Data Parallel SPMD Programming Models from Fortran to Java

— A Proposal to the NSF in response to Advanced Computational Infrastructure and Research Programs, New Technologies Program

Geoffrey C. Fox

Northeast Parallel Architectures Center at Syracuse University
Syracuse, New York 13244-4100
315-443-2163, {gcf,dbc,lxm,wen,zgs}@npac.syr.edu

March 14, 1998

A A. Project summary

B Table of Contents

C Project Description

C.1 Motivation of the proposed work

It is generally accepted that data parallel programming has a vital role in high-performance scientific computing. The basic implementation issues related to this paradigm are well understood. But the choice of high-level programming environment remains uncertain. Five years ago the High Performance Fortran Forum published the first standardized definition of a language for data parallel programming. In the intervening period considerable progress has been made in HPF compiler technology, and the HPF language definition has been extended and revised in response to demands of compiler-writers and end-users. Yet it seems to be the case that most programmers developing parallel applications—or environments for parallel application development—*do not* code in HPF. The slow uptake of HPF can be attributed in part to immaturity in the current generation of compilers. But there is the suspicion that many programmers are actually more comfortable with the lower-level Single Program Multiple Data (SPMD) programming style, perhaps because the effect of executing an SPMD program is more controllable, and the process of tuning for efficiency is more intuitive.

SPMD programming has been very successful. There are countless applications written in the most basic SPMD style, using direct message-passing through MPI or similar low-level packages. Many higher-level parallel programming environments and libraries assume the SPMD style as their basic model. Examples include ScaLAPACK, PetSc, DAGH, Kelp, the Global Array Toolkit and NWChem. While there remains a prejudice that HPF is best suited for problems with very regular data structures and regular data access patterns, SPMD frameworks like DAGH and Kelp have been designed to deal directly with irregularly distributed data, and other libraries like CHAOS and Global Arrays support unstructured access to distributed arrays. These successes aside, the library-based SPMD approach to data-parallel programming certainly lacks the uniformity and elegance of HPF. All the environments referred to above have some idea of a distributed array, but they all describe those arrays differently. Compared with HPF, creating distributed arrays and accessing their local and remote elements is clumsy and error-prone. Because the arrays are managed entirely in libraries, the compiler offers little support and no safety net of compile-time checking.

The proposed work will investigate a class of programming languages that borrow certain ideas, various run-time technologies, and some compilation techniques from HPF, but relinquish some of its basic tenets, in particular: that the programmer should write in a language with (logically) a single global thread of control, that the compiler should determine automatically which processor executes individual computations in a program, and that the compiler should automatically insert communications if an individual computation involves accesses is to array element held outside the local processor.

If these foundational assumptions are removed from the HPF model, does anything useful remain? In fact, yes. What will be retained is *an explicitly SPMD programming model* complemented by syntax for representing distributed arrays, syntax for expressing that certain computations are localized to certain processors, and syntax for expressing concisely a *distributed form* of the parallel loop. The claim is that these are just the features needed to make calls to various data-parallel libraries, including application-oriented libraries and high-level libraries for communication (needed explictly now the compiler relinquishes responsiblity for communication), about as convenient as, say, making a call to an array transformational intrinsic function in Fortran 90. We will illustrate that, besides their advantages as a framework for library usage, the resulting programming languages can conveniently express various practical data-parallel algorithms.

This proposal brings together several important research areas including parallel compilers, data parallel SPMD libraries and object oriented programming models. We research combinations of these ideas which achieve high performance with an approach that implies more work for the programmer than envisaged in systems such as HPF, but can more clearly be implemented in a robust fashion on a range of languages. Explicitly we are combining our research on the use of Java and Web technologies with the high performance SPMD libraries and some of the compiler techniques developed as part of HPF research. Java has many features that suggest it could be a very attractive language for scientific and engineering or what we now term "Grande" applications. Clearly Java needs many improvements both to the language and the support environment to achieve the required linkage of high performance with expressivity. This cannot be guaranteed but we have helped set in motion a community activity involving academia, government and industry (including IBM, Intel, Microsoft, Oracle, Sun and perhaps most importantly James Gosling from Javasoft) which is designed to both address language changes and the establishment of standards for numerical libraries and distributed scientific objects. The Java environment is still malleable and we are optimistic that this effort will be succesful and Java will emerge as a premier language for large scale computation. Our research will be aimed at multi-language programming paradigms but our new implementations will focus on Java exploiting existing high performance C++ and Fortran libraries. Our collaborater Professor Xiaoming Li from Peking University will be developing the Fortran and C++ aspects of this general high level SPMD environment. We can consider our work from either of two points of view; bringing the power of Java to a data parallel SPMD environment or alternatively researching the expression of data parallelism within Java. Note that we are adopting a more modest approach than a full scale data parallel compiler like HPF; we believe this is an appropriate approach to Java where the situation is changing rapidly and one needs to be very flexible.

We should stress what we are not doing! Many of the discussions of Java at the recent "Grande" workshops have focussed on its use in distributed object and mobile or Web client based computing. In fact our group also is looking into this for composing large scale distributed systems. However in this proposal, we are addressing "hard-core" science and engineering computations where data parallelism and the highest performance are viewed as critical.

