Benchmarking HPJava: Prospects for Performance

Han-Ku Lee, Bryan Carpenter, Geoffrey Fox, Sang Boem Lim
{*hkl, dbc, fox, slim*}*@csit.fsu.edu*

Pervasive Technology Labs
Indiana University
Bloomington, Indiana 47401-3730

School of Computational Science and Information Technology,
Florida State University,
Tallahassee, Florida 32306-4120

February 8, 2002

Abstract

The *HPspmd programming language model* is a flexible hybrid of HPF-like data-parallel language features and the popular, library-oriented, SPMD style, omitting some basic assumptions of the HPF model. Here, we will discuss a Java-based HPspmd language, called *HPJava*. HPJava extends the Java language with some additional syntax and pre-defined classes for handling distributed arrays, and a few new control constructs. We discuss the compilation system, including distributed array types, HPspmd classes, translation schemes, optimization strategies, benchmarks and the current status of the HPJava system.

1 Introduction

Historically, data parallel programming and data parallel languages have played a major role in high-performance computing. By now we know many of the implementation issues, but we remain uncertain what the high-level programming environment should be. Based on the observations that compilation of HPF [8] is such difficult problem while library based approaches more easily attain acceptable levels of efficiency, a hybrid approach called *HPspmd* has been proposed [4].

The major goal of the system we are building is to provide a programming model which is a flexible hybrid of HPF-like data-parallel language and the popular, library-oriented, SPMD style without the basic assumptions of the HPF model. We refer to this model as the *HPspmd programming language model*. It incorporates syntax for representing distributed arrays, for expressing that some computations are localized to some processors, and for writing a distributed form of the parallel loop. Crucially, it also supports binding from the extended languages to various communication and arithmetic libraries. These might involve simply new interfaces to some subset of PARTI [1], Global Arrays [10], Adlib [7], MPI [9], and so on. Providing the libraries for *irregular* communication may well be important. Evaluating HPspmd programming language model on large scale applications is also an important issue.

2 HPSpmd Programming Language Model

2.1 HPspmd Language Extensions

In order to support a flexible hybrid of the data parallel and low-level SPMD approaches, we need HPF-like distributed arrays as language primitives in our model. All accesses to non-local array elements, however will be done via library functions such as calls to a collective communication libraries, or simply `get` or `put` functions for access to remote blocks of a distributed array. This explicit communication encourages the programmer to write algorithms that exploit locality, and greatly simplifies the compiler developer's task.

The HPspmd model we discuss here has some similar characteristics to the HPF model. But, the HPF-like semantic equivalence between the data parallel program and a sequential program is given up in favor of a literal equivalence between the data parallel program and an SPMD program. Understanding a SPMD program is obviously more difficult than understanding a sequential program. This means that our language model may be a little bit harder to learn and use than HPF. In contrast, understanding the performance of a program should be easier.

An important feature of the HPspmd programming language model is that if a portion of HPspmd program text looks like program text from the unenhanced base language, it doesn't need to be translated and behaves like a local sequential code. Only statements including the extended syntax are translated. This makes preprocessor-based implementation of the HPspmd model relatively straightforward, allows sequential library codes called directly, and allows the programmer control over generated codes.

2.2 Integration of High-Level Libraries

Libraries play a most important role in our HPspmd programming language model. In a sense, the HPspmd language extensions are simply a framework to make calls to libraries that operate on distributed arrays. Thus, an essential component of our HPspmd model is to define a series of bindings of SPMD libraries and environments in HPspmd languages.

Various issues must be mentioned in interfacing to multiple libraries. For instance, low-level communication or scheduling mechanisms used by the different libraries might be incompatible. As a practical matter, these incompatibilities must be mentioned, but the main thrust of the proposed research is at the level of designing compatible interfaces, rather than solving interference problems in specific implementations.

The survey of run-time communication libraries for our HPspmd model is well-documented in [4]. Our HPJava system is using Adlib and MPJ libraries which are Java Native Interface (JNI) wrapper. Currently, we are developing *pure-Java* versions of the run-time libraries, Adlib and MPJ to achieve the aim of Java—write–once–run–anywhere—for parallel computing.

2.3 The HPJava Language

HPJava [5, 6] is an implementation of our HPspmd programming language model. It extends the Java language with some additional syntax and with some pre-defined classes for handling distributed arrays. The distributed array model is adopted from the HPF array model. But, the programming model is quite different from HPF. It is one of explicitly cooperating processes. All processes carry out the same program, but the components of data structures are divided across processes. Each process operates on locally held segment of an entire distributed array.

```
Procs2 p = new Procs2(2, 3) ;
on(p) {
  Range x = new ExtBlockRange(N, p.dim(0)) ;
  Range y = new ExtBlockRange(N, p.dim(1)) ;

  double [[-,-]] a = new double [[x, y]] ;

  ... initialization for 'a'

  for(int iter=0; iter<count; iter++){

    Adlib.writeHalo(a, wlo, whi);

    overall(i = x for 1 : N - 2)
      overall(j = y for 1 + (i' + iter) % 2 : N - 2 : 2) {
        a[i, j] = 0.25F * (a [i - 1, j] + a [i + 1, j] +
                           a [i, j - 1] + a [i, j + 1]);
  }
}
```

Figure 1: A red-black iteration in HPJava.

