TOWARDS EFFICIENT COMPILATION OF THE HPJAVA LANGUAGE FOR HIGH PERFORMANCE COMPUTING

Name: Han-Ku Lee
Department: Department of Computer Science
Major Professor: Gordon Erlebacher
Degree: Doctor of Philosophy
Term Degree Awarded: Summer, 2003

This dissertation is concerned with efficient compilation of our Java-based, high-performance, library-oriented, SPMD style, data parallel programming language: *HPJava*.

It starts with some historical review of data-parallel languages such as High Performance Fortran (HPF), message-passing frameworks such as p4, PARMACS, and PVM, as well as the MPI standard, and high-level libraries for multiarrays such as PARTI, the Global Array (GA) Toolkit, and Adlib.

Next, we will introduce our own programming model, which is a flexible hybrid of HPF-like data-parallel language features and the popular, library-oriented, SPMD style. We refer to this model as the *HPspmd programming model*. We will overview the motivation, the language extensions, and an implementation of our HPspmd programming language model, called HPJava. HPJava extends the Java language with some additional syntax and pre-defined classes for handling multiarrays, and a few new control constructs. We discuss the compilation system, including HPspmd classes, type-analysis, pre-translation, and basic translation scheme. In order to improve the performance of the HPJava system, we discuss optimization strategies we will apply such as Partial Redundancy Elimination, Strength Reduction, Dead Code Elimination, and Loop Unrolling. We experiment with and benchmark large scientific and engineering HPJava programs on Linux machine, shared memory machine, and distributed memory machine to prove our compilation and proposed optimization schemes are appropriate for the HPJava system.

Finally, we will compare our HPspmd programming model with modern related languages including Co-Array Fortran, ZPL, JavaParty, Timber, and Titanium.

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

TOWARDS EFFICIENT COMPILATION OF THE HPJAVA LANGUAGE FOR HIGH PERFORMANCE COMPUTING

By

HAN-KU LEE

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Summer Semester, 2003

The members of the Committee approve the dissertation of Han-Ku Lee defended on June 12, 2003.

Gordon Erlebacher
Professor Directing Thesis

Larry Dennis
Outside Committee Member

Geoffrey C. Fox
Committee Member

Michael Mascagni
Committee Member

Daniel Schwartz
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

To my mother, father, big sisters, and all my friends ...
Especially I thank Dr. Fox, Dr. Carpenter, Dr. Erlebacher, and all my committee members for every concern.

ACKNOWLEDGEMENTS

Development of the HPJava environment was funded in part by National Science Foundation Division of Advanced Computational Infrastructure and Research, contract number 9872125 (any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This dissertation is concerned with efficient compilation of our Java-based, high-performance, library-oriented, SPMD style, data parallel programming language: *HPJava*.

It starts with some historical review of data-parallel languages such as High Performance Fortran (HPF), message-passing frameworks such as p4, PARMACS, and PVM, as well as the MPI standard, and high-level libraries for multiarrays such as PARTI, the Global Array (GA) Toolkit, and Adlib.

Next, we will introduce our own programming model, which is a flexible hybrid of HPF-like data-parallel language features and the popular, library-oriented, SPMD style. We refer to this model as the *HPspmd programming model*. We will overview the motivation, the language extensions, and an implementation of our HPspmd programming language model, called HPJava. HPJava extends the Java language with some additional syntax and pre-defined classes for handling multiarrays, and a few new control constructs. We discuss the compilation system, including HPspmd classes, type-analysis, pre-translation, and basic translation scheme. In order to improve the performance of the HPJava system, we discuss optimization strategies we will apply such as Partial Redundancy Elimination, Strength Reduction, Dead Code Elimination, and Loop Unrolling. We experiment with and benchmark large scientific and engineering HPJava programs on Linux machine, shared memory machine, and distributed memory machine to prove our compilation and proposed optimization schemes are appropriate for the HPJava system.

Finally, we will compare our HPspmd programming model with modern related languages including Co-Array Fortran, ZPL, JavaParty, Timber, and Titanium.

CHAPTER 1

INTRODUCTION

1.1 Research Objectives

Historically, data parallel programming and data parallel languages have played a major role in high-performance computing. By now we know many of the implementation issues, but we remain uncertain what the high-level programming environment should be. Ten years ago the High Performance Fortran Forum published the first standardized definition of a language for data parallel programming. Since then, substantial progress has been achieved in HPF compilers, and the language definition has been extended because of the requests of compiler-writers and the end-users. However, most parallel application developers today don't use HPF. This does *not* necessarily mean that most parallel application developers prefer the lower-level SPMD programming style. More likely, we suggest, it is either a problem with implementations of HPF compilers, or rigidity of the HPF programming model.

Research on compilation technique for HPF continues to the present time, but efficient compilation of the language seems still to be an incompletely solved problem. Meanwhile, many successful applications of parallel computing have been achieved by hand-coding parallel algorithms using low-level approaches based on the libraries like Parallel Virtual Machine (PVM), Message-Passing Interface (MPI), and Scalapack.

Based on the observations that compilation of HPF is such difficult problem while library based approaches more easily attain acceptable levels of efficiency, a hybrid approach called *HPspmd* has been proposed[8]. Our HPspmd Model combines data parallel and low-level SPMD paradigms. HPspmd provides a language framework to facilitate direct calls to established libraries for parallel programming with distributed data. A preprocessor supports special syntax to represent distributed arrays, and language extensions to support common operations such as distributed parallel loops. Unlike HPF, but similar to MPI, communication is always explicit. Thus, irregular patterns of access to array elements can be handled by making explicit bindings to libraries.

The major goal of the system we are building is to provide a programming model which is a flexible hybrid of HPF-like data-parallel language and the popular, library-oriented, SPMD style. We refer to this model as the *HPspmd programming model*. It incorporates syntax for representing multiarrays, for expressing that some computations are localized to some processors, and for writing a distributed form of the parallel loop. Crucially, it also supports binding from the extended languages to various communication and arithmetic libraries. These might involve simply new interfaces to some subset of PARTI, Global Arrays, Adlib, MPI, and so on. Providing libraries for *irregular* communication may well be important. Evaluating the HPspmd programming model on large scale applications is also an important issue.

What would be a good candidate for the base-language of our HPspmd programming model? We need a base-language that is popular, has the simplicity of Fortran, enables modern object-oriented programming and high performance computing with SPMD libraries, and so on. Java is an attractive base-language for our HPspmd model for several reasons: it has clean and simple object semantics, cross-platform portability, security, and an increasingly large pool of adept programmers. Our research aim is to power up Java for the data parallel SPMD environment. We are taking on a more modest approach than a full scale data parallel compiler like HPF. We believe this is a proper approach to Java where the situation is rapidly changing and one needs to be flexible.

1.2 Organization of This Dissertation

We will survey the history and recent developments in the field of programming languages and environments for data parallel programming. In particular, we describe ongoing work at Florida State University and Indiana University, which is developing a compilation system for a data parallel extension of the Java programming language.

The first part of this dissertation is a brief review of the evolution of data parallel languages starting with dialects of Fortran that developed through the 1970s and 1980s. This line of development culminated in the establishment of the High Performance Fortran

(HPF) in the 1990s. We will review essential features of the HPF language and issues related to its compilation.

The second part of the dissertation is about the HPspmd programming model, including the motivation, HPspmd language extensions, integration of high-level libraries, and an example of the HPspmd model in Java—HPJava.

The third part of the dissertation covers background and current status of the HPJava compilation system. We will describe the multiarray types and HPspmd classes, basic translation scheme, and optimization strategies.

The fourth part of the dissertation is concerned with experimenting and benchmarking some scientific and engineering applications in HPJava, to prove the compilation and optimization strategies we will apply to the current system are appropriate.

The fifth part of the dissertation compares and contrasts with modern related languages including Co-Array Fortran, ZPL, JavaParty, Timber, and Titanium. Co-Array Fortran is an extended Fortran dialect for SPMD programming; ZPL is an array-parallel programming language for scientific computations; JavaParty is an environment for cluster-based parallel programming in Java; Timber is a Java-based language for array-parallel programming. Titanium is a language and system for high-performance parallel computing, and its base language is Java; We will describe the systems and discuss the relative advantages of our proposed system.

Finally, the dissertation gives the conclusion and contribution of our HPspmd programming model and its Java extension, HPJava, for modern and future computing environments. Future work on the HPspmd programming model, such as programming support for high-performance grid-enabled applications and the Java Numeric efforts in Java Grande, may contribute to a merging of the high-performance computing and the Grid computing envionments of the future.

CHAPTER 2

BACKGROUND

2.1 Historical Review of Data Parallel Languages

Many computer–related scientists and professionals will probably agree with the idea that, in a short time—less than decade—every PC will have multi-processors rather than uni-processors[1]. This implies that parallel computing plays a critically important role not only in scientific computing but also the modern computer technology.

There are lots of places where parallel computing can be successfully applied—supercomputer simulations in government labs, scaling Google's core technology powered by the world's largest commercial Linux cluster (more than 10,000 servers), dedicated clusters of commodity computers in a Pixar RenderFarm for animations [43, 39]. Or by collecting cycles of many available processors over Internet. Some of the applications involve large-scale "task farming," which is applicable when the task can be completely divided into a large number of independent computational parts. The other popular form of massive parallelism is "data parallelism." The term data parallelism is applied to the situation where a task involves some large data-structures, typically arrays, that are split across nodes. Each node performs similar computations on a different part of the data structure. For data parallel computation to work best, it is very important that the volume of communicated values should be small compared with the volume of locally computed results. Nearly all successful applications of massive parallelism can be classified as either task farming or data parallel.

For task farming, the level of parallelism is usually coarse-grained. This sort of parallel programming is naturally implementable in the framework of conventional sequential programming languages: a function or a method can be an abstract task, a library

[1]This argument was made in a Panel Session during the 14th annual workshop on Languages and Compilers for Parallel Computing LCPC 2001).

can provide an abstract, problem-independent infrastructure for communication and load-balancing. For data parallelism, we meet a slightly different situation. While it is possible to code data parallel programs for modern parallel computers in ordinary sequential languages supported by communication libraries, there is a long and successful history of special languages for data parallel computing. Why is data parallel programming special?