C.2 Objective and expected significance

The main thrust of the proposed work will be on researching compiler (or preprocessor) support for our extended languages, and development of exemplar interfaces from the new languages to a subset of the libraries mentioned above. The research aspects of the proposed work involve investigation of compiler optimizations and safety checks peculiar to the new languages, extensions to the basic language model to improve support of irregular problems, and design of attractive class-library bindings for the various SPMD environments involved in the project.

Primary goals of the current project therefore include

1. Providing a small set of syntax extensions to various base languages (including Java, Fortran, and C++). These syntax extensions add distributed arrays as language primitives, and introduce a few new control constructs, such as the distributed loop.

2. Providing bindings from the extended languages to various communication and arithmetic libraries. These may include libraries modelled on, or simply new interfaces to, some subset of Adlib, CHAOS, Global Arrays, MPI, DAGH, ScaLAPACK, etc.

The resulting system will support a programming model that is a flexible hybrid of the data-parallel, language-oriented, HPF style, and the established and popular, library-oriented, SPMD style. We refer to this model as *HPspmd*.

[Summarize what is in following subsections.]

C.2.1 HPspmd language extensions

We aim to provide a flexible hybrid of the data parallel and low-level SPMD approaches. To this end HPF-like distributed arrays appear as language primitives. A design decision is made that all access to *non-local* array elements should go through library functions—for example, calls to a collective communication library, or simply `get` and `put` functions for access to remote blocks of a distributed array. This puts an extra onus on the programmer; but making communication explicit encourages the programmer to write algorithms that exploit locality, and simplifies the task of the compiler writer.

For the newcomer to HPF, one of its advantages lies in the fact that the effect of a particular operation is logically identical to its effect in the corresponding sequential program. This means that, assuming the programmer understands conventional Fortran, it is very easy for him or her to understand the behaviour of a program at the level of what values are held in program variables, and the final results of procedures and programs. Unfortunately, the ease of understanding this "value semantics" of a program is counterbalanced by the difficulty in knowing exactly how the compiler translated the program. Understanding the *performance* of an HPF program may require the programmer to have very detailed knowledge of how arrays are distributed over processor memories, and what strategy the compiler adopts for distributing computations across processors.

The language model we discuss has various similarities to the HPF model, but the HPF-style semantic equivalence between the data-parallel program and a sequential program is abandoned in favour of a literal equivalence between the data-parallel program and an SPMD program. Because understanding an SPMD program is presumably more difficult than understanding a sequential program, learning naive use of our language may be slightly harder than for HPF.

The distributed arrays of the new languages will be kept strictly separate from ordinary arrays. They are a different kind of object, not type-compatible with ordinary arrays. An important property of the languages we describe is that if a section of program text *looks like* program text from the unenhanced base language (Java or Fortran 90 for example), it is translated exactly as for the base language—as local sequential code. Only statements involving the extended syntax behave specially. This makes preprocessor-based implementation of the new languages very straightforward, allows sequential library code to be called directly, and gives the programmer good control over the generated code—he or she can be confident no unexpected overhead have been introduced in code that looks like ordinary Fortran (say).

In the baseline language we adopt a distributed array model semantically equivalent to to the HPF data model in terms of how elements are stored, the options for distribution and alignment, and facilities for describing *regular sections* of arrays. Distributed arrays may be subscripted with global subscripts, as in HPF. *But* a subscripting operation must not imply access to an element on a different processor. We will sometimes be refer to this restriction as the *SPMD constraint*. To simplify the task of the programmer, who must ensure an accessed element is held locally, the languages will typically add *distributed control* constructs. These play a role something like the `ON HOME` directives of HPF 2.0 and earlier data parallel languages. A further special control construct will facilitate access to elements in the locally held section of a particular array (or group of aligned arrays). This is the *distributed loop* or *overall construct*.

Java, Fortran and C++ versions. A Java instantiation (HPJava) of the HPspmd language model described above has been described *[blah]*. A brief review is given in section C.4.1.

In reference *[HIPS '98 paper]* we have outlined possible syntax extensions to Fortran to provide similar semantics to HPJava. As emphasized previously, a distinguishing property of the proposed system, compared to HPF, is that it includes ordinary Fortran as a strict subset, *and ordinary Fortran constructs are unchanged by the translator.* The suggested system would *not* attempt to exploit parallelism even in "explicitly parallel" constructs such as the array syntax of Fortran 90 or the FORALL statement of Fortran 95. This policy drastically simplifies the translator, and gives the programmer much finer control over the generated code.

So far as C++ is concerned, a working prototype of our language model exists in the form of the ad++ interface to Adlib. This extends C++ only by class libraries and macros. In C++ we can use features like operator-overloading, templates, reference-valued functions, and macros to effectively prototype new language constructs. *But* the current ad++ is very inefficient (and the concrete syntax is quite clumsy) compared with what could be achieved with a purpose-built compiler or preprocessor.

In the proposed work, research into optimizing compilers and preprocessor for HPspmd versions of Fortran and C++ will be led by our collaborater Professor Xiaoming Li from Peking University.