Figure 1 is a basic HPJava program for a red-black iteration. `Procs2` is a subclass of the special base class `Group`. It describes 2-dimensional grids of processes. The distributed range class `ExtBlockRange` is a library class derived from `Range`. It represents a range of subscripts to create arrays with ghost regions over a specific process dimension. The `on` construct limits control to processes in its parameter group. The code in the `on` construct is *only* executed by processes that belong to `p`. The `on` construct fixes `p` as the *active process group* within its body.

The most important feature HPJava adds to Java is the *distributed array*. A distributed array is a collective object shared by a number of processes. Like an ordinary array, a distributed array has some index space and stores a collection of elements of fixed type. Unlike an ordinary array, the index space and associated elements are scattered across the processes that share the array. There are some similarities and differences between HPJava distributed arrays and the ordinary Java arrays. Aside from the way that elements of a distributed array are distributed, the distributed array of HPJava is a true multi-dimensional array like that of Fortran. Like in Fortran, one can form a *regular section* of an array. These features of Fortran arrays are adapted and evolved to support scientific and parallel algorithms.

The `overall` construct is another control construct of HPJava. It represents a distributed parallel loop, sharing some characteristics of the *forall* construct of HPF. The symbol `i` scoped by the `overall` construct is called a *distributed index* Its value is a *location*, rather an abstract element of a distributed range than an integer value. It is very important that with a few exceptions we will mention later, the subscript of a distributed array must be a distributed index, and the location should be an element of the range associated with the array dimension. This restriction is a important feature, ensuring that referenced array elements are locally held. The i' is read "i-primed", and yields the integer *global index* value for the distributed loop. A library function called `Adlib.writeHalo` updates the cached values in the ghost regions with proper element values from neighboring processes.

There are some other language extensions (not shown from Figure 1) in HPJava.

HPJava supports subarrays modeled on the *array sections* of Fortran 90. The ranges of an array section are called *subranges*. A *restricted group* is the subset of processes in some parent group to which a specific location is mapped. Using these subranges and subgroups, HPJava can be relatively simpler since it can avoid all the alignment options of HPF.

3 Compilation Strategies for HPJava

3.1 Distributed Array Types and HPspmd Classes

A distributed array type is not treated as a class type. If we said, "distributed arrays have a class", it would probably commit us to either extending the definition of class in the Java base language, or creating genuine Java classes for each type of HPJava array that might be needed. The fact that a distributed array is not a member of any Java class means that a distributed array *cannot* be an element of an ordinary Java array, nor can a distributed array reference be stored in a standard library class like `Vector`, which expects an `Object`. In practise, this is not such a big restriction as it sounds. We *do* allow distributed arrays to be a members of classes. So, the programmer can make wrapper classes for specific types of distributed array.

The HPJava translator tries to tell HPJava code from Java code. It introduces a special interface, `hpjava.lang.HPspmd`, which must be implemented by any class that uses the special syntax. An *HPspmd class* is a class that implements the `hpjava.lang.HPspmd` interface. The extended syntax of HPJava can only be used in methods, constructors and fields declared in HPspmd classes and interfaces. The details of HPspmd class restrictions can be found in [6].

Many of the special operations in HPJava rely on the knowledge of the currently active process group—the *APG*. This is a context value that will change during the course of the program as distributed control constructs limit control to different subsets of processors. In the current HPJava translator the value of the APG is passed as a hidden argument to methods and constructors of HPspmd classes.

3.2 Translation Scheme

All of current translation schemes is documented in the HPJava manual [6] and translation scheme [3]. The schemes are now in progess. Thus, the document will be updated as the translator becomes evolved.

3.3 Optimization Strategies

Based on the observations for parallel algorithms such as Laplace equation using red-black iterations, Laplace equation using jacobi relaxations, etc, a distributed array element access is generally located in inner `overall` loops. The main issue of our optimization strategies is the complexity of the associated terms in the subscript expression of a distributed array element access. The following optimization strategies should remove overheads of some naive translation schemes (especially for `overall` construct), and speed up HPJava, i.e. produce a Java-based environment competitive with (and perhaps ultimately, superior to) existing Fortran programming environments..