Historically, a motivation for the development of data parallel languages is strongly related with Single Instruction Multiple Data (SIMD) computer architectures. In a SIMD computer, a single control unit dispatches instructions to large number of compute nodes, and each node executes the same instruction on its own local data.

The early data parallel languages developed for machines such as the Illiac IV and the ICL DAP were very suitable for efficient programming by scientific programmers who would not use a parallel assembly language. They introduced a new programming language concept—*distributed* or *parallel arrays*—with different operations from those allowed on sequential arrays.

In the 1980s and 1990s microprocessors rapidly became more powerful, more available, and cheaper. Building SIMD computers with specialized computing nodes gradually became less economical than using general purpose microprocessors at every node. Eventually SIMD computers were replaced almost completely by Multiple Instruction Multiple Data (MIMD) parallel computer architectures. In MIMD computers, the processors are autonomous: each processor is a full-fledged CPU with both a control unit and an ALU. Thus each processor is capable of executing its own program at its own pace at the same time: asynchronously.

The asynchronous operations make MIMD computers extremely flexible. For instance, they are well-suited for the task farming parallelism, which is hardly practical on SIMD computers at all. But this asynchrony makes general programming of MIMD computers hard. In SIMD computer programming, synchronization is not an issue since every aggregate step is synchronized by the function of the control unit. In contrast, MIMD computer programming requires concurrent programming expertise and hard work. A major issue on MIMD computer programming is the explicit use of synchronization primitives to control nondeterministic behavior. Nondeterminism appears almost inevitably in a system that has multiple independent threads of the control—for example, through race conditions. Careless synchronization leads to a new problem, called deadlock.

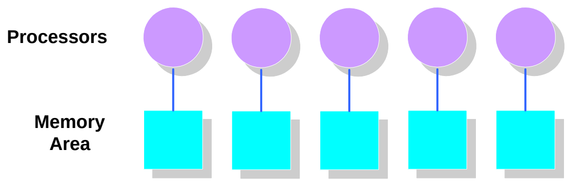

Figure 2.1. Idealized picture of a distributed memory parallel computer

Soon a general approach was proposed in order to write data parallel programs for MIMD computers, having some similarities to programming SIMD computers. It is known as Single Program Multiple Data (SPMD) programming or the Loosely Synchronous Model. In SPMD, each node executes essentially the same thing at essentially the same time, but without any single central control unit like a SIMD computer. Each node has its own local copy of the control variables, and updates them in an identical way across MIMD nodes. Moreover, each node generally communicates with others in well-defined collective phases. These data communications implicitly or explicitly[2] synchronize the nodes. This aggregate synchronization is easier to deal with than the complex synchronization problems of general concurrent programming. It was natural to think that catching the SPMD model for programming MIMD computers in data parallel languages should be feasible and perhaps not too difficult, like the successful SIMD languages. Many distinct research prototype languages experimented with this, with some success. The High Performance Fortran (HPF) standard was finally born in the 90s.

2.1.1 Multi-processing and Distributed Data

Figure 2.1 represents many successful parallel computers obtainable at the present time, which have a large number of autonomous processors, each possessing its own local memory area. While a processor can very rapidly access values in its own local memory, it must access values in other processors' memory either by special machine instructions or message-passing software. But the current technology can't make remote memory accesses nearly as fast as local memory accesses. Who then should get rid of this burden to minimize the number of

[2]Depending on whether the platform presents a message-passing model or a shared memory model.

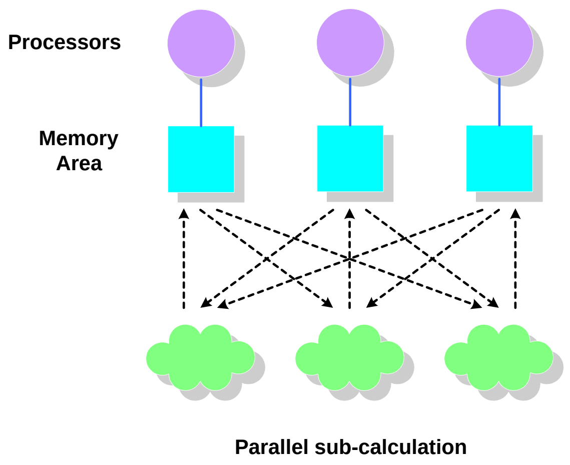

Figure 2.2. A data distribution leading to excessive communication

accesses to non-local memory? The programmer or compiler? This is the *communication* problem.

A next problem comes from ensuring each processor has a fair share of the whole work load. We absolutely lose the advantage of parallelism if one processor, by itself, finishes almost all the work. This is the *load-balancing* problem.

High Performance Fortran (HPF) for example allows the programmer to add various *directives* to a program in order to explicitly specify the distribution of program data among the memory areas associated with a set of (physical or virtual) processors. The directives don't allow the programmer to directly specify which processor will perform a specific computation. The compiler must decide where to do computations. We will see in chapter 3 that HPJava takes a slightly different approach but still requires programmers to distribute data.

We can think of some particular situation where all operands of a specific sub-computation, such as an assignment, reside on the same processor. Then, the compiler can allocate that part of the computation to the processor having the operands, and no remote memory access will be needed. So an onus on parallel programmers is to distribute data across the processors in the following ways:

- to minimize remote memory accesses, operands allocated to the same processor should involve as many sub-computations as possible, on the other hand,

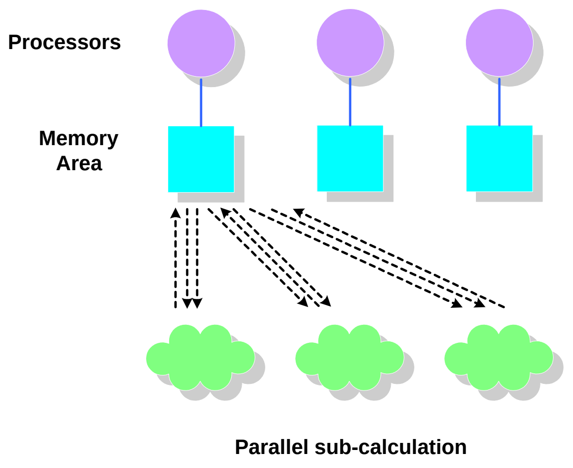

Figure 2.3. A data distribution leading to poor load balancing

- to maximize parallelism, the group of *distinct* sub-computations which can execute in parallel at any time should involve data on as many different processors as possible.

Sometimes these are contradictory goals. Highly parallel programs can be difficult to code and distribute efficiently. But, equally often it is possible to meet the goals successfully.

Suppose that we have an ideal situation where a program contains p sub-computations which can be executed in parallel. They might be p basic assignments consisting of an array assignment or `FORALL` construct. Assume that each expression which is computed combines two operands. Including the variable being assigned, a single sub-computation therefore has three operands. Moreover, assume that there are p processors available. Generally, the number of processors and the number of sub-computations are probably different, but this is a simplified situation.

Figure 2.2 depicts that all operands of each sub-computation are allocated in different memory areas. Wherever the computation is executed, each assignment needs at least two communications.

Figure 2.3 depicts that no communication is necessary, since all operands of all assignments reside on a single processor. In this situation, the computation might be executed where the data resides. In this case, though, no effective parallelism occurs. Alternatively, the compiler might decide to share the tasks out anyway. But then, all operands of tasks on processors other than the first would have to be communicated.

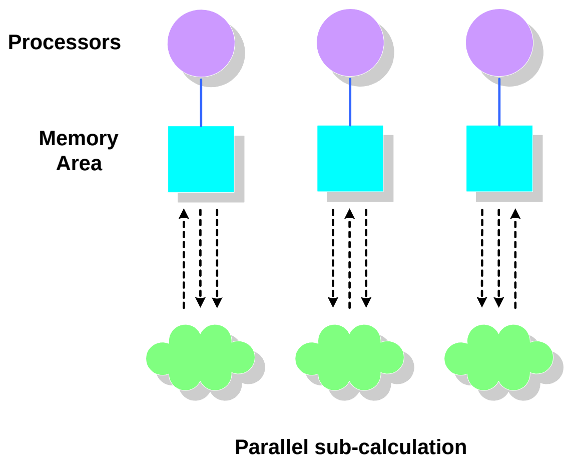

Figure 2.4. An ideal data distribution

Figure 2.4 depicts that all operands of an individual assignment occur on the same processor, but each group of the sub-computations is uniformly well-distributed over processors. In this case, we can depend upon the compiler to allocate each computation to the processor holding the operands, requiring no communication, and perfect distribution of the work load.

Except in the most simple programs, it is impossible to choose a distribution of program variables over processors like Figure 2.4. The most important thing in distributed memory programming is to distribute the *most critical regions* of the program over processors to make them as much like Figure 2.4 as possible.

2.2 High Performance Fortran

In 1993 the *High Performance Fortran Forum*, a league of many leading industrial and academic groups in the field of parallel processing, established an informal language standard, called *High Performance Fortran* (HPF) [22, 28]. HPF is an extension of Fortan 90 to support the data parallel programming model on *distributed memory* parallel computers. The standard of HPF was supported by many manufactures of parallel hardware, such as Cray, DEC, Fujitsu, HP, IBM, Intel, Maspar, Meiko, nCube, Sun, and Thinking Machines. Several companies broadcast their goal to develop HPF implementations, including ACE, APR, KAI, Lahey, NA Software, Portland, and PSR.

Since then, except a few vendors like Portland, many of those still in business have given up their HPF projects. The goals of HPF were immensely aspiring, and perhaps attempted to put too many ideas into one language. Nevertheless, we think that many of the originally defined goals for HPF standardization of a distributed data model for SPMD computing are important.

2.2.1 The Processor Arrangement and Templates

The programmer often wishes to explicitly distribute program variables over the memory areas with respect to a set of processors. Then, it is desirable for a program to have some representations of the set of processors. This is done by the `PROCESSORS` directive.

The syntax of a `PROCESSOR` definition looks like the syntax of an array definition of Fortran:

```
!HPF$ PROCESSORS P (10)
```

This is a declaration of a set of 10 abstract processors, named `P`. Sets of abstract processors, or *processor arrangements*, can be multi-dimensional. For instance,

```
!HPF$ PROCESSORS Q (4, 4)
```

declares 16 abstract processors in a 4×4 array.