General translation issues. The language extensions described earlier were devised partly to provide a convenient interface to a distributed-array library developed in the Parallel Compiler Runtime Consortium (PCRC) project.

Compared with HPF, translation of the HPspmd languages is very straightforward. The HPJava compiler, for example, is being implemented initially as a translator to ordinary Java, through a compiler construction framework developed in the PCRC project. The distributed arrays of the extended language appear in the emitted code as a pair—an ordinary Java array of local elements and a Distributed Array Descriptor object (DAD). In the initial implementation, details of the distribution format, including non-trivial details of global-to-local translation of the subscripts, are managed in the runtime library.

Even with these overheads, acceptable performance is achievable, because in useful parallel algorithms most work on distributed arrays occurs inside *overall* constructs with large ranges. In normal usage, the formulae for address translation can be linearized inside these constructs, and the cost of runtime calls handling non-trivial aspects of address translation (including array bounds checking) can be amortized in the startup overheads of the loop. If array accesses are genuinely irregular, the necessary subscripting cannot usually be *directly* expressed in our language; subscripts cannot be computed randomly in parallel loops without violating the SPMD restriction that accesses be local. This is not necessarily a shortcoming: it forces explicit use of an appropriate library package for handling irregular accesses (such as CHAOS, see section **??**).

The basic HPJava translator will be available by the start date of the proposed work.

In figure 1 we give benchmark results for HPJava examples manually converted to Java, following the translation scheme outlined above. The examples are essentially the ones described in section C.4.1. above. The parallel programs are executed on 4 sparc-sun-solaris2.5.1 using MPICH and the Java JIT compiler in JDK 1.2Beta2, through a JNI interface to Adlib for collective communications. In both cases arrrays are 1024 by 1024. For Jacobi iteration, the timing is for about 90 iterations. Timings are compared the *sequential* Java and C++ version of the code (horizontal lines). Note that poor scaling in the Cholesky case is attributable to the poor performance of MPICH on this platform *not* overheads of HPJava. Scaling will be much improved by using SunHPC MPI.

The single-processor HPJava performance is better than sequential Java, because the pure Java version was coded in the natural way, using two-dimensional arrays, which are quite inefficient in Java. The HPJava translation scheme linearizes arrays. Although absolute performance is still somewhat lower than C++, Java performance has improved dramatically over the last year, and we expect to see further gains. Parity between Java and C no longer seems and unrealistic expectation. In fact, even if the performance of Java does not rapidly approach that of C and Fortran, Java remains an excellent research platform for the general language model we espouse. It combines strong support for dynamic and object-oriented programming in a *relatively simple* language, for which preprocessors for extended versions of the language ("little languages") are a straightforward proposition. (We remark that in recent workshops James Gosling has stated that this is the approved approach to introducing generalized array-like structure in Java!)

C.2.2 Integration of high-level libraries, regular and irregular

Libraries are at the heart of the HPspmd model. From one point of view, the language extensions are simply a framework for invoking libraries that operate on distributed arrays. The base language model was originally motivated by work on HPF runtime libraries carried out in the Parallel Compilers Runtime Consortium (PCRC) project at Syracuse (and earlier related work by one of us).

Hence an essential component of the proposed work is to define a series of bindings from our languages to established SPMD libraries and environments. Because our language model is explicitly SPMD, such bindings are a more straightforward proposition than in HPF, where one typically has to pass some extrinsic interface barrier before invoking SPMD-style functions.

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

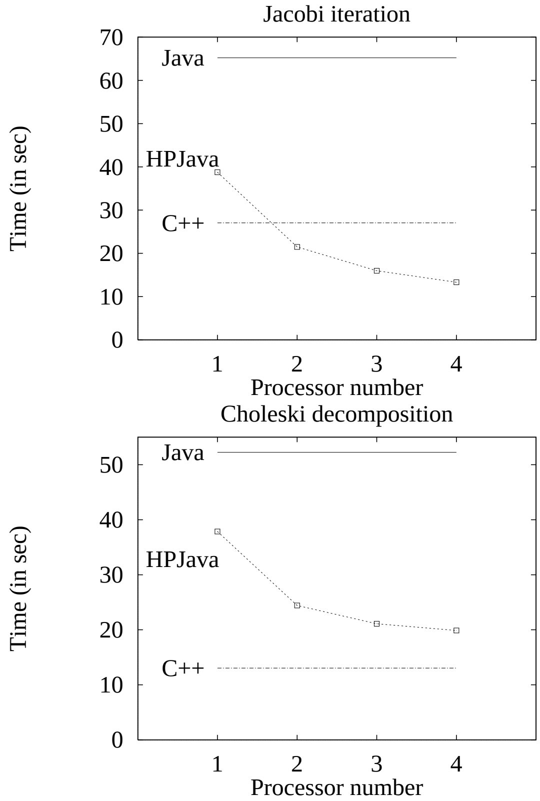

Figure 1: Preliminary HPJava performance

We will group the existing SPMD libraries for data parallel programming into three classes, loosely based on the complexity of design issues involved in integrating them into our language framework.

