Collective Communications for HPJava

Sang Boem Lim*, Bryan Carpenter[§], Geoffrey Fox[§], and Han-ku Lee[†]

* Supercomputing Application Technology Department,
Korea Institute of Science and Technology Information (KISTI)
DaeJeon, Republic of Korea
slim@kisti.re.kr

[§]Community Grid Labs, Indiana University
Bloomington, IN 47404 USA
{dbcarpen, gcf }@indiana.edu

[†]School of Internet & Multimedia Engineering, Konkuk University
Seoul, Republic of Korea
hlee@konkuk.ac.kr

Abstract—We discuss implementation of high-level collective communication library, called Adlib, for scalable programming in Java. We are using Adlib as basis of our system, called HPJava. Many functionalities of Java version of high-level communication library is following its predecessor–C++ library developed by in the Parallel Compiler Runtime Consortium (PCRC). 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.

Keywords : Collective Library, HPJava, High-performance communication Library

1.0 Bakground

A C++ library called *Adlib* [2] was completed in the Parallel Compiler Runtime Consortium (PCRC) [3] project. It was a high-level runtime library designed to support translation of data-parallel languages. Initial emphasis was on High Performance Fortran (HPF), and two experimental HPF translators used the library to manage their communications [4] [5]. 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.

Initially HPJava used a JNI wrapper interface to the C++ kernel of the PCRC library. The library described here borrows many ideas form the PCRC library, but for this project we rewrote high-level library from the scratch for Java. It was extended to support Java object types, to target Java based communication platforms and to use Java exception handling—making it "safe" for Java.

2.0 Features of HPJava

In this section, we present a high level overview of our HPJava. Some predefined classes and some extra syntax for dealing with distributed arrays are added into the basic language, Java. We will briefly review the features of the HPJava language by showing simple HPJava examples. In this section, we will only give an overview of features. Detailed description of those features will be presented in the Section 3.

```
float [[*, *]] c = new float [[M, N]];
float [[*, *]] a = new float [[M, L]];
float [[*, *]] b = new float [[L, N]];

... initialize 'a', 'b'

for(int i = 0; i < M; i++)
  for(int j = 0; j < N; j++) {

    c [i, j] = 0;
    for(int k = 0; k < L; k++)
      c [i, j] += a [i, k] + b [k, j];
  }
```

Figure 1: Sequential Matrix multiplication in HPJava.

Figure 1 is a basic HPJava program for sequential matrix multiplication. This program is simple and similar to the ordinary Java program. It uses simple sequential *multiarrays*—a feature HPJava adds to standard Java. A multiarray uses double brackets to distinguish the type signature from a standard Java array. The multiarray and ordinary Java array has many similarities. Both arrays have some index space and store a collection of elements of fixed type. Syntax of accessing a multiarray is very similar with accessing ordinary Java array which uses single brackets, but an HPJava sequential multiarray uses double bracket and asterisks for its type signature. The most significant difference between ordinary Java array and the multiarray of HPJava is that the distributed array is *true multi-dimensional array* like the arrays of Fortran, while ordinary Java only provides arrays of arrays. These features of Fortran arrays have adapted and evolved to support scientific and parallel algorithms.

We can create a general purpose matrix multiplication routine that works for arrays with any distributed format (Figure 2). This program takes arrays which may be distributed in both their dimensions, and copies into the temporary array with a special distribution 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 dissertation, the most important part of this code is communication method. We can

divide the communication library of HPJava into two parts: the high-level *Adlib* library, and the low-level *mpjdev* library. The Adlib library is responsible for the collective communication schedules and the mpjdev library is responsible for the actual communication. 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

```
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;
      }
}
```

Figure 2: A general Matrix multiplication in HPJava.

3.0 Implementation of Collectives

In this section we will discusses Java implementation of the Adlib collective operations. 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** in [1]. 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.

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

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

```
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() { ... }

  ...
}
```

Figure 3: API of the class BlockMessSchedule.

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 [1]).

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.

Some of the relevant virtual functions of the range are displayed in the API of Figure 4. 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.

```
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) {...}
}
```

Figure 4: Partial API of the class Range.

4.0 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.

The initial release of HPJava was made on April 1, 2003. It is freely available from **www.hpjava.org**. This release includes complete HPJava translator, two implementations of communication libraries (mpiJava-based and multithreaded), test suites, and all the applications described in this dissertation. In the future, further optimization of the HPJava translator is needed.

5.0 Reference

[1] 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.

[2] Bryan Carpenter, Guansong Zhang, and Yuhong Wen. *NPAC PCRC runtime kernel definition*. Technical Report CRPC-TR97726, Center for Research on Parallel Computation, 1997.

[3] Parallel Compiler Runtime Consortium. *Common runtime support for high performance parallel languages*. In Supercomputing ‘93. IEEE Computer Society Press, 1993.

[4] John Merlin, Bryan Carpenter, and Tony Hey. *shpf: a subset High Performance Fortran compilation system.* Fortran Journal, pages 2-6, March 1996.

[5] 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.