The programmer may directly distribute data arrays over processors. But it is often more satisfactory to go through the intermediate of an explicit *template*. Templates are different from processor arrangements. The collection of abstract processors in an HPF processor arrangement may not correspond identically to the collection of physical processors, but we implicitly assume that abstract processors are used at a similar level of granularity as the physical processors. It would be unusual for shapes of the abstract processor arrangements to correspond to those of the data arrays in the parallel algorithms. With the template concept, on the other hand, we can capture the fine-grained grid of the data array.

```
!HPF$ TEMPLATE T (50, 50, 50)
```

declares a $50 \times 50 \times 50$ three-dimensional template, called T.

Next we need to talk about how the elements of the template are distributed among the elements of the processor arrangement. A directive, `DISTRIBUTE`, does this task. Suppose that we have

```
!HPF$ PROCESSORS P1 (4)
!HPF$ TEMPLATE T (17)
```

There are various schemes by which T may be distributed over P1. A distribution directive specifies how template elements are mapped to processors.

Block distributions are represented by

```
!HPF$ DISTRIBUTE T1 (BLOCK) ONTO P1
!HPF$ DISTRIBUTE T1 (BLOCK (6)) ONTO P1
```

In this situation, each processor takes a contiguous block of template elements. All processors take the identically sized block, if the number of processors evenly divides the number of template elements. Otherwise, the template elements are evenly divided over most of the processors, with last processor(s) holding fewer. In a modified version of block distribution, we can explicitly specify the specific number of template elements allocated to each processor.

Cyclic distributions are represented by

```
!HPF$ DISTRIBUTE T1 (CYCLIC) ONTO P1
!HPF$ DISTRIBUTE T1 (CYCLIC (6)) ONTO P1
```

In the basic situation, the first template element is allocated on the first processor, and the second template element on the second processor, etc. When the processors are used up, the next template element is allocated from the first processor in wrap-around fashion. In a modified version of cyclic distribution, called block-cyclic distribution, the index range is first divided evenly into contiguous blocks of specified size, and these blocks are distributed cyclically.

In the multidimensional case, each dimension of the template can be independently distributed, mixing any of the four distribution patterns above. In the example:

```
!HPF$ PROCESSOR P2 (4, 3)
!HPF$ TEMPLATE T2 (17, 20)
!HPF$ DISTRIBUTE T2 (CYCLIC, BLOCK) ONTO P2
```

the first dimension of T2 is cyclically distributed over the first dimension of P2, and the second dimension of of T2 is distributed blockwise over the second dimension of P2.

Another important feature is that some dimensions of a template might have *collapsed mapping*, allowing a template to be distributed onto a processor arrangement with fewer dimensions than template:

```
!HPF$ DISTRIBUTE T2 (BLOCK, *) ONTO P1
```

represents that the first dimension of T2 will be block-distributed over P1. But, for a fixed value of the index T2, all values of the second subscript are mapped to the same processor.

2.2.2 Data Alignment

The directive, ALIGN aligns arrays to the templates. We consider an example. The core code of an LU decomposition subroutine looks as follows;

```
01       REAL A (N, N)
02       INTEGER N, R, R1
03       REAL, DIMENSION (N) :: L_COL, U_ROW
04
05       DO R = 1, N - 1
06         R1 = R + 1
07         L_COL (R  : ) = A (R : , R)
08         A (R , R1 : ) = A (R, R1 : ) / L_COL (R)
09         U_ROW (R1 : ) = A (R, R1 : )
10         FORALL (I = R1 : N, J = R1 : N)
11      &    A (I, J) = A (I, J) - L_COL (I) * U_ROW (J)
12       ENDDO
```

After looking through the above algorithm, we can choose a template,

```
!HPF$ TEMPLATE T (N, N)
```

The major data structure of the problem, the array A that holds the matrix, is identically matched with this template. In order to align A to T we need an ALIGN directive like;

```
!HPF$ ALIGN A(I, J) WITH T (I, J)
```

Here, integer subscripts of "alignee"—the array which is to be aligned—are called *alignment dummies.* In this manner, every element of the alignee is mapped to some element of the template.

Most of the work in each iteration of the DO-loop from our example is in the following statement, which is line 11 of the program,

```
A (I, J) = A (I, J) - L_COL (I) * U_ROW
```

With careful inspection of the above assignment statement, we see we can avoid the communications if copies of L_COL (I) and U_ROW (J) are allocated wherever A (I, J) is allocated. The following statement can manage it using a *replicated* alignment to the template T,

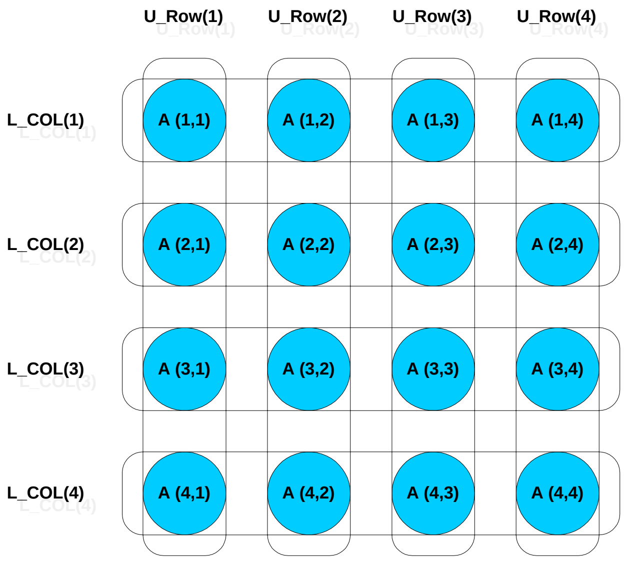

Figure 2.5. Alignment of the three arrays in the LU decomposition example

```
!HPF$ ALIGN  L_COL (I) WITH T (I, *)
!HPF$ ALIGN  U_ROW (J) WITH T (*, I)
```

where an asterisk means that array elements are replicated in the corresponding processor dimension, i.e. a copy of these elements is shared across processors. Figure 2.5 shows the alignment of the three arrays and the template. Thus, no communications are needed for the assignment in the `FORALL` construct since all operands of each elemental assignment will be allocated on the same processor. Do the other statements require some communications?

The line 8 is equivalent to

```
FORALL (J = R1 : N)  A (R, J) = A (R, J) / L_COL (R)
```

Since we know that a copy of `L_COL (R)` will be available on any processor wherever `A (R, J)` is allocated, it requires no communications.

But, the other two array assignment statements *do* need communications. For instance, the assignment to `L_COL`, which is the line 7 of the program, is equivalent to

```
FORALL (I = R : N)  L_COL (I) = A (I, R)
```

Since `L_COL (I)` is replicated in the `J` direction, while A (I, R) is allocated only on the processor which holds the template element where $J = R$, updating the `L_COL` element is to broadcast the A element to all concerned parties. These communications will be properly inserted by the compiler.

The next step is to distribute the template (we already aligned the arrays to a template). A BLOCK distribution is not good choice for this algorithm since successive iterations work on a shrinking area of the template. Thus, a block distribution will make some processors idle in later iterations. A CYCLIC distribution will accomplish better load balancing

In the above example, we illustrated simple alignment—"identity mapping" array to template—and also replicated alignments. What would general alignments look like?

One example is that we can transpose an array to a template.

```
      DIMENSION B(N, N)
!HPF$ ALIGN B(I, J) WITH T(J, I)
```

transpositionally maps B to T (B (1, 2) is aligned to T (2, 1), and so on). More generally, a subscript of an *align target* (i.e. the template) can be a linear expression in *one* of the alignment dummies. For example,

```
      DIMENSION C(N / 2, N / 2)
!HPF$ ALIGN C(I, J) WITH T(N / 2 + I, 2 * J)
```

The rank of the alignee and the align-target don't need to be identical. An alignee can have a "collapsed" dimension, an align-target can have "constant" subscript (e.g. a scalar might be aligned to the first element of a one-dimensional template), or an alignee can be "replicated" over some dimensions of the template:

```
      DIMENSION D(N, N, N)
!HPF$ ALIGN D(I, J, K) WITH T(I, J)
```

is an example of a collapsed dimension. The element of the template, T, is *not* dependent on K. For fixed I and J, each element of the array, D, is mapped to the same template element.

In this section, we have covered HPF's processor arrangement, distributed arrays, and data alignment which we will basically adopt to the HPspmd programming model we present in chapter 4.

2.3 Message-Passing for HPC

2.3.1 Overview and Goals

The message passing paradigm is a generally applicable and efficient programming model for distributed memory parallel computers, that has been widely used for the last decade and an half. Message passing is a different approach from HPF. Rather than designing a new parallel language and its compiler, message passing library routines explicitly let processes communicate through messages on some classes of parallel machines, especially those with distributed memory.

Since there were many message-passing vendors who had their own implementations, a message-passing standard was needed. In 1993, the *Message Passing Interface* Forum established a standard API for message passing library routines. Researchers attempted to take the most useful features of several implementations, rather than singling out one existing implementation as a standard. The main inspirations of MPI were from PVM [17], Zipcode [44], Express [14], p4 [7], PARMACS [41], and systems sold by IBM, Intel, Meiko Scientific, Cray Research, and nCube.

The major advantages of making a widely-used message passing standard are *portability* and *scalability*. In a distributed memory communication environment where the higher level of routines and/or abstractions build on the lower level message passing routines, the benefits of the standard are obvious. The message passing standard lets vendors make efficient message passing implementations, accommodating hardware support of scalability for their platform.

2.3.2 Early Message-Passing Frameworks

p4 [7, 5] is a library of macros and subroutines elaborated at Argonne National Laboratory for implementing parallel computing on diverse parallel machines. The predecessor of p4 (*Portable Programs for Parallel Processors*) was the m4-based "Argonne macros" system, from which it took its name. The p4 system is suitable to both shared-memory computers using monitors and distributed-memory parallel computers using message-passing. For the shared-memory machines, p4 supports a collection of primitives as well as a collection of monitors. For the distributed-memory machines, p4 supports send, receive, and process creation libraries.

p4 is still used in MPICH [21] for its network implementation. This version of p4 uses Unix sockets in order to execute the actual communication. This strategy allows it to run on a diverse machines.

