and seems to be a property of the hardware).

The flat lines at the bottom of the graph give the sequential Java and Fortran performances, for orientation. We did not use any auto parallelization feature

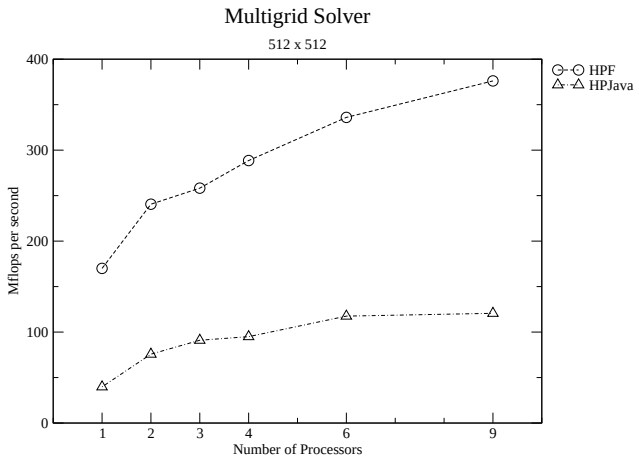

Fig. 9. Multigrid solver with size of 512^2.

here. Corresponding results for the complete multigrid code are given in Figure 9. The results here are not as good as for simple red-black relaxation-both HPJava speed relative to HPF, and the parallel speedup of HPF and HPJava are less satisfactory.

The poor performance of HPJava relative to Fortran in this case can be attributed largely to the naive nature of the translation scheme used by the current HPJava system. The overheads are especially significant when there are many very tight overall constructs (with short bodies). Experiments done elsewhere [11] leads us to believe these overheads can be reduced by straightforward optimization strategies which, however, are not yet incorporated in our source-to-source translator.

The modest parallel speedup of both HPJava and HPF is due to communication overheads. The fact that HPJava and HPF have similar scaling behavior, while absolute performance of HPJava is lower, suggests the communication library of HPJava is slower than the communications of the native SP3 HPF (otherwise the performance gap would close for larger numbers of processors). This is not too surprising because Adlib is built on top of a portability layer called *mpjdev*, which is in turn layered on MPI. We assume the SP3 HPF is more carefully optimized for the hardware. Of course the lower layers of Adlib could be ported to exploit low-level features of the hardware (we already did some experiments in this direction, interfacing Java to LAPI [13]).

5 Conclusions and Future Work

We have explored enabling parallel, high-performance computation-in particular development of scientific software in the network-aware programming language, Java. Traditionally, this kind of computing was done in Fortran. Arguably, Fortran is becoming a marginalized language, with limited economic incentive for

vendors to produce modern development environments, optimizing compilers for new hardware, or other kinds of associated software expected by today's programmers. Java looks like a promising alternative for the future.

We have discussed in detail the design and development of high-level library for HPJava-this is essentially communication library. The Adlib API is presented as high-level communication library. This API is intended as an example of an application level communication library suitable for data parallel programming in Java. This library fully supports Java object types, as part of the basic data types. We discussed implementation issues of collective communications in depth. The API and usage of other types of collective communications were also presented.

References

1. mpiJava Home Page. http://www.hpjava.org/mpiJava.html.
2. Timber Compiler Home Page. http://pds.twi.tudelft.nl/timber.
3. Titanium Project Home Page. http://www.cs.berkeley.edu/projects/titanium.
4. A. Agrawal, A. Sussman, and J. Saltz. An integrated runtime and compiletime approach for parallelizing structured and block structured applications. *IEEE Transactions on Parallel and Distributed Systems*, 6, 1995.
5. William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial*. The Society for Industrial and Applied Mathematics (SIAM), 2000.
6. Bryan Carpenter, Guansong Zhang, and Yuhong Wen. NPAC PCRC runtime kernel definition. Technical Report CRPC-TR97726, Center for Research on Parallel Computation, 1997. Up-to-date version maintained at http://www.npac.syr.edu/projects/pcrc/doc.
7. Parallel Compiler Runtime Consortium. Common runtime support for high-performance parallel languages. In *Supercomputing '93*. IEEE Computer Society Press, 1993.
8. Jayant DeSouza and L. V. Kale. Jade: A parallel message-driven java. In *Proceedings of the 2003 Workshop on Java in Computational Science*, Melbourne, Australia, 2003. Available from `http://charm.cs.uiuc.edu/papers/ParJavaWJCS03.shtml`.
9. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification,* Second Edition. Addison-Wesley, 2000.
10. HPJava project home page. www.hpjava.org.
11. Han-Ku Lee. *Towards Efficient Compilation of the HPJava Language for High Performance Computing*. PhD thesis, Florida State University, June 2003.
12. Sang Boem Lim. *Platforms for HPJava: Runtime Support for Scalable Programming in Java*. PhD thesis, Florida State University, June 2003.
13. Sang Boem Lim, Bryan Carpenter, Geoffrey Fox, and Han-Ku Lee. A device level communication library for the hpjava programming language. In *the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, November 2003.
14. Guansong Zhang, Bryan Carpenter, Geoffrey Fox, Xiaoming Li, Xinying Li, and Yuhong Wen. PCRC-based HPF compilation. In Zhiyuan Li et al, editor, *10th International Workshop on Languages and Compilers for Parallel Computing*, volume 1366 of *Lecture Notes in Computer Science*. Springer, 1997. http://www.hpjava.org/pcrc/npacWork.html.