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 [19, 30]. 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 [20]. 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. (Partially, no doubt, this reflects a status quo where *expert* programmers build parallel programs and less experienced programmers merely use them.)

SPMD programming has been very successful. There are countless applications written in the most basic SPMD style, using direct message-passing through MPI [21] or similar low-level packages. Many higher-level parallel programming environments and libraries assume the SPMD style as their basic model. Examples include ScaLAPACK [4], PetSc [2], DAGH [36], Kelp [18, 31], the Global Array Toolkit [34] and NWChem [3, 28]. 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/PARTI [16, 37] 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, about as convenient as, say, making a call to an array transformational intrinsic function in Fortran 90. We hope to illustrate that, besides their advantages as a framework for library usage, the resulting programming languages can conveniently express various practical data-parallel algorithms. The resulting framework may also have better prospects for dealing effectively with *irregular* problems than is the case for HPF.

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 [23–25] 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.

The work proposed in this project continues research conducted in the the Parallel Compiler Runtime Consortium (PCRC) project [14]. PCRC was a DARPA-supported project involving Rice, Maryland, Austin, Indiana, CSC, Rochester and Florida, with NPAC as

prime contractor. Achievements included construction of an experimental HPF compilation system [42], delivery of the NPAC PCRC runtime kernel (Adlib) [11] and early work on the design and implementation of HPJava [10].

C.2 Objective and expected significance

Our system aims to 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*.

Primary goals of the current project 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. Supporting the libraries for irregular communication will be an important goal.

3. Testing and evaluating HPJava and the HPspmd model in general on large scale applications.

A major 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.

The next four subsections overview the language extensions we are investigating, the libraries we will study, issues concerning low-level MPI programming in the proposed environment, and the parallel machine model.

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, our language may be slightly harder to learn and use than HPF. But understanding performance of programs should be much easier.

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 (for example).

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 [29]. A further special control construct will facilitate access to all 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 outlined above has been described in [9, 10]. A brief review is given in section C.4.1. HPJava is a superset of the Java language that adds predefined classes and some additional syntax for dealing with distributed arrays. It also adds three new control constructs, including the *overall* distributed loop, which is used to traverse local elements of distributed arrays.

In [7] 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 proposed system would *not*

attempt to exploit parallelism even in constructs such as the array syntax of Fortran 90 or the FORALL statement of Fortran 95, because those constructs operate on the standard sequential arrays of the language. 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 [5, 12]. 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 [14].

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. These compiler optimizations will be important in the base level translator. 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 C.2.2).

The basic HPJava translator will be available by the start date of the proposed work. In figure C.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. 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 arrays are 1024 by 1024. For Jacobi iteration, the timing is for about 90 iterations. Timings are compared with *sequential* Java and C++ versions 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—quite inefficient in Java. The HPJava translation scheme linearizes arrays. (We remark that in recent workshops James Gosling has stated that this is his preferred approach to adding generalized array-like structure in Java.) 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 or Fortran no longer seems an 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.

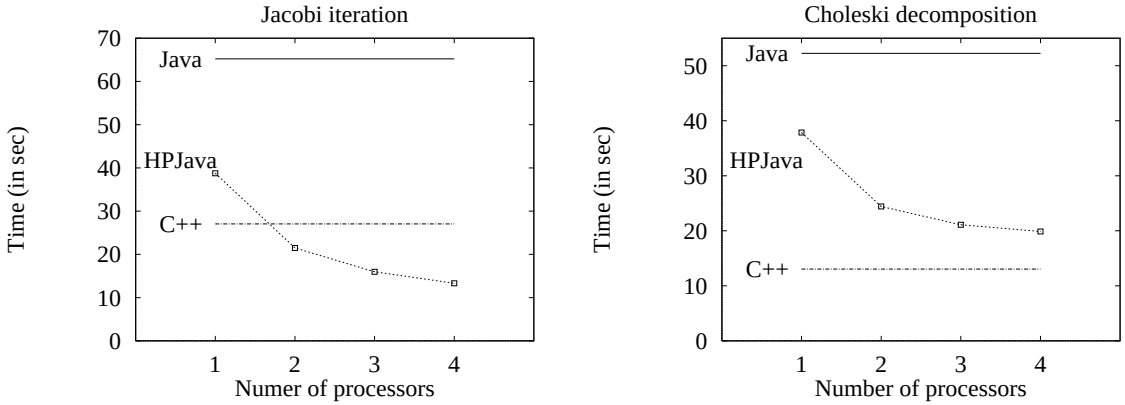

Figure C.1: Preliminary HPJava performance

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 Compiler Runtime Consortium (PCRC) project [14] led by Syracuse (and earlier related work by one of us [12]).

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.

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 [4] and PetSc [2] 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 believe 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 [11]).

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 [1], and the Global Array toolkit [34].

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 [20]). 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 [3, 28] 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 C.4.1) 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 [16, 37] and DAGH [36].

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.

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 values 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 [13, 17, 20]. Extensions to the HPspmd language model may be indicated.

DAGH (Distributed Adaptive Grid Hierarchy) was developed at Texas, Austin as a computational toolkit for several projects including 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.