PARMACS [41] is tightly associated with the p4 system. PARMACS is a collection of macro extensions to the p4 system. In the first place, it was developed to make Fortran interfaces to p4. It evolved into an enhanced package which supported various high level global operations. The macros of PARMACS were generally used in order to configure a set of p4 processes. For example, the macro `torus` created a configuration file, used by p4, to generate a 3-dimensional graph of a torus. PARMACS influenced the topology features of MPI.

PVM (Parallel Virtual Machine) [17] was produced as a byproduct of an ongoing heterogeneous network computing research project at Oak Ridge National Laboratory in the summer of 1989. The goal of PVM was to probe heterogeneous network computing—one of the first integrated collections of software tools and libraries to enable machines with varied architecture and different floating-point representation to be viewed as a single parallel virtual machine. Using PVM, one could make a set of heterogeneous computers work together for concurrent and parallel computations.

PVM supports various levels of heterogeneity. At the application level, tasks can be executed on best-suited architecture for their result. At the machine level, machines with different data formats are supported. Also, varied serial, vector, and parallel architectures are supported. At the network level, a Parallel Virtual Machine consists of various network types. Thus, PVM enables different serial, parallel, and vector machines to be viewed as one large distributed memory parallel computer.

There was a distinct parallel processing system, called *Express* [14]. The main idea of Express was to start with a sequential version of a program and to follow Express recommended procedures for making an optimized parallel version. The core of Express is a collection of communication, I/O, and parallel graphic libraries. The characteristics of communication primitives were very similar with those of other systems we have seen. It included various global operations and data distribution primitives as well.

2.3.3 MPI: A Message-Passing Interface Standard

The main goal of the MPI standard is to define a common standard for writing message passing applications. The standard ought to be practical, portable, efficient, and flexible for message passing. The following is the complete list of goals of MPI [18], stated by the MPI Forum.

- Design an application programming interface (not necessarily for compilers or a system implementation library).
- Allow efficient communication: avoid memory-to-memory copying and allow overlap of computation and communication, and offload to communication co-processor, where available.
- Allow for implementations that can be used in a heterogeneous environment.
- Allow convenient C and Fortran 77 bindings for the interface.
- Assume a reliable communication interface: the user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.
- Define an interface that is not too different from current practice, such as PVM, NX, Express, p4, etc., and provide extensions that allow greater flexibility.
- Define an interface that can be implemented on many vendor's platforms, with no significant changes in the underlying communication and system software.
- Semantics of the interface should be language independent.
- The interface should be designed to allow for thread-safety.

The standard covers point-to-point communications, collective operations, process groups, communication contexts, process topologies, bindings for Fortran 77 and C, environmental management and inquiry, and a profiling interface.

The main functionality of MPI, the point-to-point and collective communication of MPI are generally executed within process groups. A *group* is an ordered set of processes, each process in the group is assigned a unique rank such that 0, 1, ..., $p-1$, where p is the number of processes. A *context* is a system-defined object that uniquely identifies a communicator. A message sent in one context can't be received in other contexts. Thus, the communication context is the fundamental methodology for isolating messages in distinct libraries and the user program from one another.

The process group is high-level, that is, it is visible to users in MPI. But, the communication context is low-level—not visible. MPI puts the concepts of the process group and communication context together into a communicator. A *communicator* is a data object that specializes the scope of a communication. MPI supports an initial communicator, MPI_COMM_WORLD which is predefined and consists of all the processes running when program execution begins.

Point-to-point communication is the basic concept of MPI standard and fundamental for send and receive operations for typed data with associated message tag. Using the point-to point communication, messages can be passed to another process with explicit message tag and implicit communication context. Each process can carry out its own code in MIMD style, sequential or multi-threaded. MPI is made *thread-safe* by not using global state[3].

MPI supports the blocking *send* and *receive* primitives. Blocking means that the sender buffer can be reused right after the send primitive returns, and the receiver buffer holds the complete message after the receive primitive returns. MPI has one blocking receive primitive, MPI_RECV, but four blocking send primitives associated with four different communication modes: MPI_SEND (Standard mode), MPI_SSEND (Synchronous mode), MPI_RSEND (Ready mode), and MPI_BSEND (Buffered mode).

Moreover, MPI supports non-blocking send and receive primitives including MPI_ISEND and MPI_IRECV, where the message buffer can't be used until the communication has been completed by a wait operation. A call to a non-blocking send and receive simply posts the communication operation, then it is up to the user program to explicitly complete the communication at some later point in the program. Thus, any non-blocking operation needs a minimum of two function calls: one call to start the operation and another to complete the operation.

A *collective communication* is a communication pattern that involves all the processes in a communicator. Consequently, a collective communication is usually associated with more than two processes. A collective function works as if it involves group synchronization. MPI supports the following collective communication functions: MPI_BCAST, MPI_GATHER, MPI_SCATTER, MPI_ALLGATHER, and MPI_ALLTOALL.

Figure 2.6 on page 19 illustrates the above collective communication functions with 6 processes.

[3]The MPI specification is thread safe in the sense that it does not *require* global state. Particular MPI implementations may or may not be thread safe. MPICH is not.

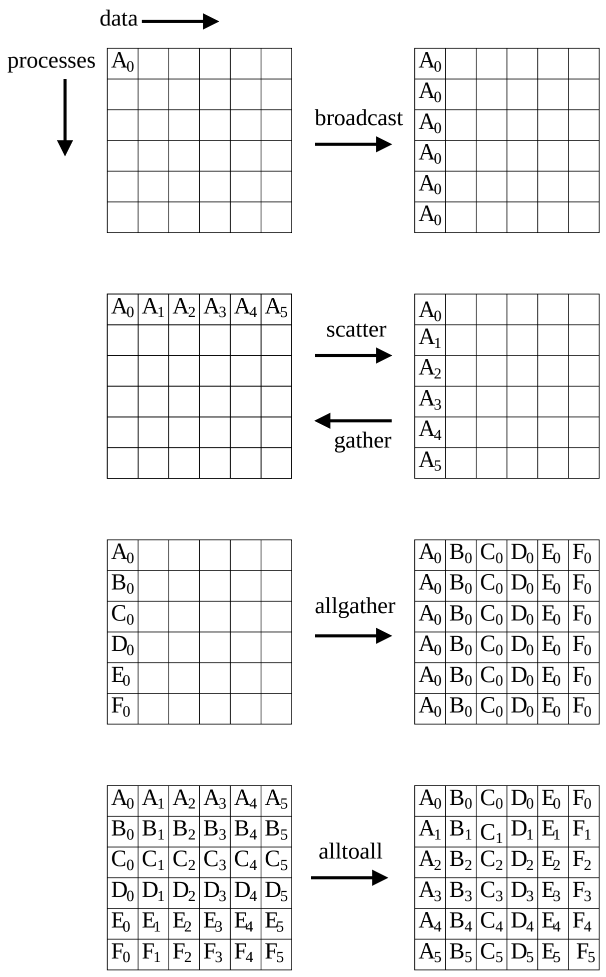

Figure 2.6. Collective communications with 6 processes.

A *user-defined data type* is taken as an argument of all MPI communication functions. The data type can be, in the simple case, a primitive type like integer or floating-point number. As well as the primitive types, a user-defined data type can be the argument, which makes MPI communication powerful. Using user-defined data types, MPI provides for the communication of complicated data structures like array sections.

2.4 High Level Libraries for Distributed Arrays

A *distributed array* is a *collective array* shared by a number of processes. Like an ordinary array, a distributed array has some index space and stores a collection of elements of fixed

```
DO I = 1, N
  X(IA(I)) = X(IA(I)) + Y(IB(I))
ENDDO
```

Figure 2.7. Simple sequential irregular loop.

type. Unlike an ordinary array, the index space and associated elements are scattered across the processes that share the array.

The communication patterns implied by data-parallel languages like HPF can be complex. Moreover, it may be hard for a compiler to create all the low-level MPI calls to carry out these communications. Instead, the compiler may be implemented to exploit higher-level libraries which directly manipulate distributed array data. Like the run-time libraries used for memory management in sequential programming languages, the library that a data parallel compiler uses for managing distributed arrays is often called its *run-time library.*

2.4.1 PARTI

The PARTI [16] series of libraries was developed at University of Maryland. PARTI was originally designed for irregular scientific computations.

In irregular problems (e.g. PDEs on unstructured meshes, sparse matrix algorithms, etc) a compiler can't anticipate data access patterns until run-time, since the patterns may depend on the input data, or be multiply indirected in the program. The data access time can be decreased by pre-computing which data elements will be sent and received. PARTI transforms the original sequential loop to two constructs, called the *inspector* and *executor* loops.

First, the inspector loop analyzes the data references and calculates what data needs to be fetched and where they should be stored. Second, the executor loop executes the actual computation using the information generated by the inspector.

Figure 2.7 is a basic sequential loop with irregular accesses. A parallel version from [16] is illustrated in Figure 2.8.

The first call, `LOCALIZE`, from the parallel version corresponds to `X(IA(I))` terms from the sequential loop. It translates the `I_BLK_COUNT` global subscripts in `IA` to local subscripts, which are returned in the array `LOCAL_IA`. Also, it builds up a *communication schedule*, which

```
C Create required schedules (Inspector):
   CALL LOCALIZE(DAD_X, SCHEDULE_IA, IA, LOCAL_IA, I_BLK_COUNT,
                 OFF_PROC_X)
   CALL LOCALIZE(DAD_Y, SCHEDULE_IB, IB, LOCAL_IB, I_BLK_COUNT,
                 OFF_PROC_Y)

C Actual computation (Executor):
   CALL GATHER(Y(Y_BLK_SIZE + 1), Y, SCHEDULE_IB)
   CALL ZERO_OUT_BUFFER(X(X_BLK_SIZE + 1), OFF_PROC_X)
   DO L = 1, I_BLK_COUNT
     X(LOCAL_IA(I)) = X(LOCAL_IA(I)) + Y(LOCAL_IA(I))
   ENDDO
   CALL SCATTER_ADD(X(X_BLK_SIZE + 1), X, SCHEDULE_IA)
```

Figure 2.8. PARTI code for simple parallel irregular loop.

is returned in SCHEDULE_IA. Setting up the communication schedule involves resolving the requests of accesses, sending lists of accessed elements to the owner processors, detecting proper accumulation and redundancy eliminations, etc. The result is some list of messages that contains the local sources and destinations of the data. Another input argument of LOCALIZE is the descriptor of the data array, DAD_X. The second call works in the similar way with respect to Y(IA(I)).