In the first class we have libraries like ScaLAPACK and PetSc where the primary focus is similar to conventional numerical libraries—providing implementations of standard matrix algorithms, say, but operating on elements in regularly distributed arrays. We anticipate that designing HPspmd interfaces to this kind of package will be relatively straightforward

ScaLAPACK for example, provides linear algebra routines for distributed-memory computers. These routines operate on distributed arrays—in particular, distributed matrices. The distribution formats supported are restricted to two-dimensional block-cyclic distribution for dense matrices and one-dimensional block distribution for narrow-band matrices. Since both these distribution formats are supported by HPspmd (it supports all HPF-compatible distribution formats), using ScaLAPACK routines from the HPspmd framework should present no fundamental difficulties. Problems can only arise if the caller attempts to pass in matrix with a distribution format unsupported by the ScaLAPACK routines. The interface code between HPspmd and ScaLAPACK (which converts between array descriptors) must either flag a runtime error in this case, or remap the argument array (using, for example, the `remap` primitive of Adlib. See *[blah]*).

In the second class we place libraries conceived primarily as underlying support for general parallel programs with regular distributed arrays. They emphasize high-level communication primitives for particular styles of programming, rather than specific numerical algorithms. These libraries include rutimes libraries for HPF-like languages, such as Adlib and Multiblock Parti, and the Global Array toolkit.

Adlib is a runtime library was initially designed to support HPF translation. It provides communication primitives similar to Multiblock PARTI, plus all Fortran 90 transformational intrinsics for arithmetic on distributed arrays. It also provides some gather/scatter operations for irregular access.

The array descriptor of Adlib supports the full HPF 1.0 distributed array model—including all standard distribution formats, all alignment options including replicated alignment, and a facility to map an array to an arbitrary subgroup of the set of active processors. The runtime array descriptor of the HPspmd languages will be an enhanced version of the Adlib descriptor (with a few extra features, such as support for the `GENBLOCK` distribution format of HPF 2.0). The Adlib collective communication library will provide initial library support for regular applications in HPspmd.

The Global Array (GA) toolkit, developed at Pacific Northwest National Lab, provides an efficient and portable "shared-memory" programming interface for distributed-memory computers. Each process in a MIMD parallel program can asynchronously access logical blocks of distributed arrays, without need for explicit cooperation by other processes ("one-sided communication"). This model has been popular and successful. GA is a foundation of the NWChem computational chemistry package.

The existing interface to Global Arrays only supports two-dimensional arrays with general block distribution format. Distributed arrays are created by calls to Fortran functions which return integer handles to an array descriptor. The authors of the package are currently investigating generalization to support multi-dimensional arrays, with more general distribution format. They have already expressed interest in making their library accessible through the kind of language extensions for distributed arrays described in this proposal.

Besides providing a much more tractable interface for creation of multidimensional distributed arrays, our syntax extensions will provide a more convenient interface to primitives such as `ga_get`, which copies a patch of a global array to a local array. Advantages over the existing API include the fact is that the interface can be made uniform for all ranks of arrays, and various sorts of checking can subsumed by the general mechanisms for array section creation, leading to improved safety and compile-time analysis.

Regular problems (such as the linear algebra examples in section **??**) previous section) are an important subset of parallel applications, but of course they are far from exclusive. Many important problems involve data structures too irregular to express purely through HPF-style distributed arrays.

Our third class of libraries therefore includes libraries designed to support irregular problems. These include CHAOS and DAGH.

We anticipate that irregular problems will still benefit from regular data-parallel language extensions (because, at some level they usually resort to representations involving regular arrays). But lower level SPMD programming, facilitated by specialized class libraries, are likely to take a more dominant role when dealing with irregular problems.

The CHAOS/PARTI runtime support library provides primitives for efficiently handling *irregular* problems on distributed memory computers. The complete library includes partitioners to choose optimized mapping on arrays to processors, functions to remap input arrays to meet the optimized partitioning, and functions which optimize interprocessor communications. After data is repartitioned (if necessary) CHAOS programs involve two characteristic phases. The *inspector* phase analyses data access patterns in the main loop, and generates a schedule of optimized optimized communication calls. The *executor* phase involves executing a loop essentially similar to the loop of the original sequential program (interleaved with calls to collective data movement routines which execute the communication schedule). Assuming the same access pattern is repeated a number of times (typical in various applications), the cost of the inspector phase can be amortized over many repeats of the executor phase.

How best to capture this complexity in a convenient HPspmd interface will be a subject of research in the proposed work. A baseline approach (in HPJava, for example) is to handle the translation tables, schedules, etc of CHAOS as ordinary Java objects, constructed and accessed in explicit library calls. Presumbly the initial data for the data and indirection arrays will be provided as normal HPspmd distributed arrays. The simplest assumption is that the CHAOS preprocessing phases yield new arrays: the indirection arrays may well be left as HPspmd distributed arrays, but the data arrays may be reduced to ordinary Java arrays holding local elements (in low-level SPMD style). Then, with no extensions to the currently proposed HPJava language, the parallel loops of the executor phase can be expressed using *overall* constructs. More advanced schemes may incorporate irregular maps into generalized array descriptor *[cite HPF 2, Fortran D, etc]*. This requires further research—it is unproven how important such extensions will be when and distribution of computation are explicit, as in our model. Extensions to the HPspmd language model may be indicated.