To eliminate complicated distributed index subscript expressions involving multiplication in the inner loops, the new translation scheme adopt strength-reduction optimizations. This is achieved by introducing the induction variables which can be computed

HPJava	Naive	Strength Reduction	Loop unrolling
Mflops	113.3	163.7	219.5

Table 1: Benchmarks of Laplace equation using red-blck iteration for HPJava version

Mflops	1-dimensional array	2-dimensional array	Strength Reduction
IBM JIT	54.5	55.7	215.7
C++	188.6	225.5	239.6
F77	231.8	231.8	240.8

Table 2: Benchmarks of Laplace equation for Java, C++, and F77 versions

efficiently by incrementing at suitable points with the induction increments. Another strategy is to apply loop-unrolling optimization for hoisting the special run-time support classes such as `Block` and `Group`. From the original `overall` translation scheme, we use the `localBlock()` method to compute parameters of the local loop, this translation is identical for every distribution format—block-distribution, simple-cyclic distribution, aligned subranges, and so on—supported by the language. Of course there is an overhead related to abstracting this local-block parameter computation into a method call; the method call is made at most once at the start of each loop. Moreover, we can apply common-subexpression elimination since there are many local variables newly declared for each `overall` loop. Naive translation strategy for inner loops tends to repeatly delare some variables for holding `str()` methods, global bases and steps, and so on.

Currently the optimization strategies have been "manually" applied to some HPJava programs such as Lapalce equation using red-black with 500 iterations. They have been tested using IBM Developer Kit 1.3 (JIT) with `-O` flag on Pentium4 1.5GHz Red Hat 7.2 Linux machines. The major performance improvement has been achieved by strength-reduction optimization. This strategy makes the program 50% faster than the naive translation. Also, adopting loop-unrolling optimization makes newly translated version with strength-reduction optimization 27% faster. The benchmarks on one-processor are in Table 1.

We also compared the sequential Java, C++, and Fortran version of the HPJava program, all with `-O` (i.e. maximun optimization) flag when compiling. Like recent benchmarking results from [2], Java JIT performance is identical with C++, and even Fortran GNU compilers. The most important thing is the performance of the Laplace equation in HPJava is approaching parity with that of Java, C++, and Fortran. This means that the overheads introduced by run-time support classes have been almost (not perfectly yet) removed by the above optimization strategies. The results are in Table 2.

4 Conclusion and Current Status of HPJava

The first fully functional version of the HPJava translator is now operational. The system has been tested and debugged against a small test suite of available HPJava programs. Currently most of the examples are short, although the suite does include an 800-line Multigrid code, transcribed from an existing Fortran 90 program. One pressing concern over the next few months is to develop a much more substantial body of test code and applications.

In our source-to-source translation strategy, this means that standard Java statements and expressions are copied through essentially unchanged. On the other hand the

inclusion of Java means that we do need a front-end that covers the whole of Java. The translation scheme for HPJava depends in an important way on type information. It follows that we need type analysis for the whole language, including the Java part. Writing a full type-checker for Java is not trivial (especially since the introduction of nested types). So far, development of the front-end, especially the type-checker, has been the most time-consuming step in developing the whole system.

5 Acknowledgements

This work was supported in part by the National Science Foundation Division of Advanced Computational Infrastructure and Research, contract number 9872125.

References

[1] A. Agrawal, A. Sussman, and J. Saltz. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Transactions on Parallel and Distributed Systems*, 6, 1995.

[2] M. Bull, L. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for Scientific Applications. In *ACM 2001 Java Grande/ISCOPE Conference*. ACM Press, 2001.

[3] Bryan Carpenter, Geoffrey Fox, Han-Ku Lee, and Sang Boem Lim. Translation Schemes for the HPJava Parallel Programming Language. In *11th International Workshop on Languages and Compilers for Parallel Computing 2001*, 2001.

[4] Bryan Carpenter, Geoffrey Fox, and Guansong Zhang. An HPspmd Programming Model Extended Abstract. 1999. http://aspen.csit.fsu.edu/pss/HPJava.

[5] Bryan Carpenter, Guansong Zhang, Geoffrey Fox, Xiaoming Li, Xinying Li, and Yuhong Wen. Towards a Java environment for SPMD programming. In David Pritchard and Jeff Reeve, editors, *4th International Europar Conference*, volume 1470 of *Lecture Notes in Computer Science*. Springer, 1998. http://aspen.csit.fsu.edu/pss/HPJava.

[6] Bryan Carpenter, Guansong Zhang, Han-Ku Lee, and Sang Lim. *Parallel Programming in HPJava*. Draft, 2001. http://aspen.csit.fsu.edu/pss/HPJava.

[7] 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.

[8] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, special issue, 2, 1993.

[9] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tenessee, Knoxville, TN, June 1995. http://www.mcs.anl.gov/mpi.

[10] J. Nieplocha, R.J. Harrison, and R.J. Littlefield. The Global Array: Non-uniform-memory-access programming model for high-performance computers. *The Journal of Supercomputing*, (10):197–220, 1996.