We have seen the inspector phrase for the loop. The next is the executor phrase where actual computations and communications of data elements occurs.

A collective call, GATHER fetches necessary data elements from Y into the target ghost regions which begins at Y(Y_BLK_SIZE + 1). The argument, SCHEDULE_IB, includes the communication schedule. The next call ZERO_OUT_BUFFER make the value of all elements of the ghost region of X zero.

In the main loop the results for locally owned X(IA) elements are aggregated directly to the local segment X. Moreover, the results from non-locally owned elements are aggregated to the ghost region of X.

The final call, SCATTER_ADD, sends the values in the ghost region of X to the related owners where the values are added in to the physical region of the segement.

We have seen the inspector-executor model of PARTI. An important lesson from the model is that construction of communication schedules must be isolated from execution of those schedules. The immediate benefit of this separation arises in the common situation where the form of the inner loop is constant over many iterations of some outer loop. The same communication schedule can be reused many times. The inspector phase can be moved out of the main loop. This pattern is supported by the Adlib library used in HPJava.

2.4.2 The Global Array Toolkit

The main issue of the Global Array [35] (GA) programming model is to support a portable interface that allows each process independent, asynchronous, and efficient access to blocks of physically distributed arrays without explicit cooperation with other processes. In this respect, it has some characteristics of shared-memory programming. However, GA encourages data locality since it takes more work by the programmer to access remote data than local data. Thus, GA has some characteristic of message-passing as well.

The GA Toolkit supports some primitive operations that are invoked collectively by all processes: create an array controlling alignment and distribution, destroy an array, and synchronize all processes.

In addition, the GA Toolkit provides other primitive operations that are invoked in MIMD style: fetch, store, atomic accumulate into rectangular patches of a two-dimensional array; gather and scatter; atomic read and increment array element; inquire the location and distribution of data; directly access local elements of array to provide and improve performance of application-specific data parallel operations.

2.4.3 NPAC PCRC Runtime Kernel – Adlib

NPAC PCRC Runtime Kernel (Adlib) [11] was originally implemented as a high-level runtime library, designed to support translation of data-parallel languages. An early version of Adlib was implemented in 1994 in the *shpf* project at Southampton University. A much improved—virtually rewritten—version was completed in the Parallel Compiler Runtime Consortium (PCRC) project at Northeast Parallel Architecture Center (NPAC) at Syracuse University. Adlib was initially invented for HPF. It was completely reimplemented for HPJava.

Adlib supports a built-in representation of a distributed array, and a library of communication and arithmetic operations acting on these arrays. The array model supports HPF-like distribution formats, and arbitrary sections, and is more general than GA. The Adlib communication library concentrates on collective communication rather than one-sided communication. It also provides some collective gather/scatter operations for irregular access.

A *Distributed Array Descriptor* (DAD) played an very important role in earlier implementation of the Adlib library. A DAD object describes the layout of elements of an array distributed across processors. The original Adlib kernel was a C++ library, built on MPI. The array descriptor is implemented as an object of type `DAD`. The public interface of the `DAD` had three fields. An integer value, `rank` is the dimensionality,r, of the array, greater than or equal to zero. A process group object, `group`, defines a multidimensional process grid embedded in the set of processes executing the program—the group over which the array is distributed. A vector, `maps`, has r map objects, one for each dimension of the array. Each map object was made up of an integer local memory stride and a *range object*.

There are several interfaces to the kernel Adlib. The shpf Fortran interface is a Fortran 90 interface using a kind of dope vector. The PCRC Fortran interface is a Fortran 77 interface using handles to DAD objects. The *ad++* interface is a high-level C++ interface supporting distributed arrays as template container classes. An early HPJava system used a Java Native Interface (JNI) wrapper to the original C++ version of Adlib.

2.5 Discussion

In this chapter we have given a historical review of data parallel languages, High Performance Fortran (HPF), message-passing systems for HPC, and high-level libraries for distributed arrays.

Data parallelism means a task with large data structures can be split across nodes. Historically, data parallel programming approach has gradually evolved from SIMD (Single Instruction Multiple Data) via MIMD (Multiple Instruction Multiple Data) to SPMD (Single Program Multiple Data). There are two approaches to data parallelism: languages and communication libraries.

HPF is an extension of Fortran 90 to support the data parallel programming model on distributed memory parallel computers. Despite of the limited success of HPF, many of the originally defined goals for HPF standardization of a distributed data model for SPMD programming are important and affect contemporary data parallel languages.

Message-passing is a different approach from HPF. It explicitly lets processes communicate through messages on classes of parallel machines with distributed memory. The Message Passing Interface (MPI) established a standard API for message-passing library routines, inspired by PVM, Zipcode, Express, p4, PARMACS, etc. The major advantages making MPI successful are its portability and scalability.

High-level libraries for distributed arrays, such as PARTI, The Global Array Toolkit, and NPAC PCRC Runtime Kernel – Adlib, are designed and implemented in order to avoid low-level, and complicated communication patterns for distributed arrays.

HPF, message-passing, and high-level libraries for distributed arrays have influenced the HPspmd programming model we will introduce in the next chapter. Our model takes features from HPF, is programmable in message passing paradigm, and is closely involved with high-level libraries and distributed arrays.

CHAPTER 3

THE HPSPMD PROGRAMMING MODEL

The major goal of the system we are building is to provide a programming model that is a flexible hybrid of HPF-like data-parallel language features and the popular, library-oriented, SPMD style, omitting some basic assumptions of the HPF model. We refer to this model as the *HPspmd programming model.*

The HPspmd programming model adds a small set of syntax extensions to a base language, which in principle might be (say) Java, Fortran, or C++. The syntax extensions add distributed arrays as language primitives, and a few new control constructs such as `on`, `overall`, and `at`. Moreover, it is necessary to provide bindings from the extended language to various communication and parallel computation libraries.

3.1 Motivations

The SPMD programming style has been popular and successful, and many parallel applications have been written in the most basic SPMD style with direct low-level message-passing such as MPI [18]. Moreover, many high-level parallel programming environments and libraries incorporating the idea of the distributed array, such as the Global Array Toolkit [35], can claim that SPMD programming style is their standard model, and clearly work. Despite the successes, the programming approach lacks the uniformity and elegance of HPF—there is no unifying framework. Compared with HPF, allocating distributed arrays and accessing their local and remote elements are clumsy. Also, the compiler does not provide the safety of compile-time checking.

The HPspmd programming model described here brings some ideas, and various run-time and compile-time techniques, from HPF. But it gives up some basic principles, such as the single, logical, global thread of control, the compiler-determined placement of computations, and the compiler-decided automatic insertion of communications.

Our model is an *explicitly SPMD programming model* supplemented by syntax—for representing multiarrays, for expressing that some computations are localized to some processors, and for constructing a distributed form of the parallel loop. The claim is that these features make calls to various data-parallel libraries (e.g. high level libraries for communications) as convenient as making calls to array intrinsic functions in Fortran 90. We hope that our language model can efficiently handle various practical data-parallel algorithms, besides being a framework for library usage.

3.2 HPspmd Language Extensions

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

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

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

3.3 Integration of High-Level Libraries

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

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

There are libraries such as ScaLAPACK [4] and PetSc [3] which are similar to standard numerical libraries. For example, they support implementations of standard matrix algorithms, but are executed on elements in regularly distributed arrays. We suppose that designing HPspmd interfaces for the libraries will be relatively straightforward. ScaLAPACK, for instance, supports linear algebra library routines for distributed-memory computers. These library routines work on distributed arrays (and matrices, especially). Thus, it should not be difficult to use ScaLAPACK library routines from HPspmd frameworks.

There are also libraries primarily supporting general parallel programming with regular distributed arrays. They concentrate on high-level communication primitives rather than specific numerical algorithms. We reviewed detailed examples in the section 2.4 of the previous chapter.

Finally, we mention libraries supporting irregular problems. Regular problems are important in parallel applications. But, they are only a subset of the parallel applications. There are a lot of important and critical problems involving data structures which are too irregular to illustrate using HPF-style distributed arrays. This category includes PARTI [16] and DAGH [37]. We believe that irregular problems will still take advantage of regular data-parallel language extensions. At some level, irregular problems usually somehow depend on representations including regular arrays. But, the lower-level SPMD programming, using specialized class libraries, probably plays a more important role here.

3.4 The HPJava Language

HPJava [10, 9] is an implementation of our HPspmd programming model. It extends the Java language with some additional syntax and with some pre-defined classes for handling multiarrays. The multiarray model is adopted from the HPF array model. But, the programming model is quite different from HPF. It is one of explicitly cooperating processes. All processes carry out the same program, but the components of data structures are divided across processes. Each process operates on locally held segment of an entire multiarray.

3.4.1 Multiarrays

As mentioned before, Java is an attractive programming language for various reasons. But it needs to be improved for solving large computational tasks. One of the critical Java and JVM issues is the lack of *true multidimensional arrays*[1] like those in Fortran, which are arguably the most important data structures for scientific and engineering computing.

Java does not directly provide arrays with rank greater than 1. Instead, Java represents multidimensional arrays as "array of arrays." The disadvantages of Java multidimensional arrays result from the time consumption for out-of-bounds checking, the ability to alias rows of an array, and the cost of accessing an element. In contrast, accessing an element in Fortran 90 is straightforward and efficient. Moreover, an *array section* can be implemented efficiently.

HPJava supports a true multidimensional array, called a *multiarray*, which is a modest extension to the standard Java language. The new arrays allow regular section subscripting, similar to Fortran 90 arrays. The syntax for the HPJava multiarray is a subset of the syntax introduced later for *distributed* arrays.