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 [6, 8]. Our current approach is a relatively direct transcription of standard MPI bindings, but Java object serialization introduces new possibilities for passing compound objects. Similar projects on Java MPI bindings are in progress elsewhere [15, 27].

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 Java version of HPspmd also holds special promise in the domain of metacomputing—targeting heterogeneous systems. At the other extreme, the same model can be straightforwardly implemented 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. Further possibilities arise if a few restrictions on variable usage are added to 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 (rather than source-to-source translator) 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 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 from the HPFA kernels maintained at NPAC [38], converted to use the Adlib library. The proposal includes support for application scientists familiar with DAGH and the binary black hole problem, and GA and computational chemistry. Fast Multipole and its associated irregular MPI-based library 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

Year one: In the first year we will be studying and implementing optimizations in the basic HPJava translator. Our early experiments give us confidence that the basic HPJava translation scheme can give good performance, with minimal overheads, for problems involving large arrays. But there is considerable scope for improving performance on smaller problem sets, especially in reducing run-time overheads associated with subscript conversion. Also in this year an interface will be made between HPJava and the Global Arrays toolkit, at least for some set of platforms. Application efforts will concentrate on a finite difference problem derived from the theory of Black Holes. Study of requirements for irregular problems will be an important activity.

Year two: The requirements identified in the first year's activity will feed into implementation of class library interfaces for irregular problems. This will include CHAOS-like support for irregular access to arrays, and DAGH-like support for adaptive meshes. Fast multipoles will be one focussed example of a more complex problem tackled as an application in this phase. We will look at some representative computation chemistry problems

using the GA binding developed in year one. Any extensions to the basic HPspmd language model indicated by experiences with irregular applications and libraries will be implemented. Work on optimization and compile-time checking of the translator will continue, to produce the robust system needed by the application programmers.

Year three: Emphasis will be on integration. By this stage we will have bindings to several libraries operating on various platforms. We also expect to have compilers for Fortran and C++ versions of the HPspmd languages, developed by our collaborators in China. In so far as practical we must ensure that bindings to different libraries interoperate without interference, and document any problems. The software developed in the project will be placed in the public domain.

C.3.2 Collaborations

As explained above the project involves an important collaboration with Peking University. This will require mutual visits and continuation of ongoing electronic collaboration. NPAC already have substantial sharing of software with the Peking group, exemplified by our HPF front end [32] and the `f2j` Fortran to Java translator [22], where the software was built in China but used in NPAC activities, who provided design expertise.

Some input into this project is expected from work supported by Sun Microsystems. They are providing funding for a project led by NPAC to investigate Java for large scale computing. This work will support students at Syracuse, Indiana and Illinois. It will look at Java for NCSA Alliance Grand Challenges.

C.4 Related work

C.4.1 Applicant's related work

HPJava. HPJava [9, 10] is an instance of the HPsmpd language model. 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*. 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

```
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 C.2: Choleski decomposition.

or passed to procedures. The group and ranges describing a particular distributed array are accessible through inquiry functions.

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

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

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*, base class `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

```
!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 syntax of sections in Fortran 90. Subscript triplets work in the same way as in Fortran 90.

```
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 C.3: Red-black iteration.

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 C.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 [26]. 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. Early development took place in the *shpf* [33] project at Southampton, UK. Subsequently the library was redesigned and reimplemented at Syracuse during in the PCRC project, and delivered as the NPAC PCRC runtime kernel [11]. It has been used as a foundation of two experimental HPF compilation systems [33, 42], (one in Europe and one at Syracuse), and is 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. Benchmarks reported in [42] suggested Adlib provides superior performance to the then-current version of the commercial PGI HPF compiler.

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. The runtime array descriptor of the HPspmd languages will be an enhanced version of the Adlib descriptor. The Adlib collective communication library will provide initial library support for regular applications in HPspmd.

C.4.2 Related languages

F-- [35] is an extended Fortran dialect for SPMD programming. The approach is quite different to the one proposed here. In F--, array subscripting is local by default, or involves a combination of local subscripts and explicit process ids. There is no analogue of global subscripts, or HPF-like distribution formats. In F-- the logical model of communication is built into the language—remote memory access with intrinsics for synchronization—where we follow the philosophy of providing communication through separate libraries. While F-- and our approach share an underlying programming model, we believe that our framework offers greater opportunities for exploiting established library technologies.

Spar [40] is a Java-based language for array-parallel programming. Like our language it introduces multi-dimensional arrays, array sections, and a parallel loop. There are some similarities in syntax, but semantically Spar is very different to HPJava. Spar expresses parallelism but not explicit data placement or communication—in this sense it is a higher level language—closer to HPF.

ZPL [39] is a new programming language for scientific computations. Like Spar, it is an array language. It has an idea of performing computations over a *region*, or set of indices. Within a compound statement prefixed by a *region specifier*, aligned elements of arrays distributed over the same region can be accessed. This idea has certain similarities to our *overall* construct. Communication is more explicit than, say, Spar, but not as explicit as in the language discussed in this article.

Titanium [41] is another Java-based language for high-performance computing. It provides multi-dimensional arrays and a global address space, with an SPMD programming model. It does not provide any special support for distributed arrays, and the programming style is quite different to HPJava.