DAGH (Distributed Adaptive Grid Hierarchy) was developed at Texas, Austin as a computational toolkit for the Binary Black Hole NSF Grand Challenge Project. It provides the framework to solve systems of partial differential equations using adaptive mesh refinement methods. The computations can be executed sequentially or in parallel according to the specification of the user. In the parallel case DAGH takes over communication, updating ghost regions on the boundaries of component grids.

Conceivably the HPspmd distributed array descriptor could be generalized to directly represent a DAGH grid hierarchy. This is probably unrealistic. DAGH implements a non-trivial storage scheme for its grid hierarchy, based on space-filling curves. It seems unlikely that the details of such a structure can be sensibly handled by a compiler. A more straightforward possibility is to represent the individual grid functions (on the component regular meshes of the hierachy) as essentially standard HPspmd distributed arrays. Since DAGH is supposed to maintain storage for these functions in Fortran-compatible fashion, it should be practical to create an HPspmd array descriptor for them. The hierarchy itself would be represented as a Java object from a library-defined class. This is a crude outline of a particular scenario. Devising practical and convenient HPspmd bindings for DAGH and similar application-oriented libraries is a research topic in the proposed work. It may well drive extensions to the basic language model.

C.2.3 Java MPI linkage

In HPF, with its global-thread-of-control model, a proper interface to the underlying message-passing platform is only practical through the *extrinsic procedure* mechanism. In HPspmd it is possible to access the MPI interface directly. In Fortran and C++ bindings of HPspmd probably the only major issue arising is access to the local elements of distributed arrays as standard sequential Fortran or C++ arrays, which can be passed to the standard MPI functions. Inquiry functions on distributed arrays return the sequential arrays as pointers or handles (depending on the language instantiation).

We have already implemented a Java language binding for MPI, version 1.1. Similar projects on Java MPI bindings are also taking place in elsewhere.

[Mention possibilities for serialization in the wrapper to pass general Java objects? Relation to Getov version (and JMPI)?]

C.2.4 Integration of thread-based single Java VM and multi-VM data parallel

Our language model is primarily aimed at distributed memory computers, including networks of workstations or PCs. Clearly the same model can be implemented efficiently on symmetric multiprocessors—using threads within a single Java virtual machine. The most naive approach is to directly simulate the SPMD model in this environment with a fixed set of threads. More interesting possibilities arise if a few extra restrictions are placed on the language model. The main program can execute as a single thread, with multiple threads forked only when an *overall* construct is encountered. These issues will be investigated further.

C.3 General plan of work

Work at NPAC will initially focus on the Java binding of the HPspmd language model (HPJava). The basic HPJava translator will be available for further development and initial experiments with applications. This version of HPJava will rely heavily on run-time library functions for basic operations such as subscript translation (incorporating only essential optimizations on distributed loops). Initially the only communication library available will be Adlib.

One thread in the proposed work will be to produce an optimized version of the initial HPJava translator. For example, static information will be exploited to inline and simplify calls to the runtime library wherever possible. Runtime checks on multi-dimensional array-bound violations and adherence to the "SPMD constraint" (requiring that accesses be local) will be eliminated where possible. Ultimately it would be desirable to produce a true compiler for HPJava. This will not be a primary goal in the proposed work, which emphasizes rapid implementation of, and experimentation with, novel language ideas, driven by application and library requirements.

A second major thread will be design and limited implementation of HPspmd interfaces to the libraries described in section C.2.2 (and MPI). This work will be coupled with the development of suitable demonstrator applications that exploit the libraries. Initial examples will be taken for the HPFA kernels maintained at NPAC, converted to use the Adlib library. The proposal includes support for application programmers familiar with DAGH and the binary black hole problem, and GA and computational chemistry. Fast Multipole for earthquake problems is another area of current interest at NPAC.

This application work, and in particular the requirements of the library bindings, is expected to drive the third thread: further development of the base HPspmd language model, especially in regard of supporting "irregular" problems.

The fourth major thread will involve taking the HPspmd ideas and embedding them in more conventional scientific programming languages; Fortran and C++. The main design and implementation work here will be carried out by our collaborators from the University of Peking, led by Professor Xiaoming Li. Professor Li has collaborated closely with NPAC over several years, and worked at NPAC for two years during the PCRC project, leading our HPF compiler effort.

C.3.1 Three year workplan

1st year:

- Optimization of baseline HPJava translator.

- Interfacing HPJava to GA.
- Black Hole Application in HPJava (finite difference).
- Study requirements for irregular problems.

2nd year:

- Implementing class libraries for irregular problems.
- Support for adaptive meshes.
- Fast multipoles as a focussed example of a more complex problem.
- Chemistry problems using GA binding.
- Implement extensions to the HPspmd language model, as indicated.