The type signature and constructor of the multiarray has double brackets to tell them from ordinary Java arrays.

```
int [[*,*]] a = new int [[5, 5]] ;
double [[*,*,*]] b = new double [[10, n, 20]] ;
int [[*]] c = new int [[100]] ;

int [] d = new int [100] ;
```

[1]J.E. Moreira proposed *Java Specification Request* for a multiarray package, because, for scientific and engineering computations, Java needs true multidimensional arrays like those in HPJava

a, b, c are respectively a 2-dimensional integer array, 3-dimensional double array, and 1-dimensional int array. The rank is determined by the number of asterisks in the type signature. The shape of a multiarray is rectangular. Moreover, c is similar in structure to the standard array d. But, c and d are not identical. For instance, c allows section subscripting, but d does not. The value d would not be assignable to c, or vice versa.

Access to individual elements of a multiarray is represented by a subscripting operation involving single brackets, for example,

```
for(int i = 0 ; i < 4 ; i++)
  a [i, i + 1] = i + c [i] ;
```

In the current sequential context, apart from the fact that a single pair of brackets might include several comma-separated subscripts, this kind of subscripting works just like ordinary Java array subscripting. Subscripts always start at zero, in the ordinary Java or C style (there is no Fortran-like lower bound).

Our HPJava imports a Fortran-90-like idea of array *regular sections*. The syntax for *section subscripting* is different to the syntax for local subscripting. Double brackets are used. These brackets can include scalar subscripts or *subscript triplets*. A section is an object in its own right—its type is that of a suitable multi-dimensional array. It describes some subset of the elements of the parent array. For example, in

```
int [[]] e = a [[2, :]] ;
foo(b [[ : , 0, 1 : 10 : 2]]) ;
```

e becomes an alias for the 3rd row of elements of a. The procedure foo should expect a two-dimensional array as argument. It can read or write to the set of elements of b selected by the section. The shorthand : selects the whole of the index range, as in Fortran 90.

3.4.2 Processes

An HPJava program is started concurrently in all members of some process collection[2]. From HPJava's point of view, the processes are arranged by special objects representing *process groups*. In general the processes in an HPJava group are arranged in multidimensional grids.

[2]Most time, we will talk about *processes* instead of *processors*. The different processes may be executed on different processors to achieve a parallel speedup.

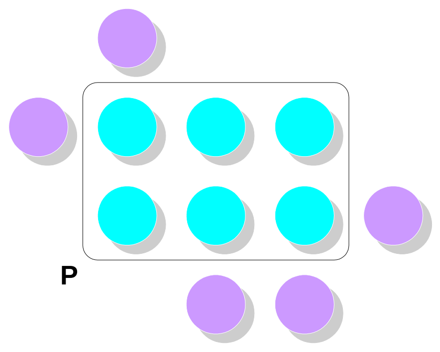

Figure 3.1. The process grids illustrated by `p`

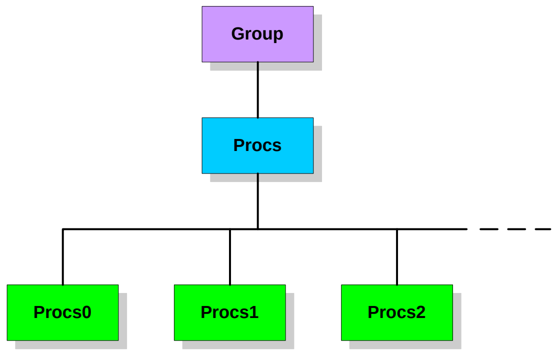

Figure 3.2. The `Group` hierarchy of HPJava.

Suppose that a program is running concurrently on 6 or more processes. Then, one defines a 2×3 process grid by:

```
Procs2 p = new Procs2(2, 3) ;
```

`Procs2` is a class describing 2-dimensional grids of processes. Figure 3.1 illustrates the grid `p`, which assumes that the program was executing in 11 processes. The call to the `Procs2` constructor selected 6 of these available processes and incorporated them into a grid.

`Procs2` is a subclass of the special base class `Group`. The `Group` class has a special status in the HPJava language. Figure 3.2 illustrates the full `Group` hierarchy. `Procs0` is a zero-dimensional process grid, which can only have one process. `Procs1` is one-dimensional process grid. Generally, `ProcsN` is an N-dimensional process grid.

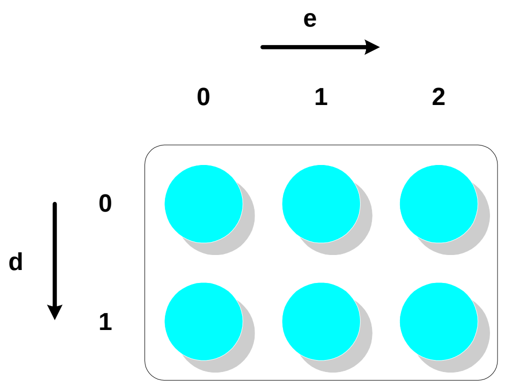

Figure 3.3. The process dimension and coordinates in p.

After creating `p`, we will want to run some code within the process group. The `on` construct limits control to processes in its parameter group. The code in the `on` construct is *only* executed by processes that belong to `p`. The `on` construct fixes `p` as the *active process group* within its body.

The `Dimension` class represents a specific dimension or axis of a particular process grid. Called *process dimensions*, these are available through the inquiry method `dim(r)`, where r is in the range $0, ..., R-1$ (R is the rank of the grid). `Dimension` has a method `crd()`, which returns the local *process coordinate* associated with the dimension. Thus, the following HPJava program crudely illustrates the process dimension and coordinates in `p` from Figure 3.3.

```
Procs2 p = new Procs2(2, 3) ;
on (p) {
  Dimension d = p.dim(0), e = p.dim(1) ;
  System.out.println("(" + d.crd() + ", " + e.crd() + ")") ;
}
```

3.4.3 Distributed Arrays

One of the most important feature HPJava adds to Java is the *distributed array.* A distributed array is a generalized kind of multiarray, a collective array shared by a number of processes. Like an ordinary multiarray, a distributed array has a rectangular,

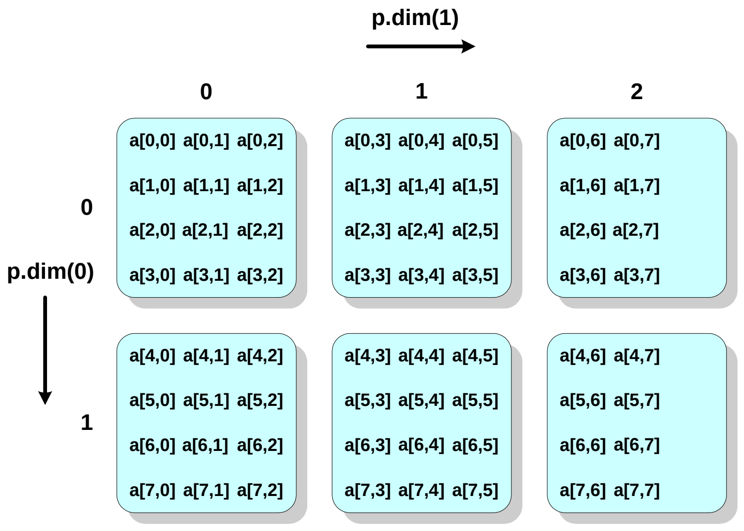

Figure 3.4. A two-dimensional array distributed over `p`.

multidimensional index space, and stores a collection of elements of fixed type. Unlike an ordinary array, associates elements are distributed over the processes that share the array.

The type signatures and constructors of distributed arrays are similar to those of sequential multiarrays (see the section 3.4.1). But the distribution of the index space is parameterized by a special class, `Range` or its subclasses. In the following HPJava program we create a two-dimensional, `N`×`N`, array of floating points distributed over the grid `p`. The distribution of the array, `a`, for `N` = 8 is illustrated in Figure 3.4.

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

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

  ...
}
```

The subclass `BlockRange` for index range of the array allows the index space of each array dimension to be divided into consecutive blocks. Ranges of a single array should be distributed over distinct dimensions of the same process grid.

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

  double [[-,-]] a = new double [[x, y]] on p ;
  double [[-,-]] b = new double [[x, y]] on p ;
  double [[-,-]] c = new double [[x, y]] on p ;

  ... initialize values in ‘a’, ‘b’

  overall(i = x for :)
    overall(j = y for :)
      c [i, j] = a [i, j] + b [i, j] ;
}
```

Figure 3.5. A parallel matrix addition.

The type signature of a multiarray is generally represented as

$$T \ \ \texttt{[[}attr_0, \ldots, attr_{R-1}\texttt{]]} \ \ bras$$

where R is the rank of the array and each term $attr_r$ is either a single hyphen, `-`, for a distributed dimension or a single asterisk, `*`, for a sequential dimension, the term *bras* is a string of zero or more bracket pairs, `[]`. In principle, T can be *any* Java type other than an array type. This signature represents the type of a multiarray whose elements have Java type

$$T \ \ bras$$

If ‘bras’ is non-empty, the elements of the multiarray are (standard Java) arrays.

3.4.4 Parallel Programming and Locations

We have seen so far how to create a distributed array. It is time to discuss how to use a distributed array for parallel programming in HPJava through a simple example. Figure 3.5 is a basic HPJava program for a parallel matrix addition.

The `overall` construct is another control construct of HPJava. It represents a distributed parallel loop, sharing some characteristics of the *forall* construct of HPF. The *index triplet*[3] of the `overall` construct represents a lower bound, an upper bound, and a step, all of which are integer expressions, as follows:

```
overall(i = x for l : u : s)
```

The step in the triplet may be omitted as follow:

```
overall(i = x for l : u)
```

where the default step is 1. The default of the lower bound, `l`, is 0 and the default of the upper bound, `u`, is $N-1$, where N is the extent of the range before the `for` keyword. Thus,

```
overall(i = x for :)
```

means "repeat for all elements, `i`, in the range `x`" from figure 3.5.

An HPJava range object can be thought of as a collection of abstract *locations*. From Figure 3.5, the two lines

```
Range x = new BlockRange(N, p.dim(0)) ;
Range y = new BlockRange(N, p.dim(1)) ;
```

mean that each of the range `x` and `y` has `N` locations. The symbol `i` scoped by the `overall` construct is called a *distributed index*. Its value is a *location*, rather an abstract element of a distributed range than an integer value. It is important that with a few exceptions we will mention later, the subscript of a multiarray must be a distributed index, and the location should be an element of the range associated with the array dimension. This restriction is a important feature, ensuring that referenced array elements are locally held.