3rd year:

- Integration, as required. Ensure library binding interoperate. Interoperability with other HPspmd languages, developed at Peking during the same time frame.

HPspmd Collaboration Model

- NPAC person to visit Peking University this fall
- Peking Student to visit NPAC early 1999

Some input into this project is expected from work supported by Sun Microsystems, who are providing funding for a project led by NPAC investigating Java for large scale computing. This work will support students at Syracuse, Indiana and Illinois.

C.4 Related work

C.4.1 Applicant's related work

A Java instantiation of the HPspmd model. HPJava is an instance of the HPsmpd language model outlined above. HPJava extends the base Java language by adding predefined classes and some additional syntax for dealing with distributed arrays, and three new control constructs.

As explained in the previous section, the underlying distributed array model is equivalent to the HPF array model. As a matter of detail, distributed array mapping is described in terms of a slightly different set of basic concepts. HPF describes the decomposition of an array through alignment to some *template*, which is in turn distributed over a *processor arrangement*. A processor arrangement is a multidimensional grid of abstract processors. A template is an abstraction of the index space of a distributed array. The analogous concepts in our parametrization of the distributed array are the *distributed range* (or simply *range*) and the *process group* (or simply *group*). A distributed range is akin a single dimension of an HPF template—it defines a map from an integer global subscript range into a particular dimension of a process group. A *process group* is equivalent to an HPF processor arrangement, or to a certain subset of such an arrangement. Switching from templates to ranges and groups is a change of parametrization only. In itself it does not change the set of allowed ways to decompose an array. The new primitives fit better with our distributed control constructs, and correspond more directly to components of our run-time array descriptor. Ranges and groups are treated as proper objects in the extended language. They are values that can be stored in variables or passed to procedures. The group and range list that describe the mapping of a particular distributed array are accessible through inquiry functions on that array.

To motivate the discussion of HPJava, we will refer to figure 2, which gives a parallel implementation of Choleski decomposition in the extended language. In pseudocode, the sequential algorithm is

$$
\begin{array}{l}
For\ k = 1\ to\ n-1 \\
\quad l_{kk} = a_{kk}^{1/2} \\
\quad For\ s = k+1\ to\ n \\
\quad\quad l_{sk} = a_{sk}/l_{kk} \\
\quad For\ j = k+1\ to\ n \\
\quad\quad For\ i = j\ to\ n \\
\quad\quad\quad a_{ij} = a_{ij} - l_{ik}l_{jk} \\
l_{nn} = a_{nn}^{1/2}
\end{array}
$$

The parallel version has been selected to introduce essentially all the new language extensions in HPJava.

In HPJava a base class `Group` describes a general group of processes. It has subclasses `Procs1`, `Procs2`, ..., that represent one-dimensional process grids, two-dimensional process grids, and so on. In the example `p` is defined as a one-dimensional grid of extent `NP`. The *on* construct in the example acts like a conditional, excluding processors outside the group `p`.

A *distributed range* defines a range of integer global subscripts, and specifies how they are mapped into a process grid dimension. In the example, the range `x` is initalized to a cyclically distributed range of extent `N`. `CyclicRange` is one of several subclasses of `Range` that define different *distribution formats*.

Now `a` and `b` are declared to be *distributed arrays*. In HPJava the type-signatures and constructors of distributed arrays use double brackets to distinguish them from ordinary Java arrays. If a particular dimension of an array has a distributed range, the corresponding slot in the type signature of the array should include a `#` symbol. Because `b` has no range distributed over the active process group (`p`) it is defined to be *replicated* across this group. The mapping of `a` and `b` is equivalent to the HPF declarations

```
Procs1 p = new Procs1(NP) ;
on(p) {
  Range x = new CyclicRange(N, p.dim(0));

  float [[,#]] a = new float [[N, x]] ;
  float [[]]   b = new float [[N]] ;  // buffer

  Location l ;
  Index m ;

  for(int k = 0 ; k < N - 1 ; k++) {

    at(l = x [k]) {
      float d =  Math.sqrt(a [k, l]) ;

      a [k, l] = d ;
      for(int s = k + 1 ; s < N ; s++)
        a [s, l] /= d ;
    }

    Adlib.remap(b [[k + 1 : ]], a [[k + 1 : , k]]);

    over(m = x | k + 1 : )
      for(int i = x.idx(m) ; i < N ; i++)
        a [i, m] -= b [i] * b [x.idx(m)] ;
  }

  at(l = x [N - 1])
    a [N - 1, l] = Math.sqrt(a [N - 1, l]) ;
}
```

Figure 2: Choleski decomposition.

```
!HPF$ PROCESSORS p(np)

!HPF$ TEMPLATE t(n)
!HPF$ DISTRIBUTE t(CYCLIC) ONTO p

      REAL a(n, n), b(n)
!HPF$ ALIGN a(i, *) WITH t(i)
!HPF$ ALIGN b(*)    WITH t(*)
```

with range `x` taking over the role of the one-dimensional template `t`.

Subscripting operations on distributed arrays are subject to a strict restriction. An access to an array element such as `a [s, k]` is legal, but *only* if the local process holds the element in question. The language provides syntax to alleviate the inconvenience of this restriction. The idea of a *location* is introduced. It can be viewed as an abstract element, or "slot", of a distributed range. Any location is mapped to a particular slice of a process grid. Locations are used to parametrize a new distributed control construct called the *at* construct. This works like *on*, except that its body is executed only on processes that hold the specified location. Locations can also be used directly as array subscripts, in place on integers (locations used as array subscripts must be elements of the corresponding ranges of the array). The array access above can be safely written in the context

```
Location l = x [k] ;
at(l)
  ... a [s, l] ...
```

(the first dimension of `a` is sequential, so we don't have to worry about the SPMD constraint for subscript `s`). In the main example, this syntax is used to ensure that the first block of code inside the loop only executes on the processor holding column `k`.

The example involves one communication operation. This is taken from the Adlib library: the function `remap` copies the elements of one distributed array or section to another of the same shape. The two arrays can have any, unrelated decompositions. Because `b` has replicated mapping, `remap` copies identical values to all processors—ie it implements a broadcast of the values in the array section `a [[k + 1 :  , k]]`. The syntax for array sections in HPJava is almost identical to the syntex of sections in Fortran 90. Subscript triplets work in the same way as in Fortran.

The last and most important distributed control construct in the language is called *over*. It is used to access all locally held locations in a particular range, and can therefore be used to access all locally held elements of arrays parametrized by that range. The *over* construct implements a distributed parallel loop. Its parameter is a member of the special class `Index` which is a subclass of `Location`. The `idx` member of `Range` can be used inside parallel loops to yield arithmetic expressions that depend on global index values. In the example the *over* construct is used to iterate over all columns of the matrix to the right of column `k`.

As promised, the Choleski example has introduced essentially all the important language ideas in HPJava. Further extensions are minor, or consist in adding new subclasses of `Range` or `Group`, rather than syntax extensions. Figure 3 gives a parallel implementation of red-black relaxation in the same language. To support this important stencil-update paradigm, *ghost regions* are allowed on distributed arrays. In our case the width of these regions is specified in a special form of the `BlockRange` constructor. The ghost regions are explicitly brought up to date using the library function `writeHalo`.

Note that the new range constructor and `writeHalo` function are *library* features (respectively from the base HPJava runtime and the Adlib communication library), *not* new language extensions. One new piece of syntax is involved: the addition and subtraction operators are overloaded so that integer offsets can be added or subtracted to locations, yielding new, shifted, locations. This kind of shifted access only works if the subscripted array has suitable ghost extensions.

Adlib. The Adlib runtime library was initially designed to support HPF translation. It has been used as a foundation of two experimental HPF compilation systems, and is

```
Procs2 p = new Procs2(NP, NP) ;

on(p) {
  Range x = new BlockRange(N, p.dim(0), 1) ;  // ghost width 1
  Range y = new BlockRange(N, p.dim(1), 1) ;  // ghost width 1

  float [[#,#]] u = new float [[x, y]] ;

  int [] widths = {1, 1} ;        // Widths updated by 'writeHalo'

  // ... some code to initialise 'u'

  for(int iter = 0 ; iter < NITER ; iter++) {
    for(int parity = 0 ; parity < 2 ; parity++) {

      Adlib.writeHalo(u, widths) ;

      Index i, j ;
      over(i = x | 1 : N - 2)
        over(j = y | 1 + (x.idx(i) + parity) % 2 : N - 2 : 2)
          u [i, j] = 0.25 * (u [i - 1, j] + u [i + 1, j] +
                             u [i, j - 1] + u [i, j + 1]) ;
    }
  }
}
```

Figure 3: Red-black iteration.

currently being used as a basis of the HPJava translator.

The Adlib kernel is C++ class library, built on MPI. Fortran, C++ and Java interfaces are available or under development. It provides communication primitives similar to Multiblock PARTI, plus the Fortran 90 transformational intrinsics for arithmetic on distributed arrays. It also provides some collective gather/scatter operations for irregular access.

The array descriptor of Adlib supports the full HPF 1.0 distributed array model—including all standard distribution formats, all alignment options including replicated alignment, and a facility to map an array to an arbitrary subgroup of the set of active processors. The runtime array descriptor of the HPspmd languages will be an enhanced version of the Adlib descriptor (with a few extra features, such as support for the `GENBLOCK` distribution format of HPF 2.0). The Adlib collective communication library will provide initial library support for regular applications in HPspmd.