Figure 3.6 is an attempt to visualize the mapping of locations from `x` and `y`. We will write the locations of `x` as `x[0]`, `x[1]`, ..., `x[N -1]`. Each location is mapped to a particular group of processes. Location `x[1]` is mapped to the three processes with coordinates $(0, 0)$, $(0, 1)$ and $(0, 2)$. Location `y[4]` is mapped to the two processes with coordinates $(0, 1)$ and $(1, 1)$.

[3]The syntax for triplets is directly from Fortran 90.

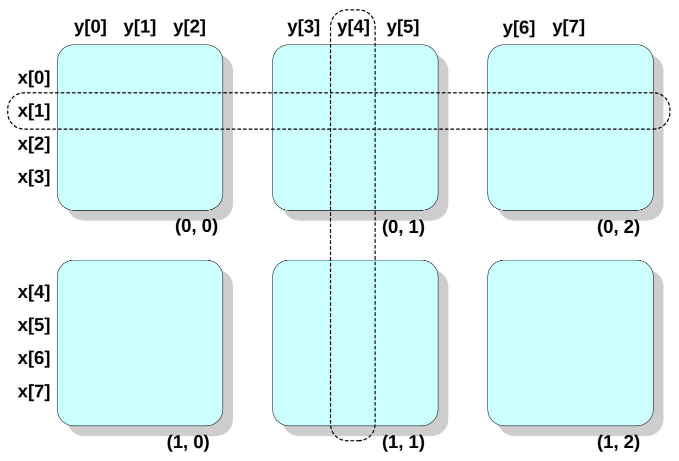

Figure 3.6. Mapping of `x` and `y` locations to the grid `p`.

In addition to the `overall` construct, there is an `at` construct in HPJava that defines a distributed index when we want to update or access a single element of a multiarray rather than accessing a whole set of elements in parallel. When we want to update `a[1, 4]`, we can write:

```
double [[-,-]] a = new double [[x, y]] ;
// a [1, 4] = 19 ;  <--- Not allowed since 1 and 4 are not distributed
//                       indices, therefore, not legal subscripts.
...
at(i = x[1])
  at(j = y[4])
    a [i, j] = 19 ;
```

The semantics of the `at` construct is similar to that of the `on` construct. It limits control to processes in the collection that hold the given location. Referring again to Figure 3.6, the outer

```
  at(i = x [1])
```

construct limits execution of its body to processes with coordinates $(0, 0)$, $(0, 1)$ and $(0, 2)$. The inner

```
  at(j = y [4])
```

then restricts execution down to just process $(0, 1)$. This is the process that owns element `a[1,4]`. The restriction that subscripts must be distributed indices helps ensure that processes only manipulate array elements stored locally. If a process has to access elements non-locally held, some explicit library call should be used to fetch them.

We often need to get the integer global index associated with a distributed index. But, using a distributed index as an integer value is not allowed in HPJava. To get the integer *global index* value for the distributed loop, we use the backquote symbol as a postfix operator on a distributed index, for example, i‘, read “i-primed”. The following nested `overall` loop is an example using an integer *global index* value:

```
overall(i = x for :)
  overall(j = y for :)
    a[i, j] = i‘ + j‘ ;
```

Figure 3.7 is a complete example of Laplace equation by Jacobi relaxation that uses HPJava language features introduced in this section.

3.4.5 More Distribution Formats

HPJava provides further distribution formats for dimensions of its multiarrays without further extensions to the syntax. Instead, the `Range` class hierarchy is extended. The full HPJava `Range` hierarchy is illustrated in Figure 3.8.

In the previous section, we have already seen the `BlockRange` distribution format and objects of class `Dimension`. The `Dimension` class is familiar, although is was not presented previously as a range class. `CyclicRange` corresponds to the cyclic distribution format available in HPF.

The `CyclicRange` distributions can result in better *load balancing* than the simple `BlockRange` distributions. For example, dense matrix algorithms don’t have the locality that favors the block distributions used in stencil updates, but do involve phases where parallel computations are restricted to subsections of the whole array. In block distributions format, these sections may be corresponding to only a fraction of the available processes, leading to a very poor distribution workload. Here is a artificial example:

```
Procs2 p = new Procs2(P, P) ;
on(p) {
  Range x = new BlockRange(N, p.dim(0)) ;
  Range y = new BlockRange(N, p.dim(1)) ;

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

  // Initialize ‘a’: set boundary values.
  overall(i = x for :)
    overall(j = y for :)
      if(i‘ == 0 || i‘ == N - 1 || j‘ == 0 || j‘ == N - 1)
        a [i, j] = i‘ * i‘ - j‘ * j‘;
      else
        a [i, j] = 0.0;

  // Main loop.
  double [[-,-]] n = new double [[x, y]] on p ;
  double [[-,-]] s = new double [[x, y]] on p ;
  double [[-,-]] e = new double [[x, y]] on p ;
  double [[-,-]] w = new double [[x, y]] on p ;
  double [[-,-]] r = new double [[x, y]] on p ;

  do {
    Adlib.shift(n, a,  1, 0) ;
    Adlib.shift(s, a, -1, 0) ;
    Adlib.shift(e, a,  1, 1) ;
    Adlib.shift(w, a, -1, 1) ;

    overall(i = x for 1 : N - 2 )
      overall(j = y for 1 : N - 2) {
        double newA = (double)0.25 * (n [i, j] + s [i, j] +
                                      e [i, j] + w [i, j]) ;
        r [i, j] = Math.abs(newA - a [i, j]) ;
        a [i, j] = newA ;
      }
  } while(Adlib.maxval(r) > EPS);

  // Output results
  ...
}
```

Figure 3.7. Solution for Laplace equation by Jacobi relaxation in HPJava.

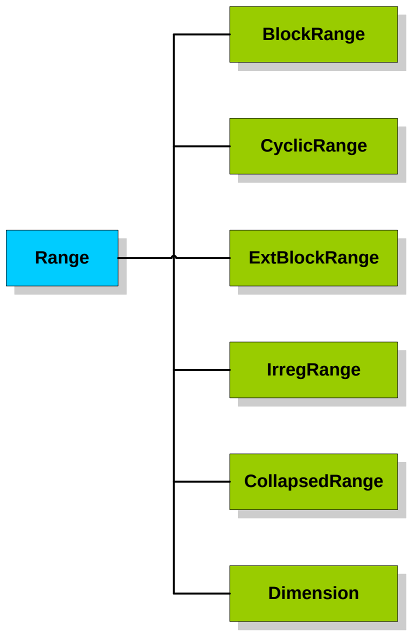

Figure 3.8. The HPJava Range hierarchy.

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

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

  overall(i = x for 0 : N / 2 - 1)
    overall(j = y for 0 : N / 2 - 1)
      a [i, j] = complicatedFunction(i`, j`) ;
}
```

The point is that the `overall` constructs only traverse half the ranges of the array they process. As shown in Figure 3.9, this leads to a very poor distribution of workload. The process with coordinates $(0, 0)$ does nearly all the work. The process at $(0, 1)$ has a few elements to work on, and all the other processes are idle.

In cyclic distribution format, the index space is mapped to the process dimension in round-robin fashion. By only changing the range constructor from `BlockRange` to `CyclicRange`, we can make better distribution workload. Figure 3.10 shows that the imbalance is not nearly as extreme. Notice that nothing changed in the program apart

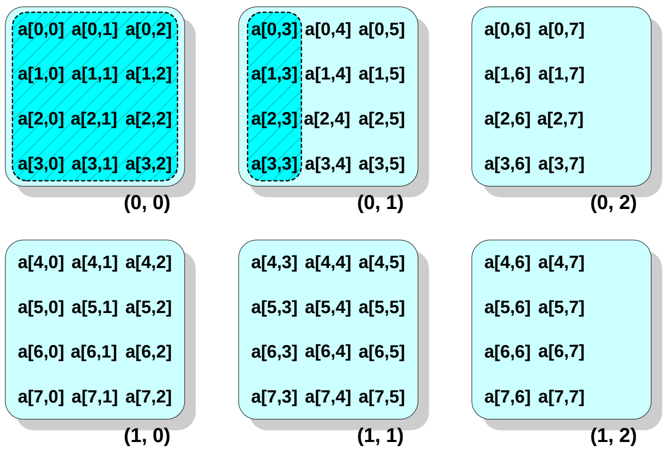

Figure 3.9. Work distribution for block distribution.

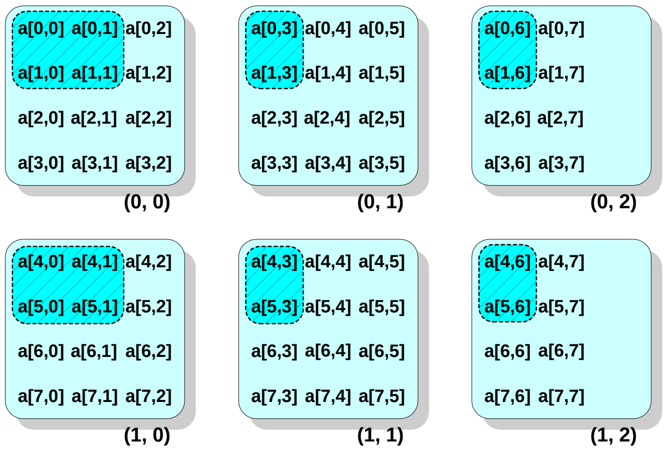

Figure 3.10. Work distribution for cyclic distribution.

from the choice of range constructors. This is an attractive feature that HPJava shares with HPF. If HPJava programs are written in a sufficiently pure data parallel style (using `overall` loops and collective array operations) it is often possible to change the distribution format of arrays dramatically while leaving much of the code that processes them unchanged.

The `ExtBlockRange` distribution describes a `BlockRange` distribution extended with *ghost regions*. The ghost region is extra space "around the edges" of the locally held block of multiarray elements. These extra locations can cache some of the element values properly belonging to adjacent processors. With ghost regions, the inner loop of algorithms for stencil updates can be written in a simple way, since the edges of the block don't need special treatment in accessing neighboring elements. Shifted indices can locate the proper values cached in the ghost region. This is a very important feature in real codes, so HPJava has a specific extension to make it possible.

For ghost regions, we relax the rule that the subscript of a multiarray should be a distributed index. As special syntax, the following expression

$$\textit{name} \pm \textit{expression}$$

is a legal subscript if *name* is a distributed index declared in an `overall` or `at` control construct and *expression* is an integer expression—generally a small constant. This is called *shifted index.* The significance of the shifted index is that an element displaced from the original location will be accessed. If the shift puts the location outside the local block *plus* surrounding ghost region, an exception will occur.

Ghost regions are not magic. The values cached around the edges of a local block can only be made consistent with the values held in blocks on adjacent processes by a suitable communication. A library function called `writeHalo` updates the cached values in the ghost regions with proper element values from neighboring processes. Figure 3.11 is a version of the Laplace equation using red-black relaxation with ghost regions. The last two arguments of the `ExtBlockRange` constructor define the widths of the ghost regions added to the bottom and top (respectively) of the local block.

The `CollapsedRange` is a range that is not distributed, i.e. all elements of the range are mapped to a single process. The example