References

E. Biographical sketch

Geoffrey Charles Fox

Address: 111 College Place, NPAC, SU, 13244 gcf@nova.npac.syr.edu , http://www.npac.syr.edu, Phone: (315) 443-2163, Fax: (315) 443-4741
Citizen Status: Permanent Resident Alien; Citizen of United Kingdom

Education: B.A. in Mathematics from Cambridge Univ., Cambridge, England (1961-1964) Ph.D. in Theoretical Physics from Cambridge University (1964-1967) M.A. from Cambridge University (1968)

Professional Experience:

1990- Professor of Computer Science, Syracuse University
1990- Professor of Physics, Syracuse University
1990- Director of Northeast Parallel Architectures Center
1979-1990 Professor of Physics, California Inst. of Tech.
1986-1988 Associate Provost for Computing, California Inst. of Tech.
1983-1985 Dean for Educational Computing, California Inst. of Tech.
1981-1983 Executive Officer of Physics, California Inst. of Tech.
1974-1979 Associate Professor of Physics, California Inst. of Tech.
1971-1974 Assistant Professor of Physics, California Inst. of Tech.
1970-1971 Millikan Research Fellow in Theoretical Physics, Caltech
1970 Visiting Scientist (April-May), Brookhaven National Laboratory
1969-1970 Research Fellow at Peterhouse College, Cavendish Lab.,Cambridge
1968-1969 Research Scientist, Lawrence Berkeley Lab., Berkeley, Calif.
1967-1968 Member of School of Natural Science, Inst. for Advanced Study, Princeton, New Jersey

Awards and Honors: Senior Wrangler, Part III Mathematics, Cambridge (1964) Alfred P. Sloan Foundation Fellowship (1973-75) Fellow of the American Physical Society (1990)

Journal Editorships:

Principal: Concurrency: Practice and Experience (John Wiley, Inc.) Physics and Computers (International Journal of Modern Physics C - World Scientific)

Associate: Journal of Supercomputing,

Selected List of Publications:

[1] Fox, G.C., Johnson, M.A., Lyzenga, G.A., Otto, S.W., Salmon, J.K., Walker, D.W., Solving Problems on Concurrent Processors, Vol. 1, Prentice-Hall, Inc. 1988; Vol. 2, 1990.

[2] Fox, G.C., Copty, N.,Ranka, S.,Shankar, R. "Solving the region growing problem on the Connection Machine," in Proceedings of the 22nd International Conference on Parallel Processing, volume 3, pages 102-105, 1993.

[3] Fox, G. C., Messina, P., Williams, R., Parallel Computing Works!, Morgan Kaufmann, San Mateo Ca, 1994.

[4] Fox G.C., Mills K., "InfoMall: an Innovative strategy for high-performance computing and communications application development", Internet Research, 4:31- 45, 1994.

[5] Fox, G.C., Hiranadani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C.W., Wu, M.Y., "FortranD Language Specifications", Rice COMP TR90079, December 1990, Revised, April 1991.

[6] Fox, G. C. "Approaches to Physical Optimization," in Proceedings of 5th SIAM Conference on Parallel Processes for Scientific Computation, pp 153-162, March 25-27, 1991, Houston, TX, J. Dongarra, K. Kennedy, P. Messina, D. Sorensen, R. Voigt, editors, SIAM, 1992. C3P-959, CRPC-TR91124

[7] Fox G.C., Mansour N., "Parallel Physical Optimization Algorithms for allocating data to multicomputer nodes", Journal of Supercomputing, 8:53-80,1994.

[8] Fox, G. C. "Parallel Computing and Education," Daedalus, Journal of the American Academy of Arts and Sciences, Vol. 121, No. 1, pps 111-118, Winter 1992. C3P-958, CRPC-TR91123.

[9] Fox, G, Bozkus, Z., Choudhary, A., Haupt, T., and Ranka, S. ""A compilation approach for Fortran 90D/HPF compilers on distributed memory MIMD computers," in Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing. Lecture Notes in Computer Science, Springer-Verlag, pp. 200–215. U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua (editors).

[10] Fox, G and Furmanski, W. "Computing on the Web – New Approaches to Parallel Processing – Petaop and Exaop Performance in the Year 2007", Submitted to IEEE Internet Computing, http://www.npac.syr.edu/users/gcf/petastuff/petaweb/

Summary of Interests: Fox leads an significant project to develop prototype high performance Fortran compilers and their runtime support. He is also a leading proponent for the development of computational science as an academic discipline and a scientific method. He has established at Syracuse University both graduate and undergraduate programs which cover both simulation and information technologies. All course have been made available on the Web and his research includes HPCC technology to support education at both K-12 and University level. His research on parallel computing has focused on development and use of this technology to solve large scale computational problems. Fox directs InfoMall, which is focused on accelerating the introduction of high speed communications and parallel computing into New York State industry and developing the corresponding software and systems industry.

Ph.D Advisor: Dr. Richard Eden Cambridge University, England

H. Facilities, equipment, and other resources

Local Computer Facilities

NPAC has a substantial computer infrastructure, including networked workstations, Mac and PC equipment, and various parallel computers. Parallel and distributed platforms include a 12-node IBM SP2, an 8-node DEC Alphafarm interconnected with a Gigaswitch, an 8-node Sun UltraSPARC cluster which will be interconnected with an ATM network shortly. These clusters are shared by all NPAC users, but available in dedicated parallel mode on a reservation basis. An 8-CPU SGI machine with R10000 processors is also available, providing a shared-memory parallel platform for testing purposes.

NPAC has a system support group which handles day to day operation and maintenance of the computer facilities.

Office

NPAC provides adequate office facilities for all participants.

Major equipment

The computer systems described above constitute the major equipment for this effort.

Other resources

NPAC provides basic secretarial services, duplicating, and the like.