```
Procs1 q = new Procs1(3) ;
on(p) {
  Range x = new CollapsedRange(N) ;
  Range y = new BlockRange(N, q.dim(0)) ;

  double [[-,-]] a = new double [[x, y]] on p ;
  ...
}
```

creates an array in which the second dimension is distributed over processes in `q`, with the first dimension *collapsed.* Figure 3.12 illustrates the situation for the case `N` = 8.

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

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

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

  // Initialize ‘a’: set boundary values.
  overall(i = x for :)
    overall(j = y for : )
      if(i‘ == 0 || i‘ == N - 1 || j‘ == 0 || j‘ == N - 1)
        a [i, j] = i‘ * i‘ - j‘ * j‘;
      else
        a [i, j] = 0.0;

  // Main loop.
  Adlib.writeHalo(a);

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

Figure 3.11. Solution for Laplace equation using red-black with ghost regions in HPJava.

However, if an array is declared in this way with an collapsed range, one still has the awkward restriction that a subscript must be a distributed index—one can't exploit the implicit locality of a collapsed range, because the compiler treats `CollapsedRange` as just another distributed range class.

In order to resolve this common problem, HPJava has a language extension adding the idea of *sequential dimensions*. If the type signature of a distributed array has the asterisk symbol, `*`, in a specific dimension, the dimension will be implicitly collapsed, and can have a subscript with an integer expression like an ordinary multiarray. For example:

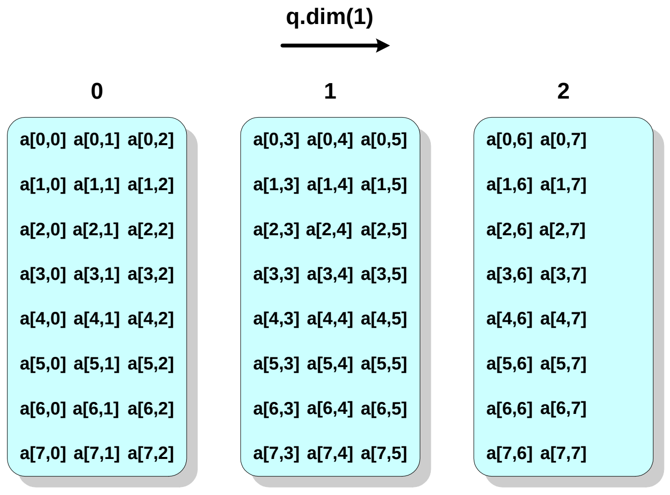

Figure 3.12. A two-dimensional array, `a`, distributed over one-dimensional grid, `q`.

```
Procs1 p = new Procs1(3);
on(p) {
  Range x = new BlockRange(N, p.dim(0)) ;
  double [[*,-]] a = new double [[N, x]] on p ;
  ...
  at(j = y [1]) a [6, j] = a [1, j] ;
}
```

An `at` construct is retained to deal with the distributed dimension, but there is no need for distributed indices in the sequential dimension. The array constructor is passed integer extent expressions for sequential dimensions. All operations usually applicable to distributed arrays are also applicable to arrays with sequential dimensions. As we saw in the previous section, it is also possible for all dimensions to be sequential, in which case we recover sequential multidimensional arrays.

On a two-dimensional process grid, suppose a one-dimensional distributed array with `BlockRange` distribution format is created, and the array dimension is distributed over the first dimension of the process grid, but no array dimension distributed over the second:

```
Procs2 p = new Procs2(P, P);
on(p) {
  Range x = new BlockRange(N, p.dim(0)) ;
  double [[-]] b = new double[[x]] on p ;
  ...
}
```

```
Procs2 p = new Procs2(P, P) ;
on(p) {
  Range x = new BlockRange(N, p.dim(0)) ;
  Range y = new BlockRange(N, p.dim(1)) ;

  double [[-,-]] c = new double [[x, y]] on p ;
  double [[-,*]] a = new double [[x, N]] on p ;
  double [[*,-]] b = new double [[N, y]] on p ;

  ... initialize 'a', 'b'

  overall(i = x for :)
    overall(j = y for :) {
      double sum = 0 ;
      for(int k = 0 ; k < N; k++) sum += a [i, k] * b [k, j] ;
      c [i, j] = sum ;
    }
}
```

Figure 3.13. A direct matrix multiplication in HPJava.

The interpretation is that the array, `b` *replicated* over the second process dimension. Independent copies of the whole array are created at each coordinated where replication occurs.

Replication and collapsing can both occur in a single array. For example,

```
Procs2 p = new Procs2(P, P);
on(p) {
  double [[*]] c = new double[[N]] on p ;
  ...
}
```

The range of `c` is sequential, and the array is replicated over both dimensions of `p`.

A simple and potentially efficient implementation of matrix multiplication can be given if the operand arrays have carefully chosen replicated/collapsed distributions. The program is given in Figure 3.13. As illustrated in Figure 3.14, the rows of `a` are replicated in the process dimension associated with `y`. Similarly the columns of `b` are replicated in the dimension associated with `x`. Hence all arguments for the inner scalar product are already in place for the computation—no communication is needed.

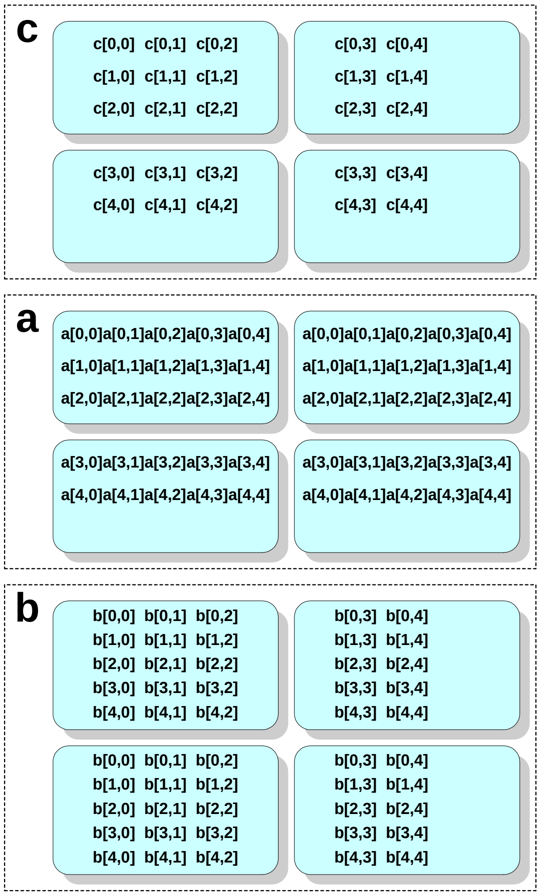

Figure 3.14. Distribution of array elements in example of Figure 3.13. Array `a` is replicated in every column of processes, array `b` is replicated in every row.

We would be very lucky to come across three arrays with such a special *alignment relation* (distribution format relative to one another). There is an important function in the Adlib communication library called `remap`, which takes a pair of arrays as arguments. These must have the same shape and type, but they can have unrelated distribution formats. The elements of the source array are copied to the destination array. In particular, if the destination array has a replicated mapping, the values in the source array are broadcast appropriately.

Figure 3.15 shows how we can use `remap` to adapt the program in Figure 3.13 and create a general purpose matrix multiplication routine. Besides the `remap` function, this example introduces the two inquiry methods `grp()` and `rng()` which are defined for any multiarray.

```
void matmul(double [[,]] c, double [[,]] a, double [[,]] b) {
  Group p = c.grp() ;
  Range x = c.rng(0), y = c.rng(1) ;
  int   N = a.rng(1).size() ;

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

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

  on(p)
    overall(i = x for :)
      overall(j = y for :) {
        double sum = 0 ;
        for(int k = 0 ; k < N ; k++) sum += ta [i, k] * tb [k, j] ;
        c [i, j] = sum ;
      }
}
```

Figure 3.15. A general matrix multiplication in HPJava.

The inquiry `grp()` returns the distribution group of the array, and the inquiry `rng(`r`)` returns the rth range of the array. The argument r is in the range $0, \ldots, R-1$, where R is the rank (dimensionality) of the array.

Figure 3.15 also introduces the most general form of multiarray constructors. So far, we have seen arrays distributed over the whole of the active process group, as defined by an enclosing `on` construct. In general, an *on clause* attached to an array constructor itself can specify that the array is distributed over some subset of the active group. This allows one to create an array outside the `on` construct that will processes its elements.

3.4.6 Array Sections

HPJava supports subarrays modeled on the *array sections* of Fortran 90. The syntax of an array section is similar to a multiarray element reference, but uses double brackets. Whereas an element reference is a variable, an array section is an expression that represents a new multiarray object. The new multiarray object doesn't contain new elements. It contains a subset of the elements of the parent array.

The options for subscripts in array sections are more flexible than those in multiarrays. Most importantly, subscripts of an array section may be *triplets*[4] as in Fortran 90. An array section expression has an array dimension for each triplet subscript. So the rank of an array section expression is equal to the number of triplet subscripts. If an array section has some scalar subscripts similar to those in element references, the rank of the array section is lower than the parent array. The following HPJava code contains the example of array section expressions.

```
Procs2 p = new Procs2(P, P) ;
on(p) {
  Range x = new BlockRange(N, p.dim(0)) ;
  Range y = new BlockRange(N, p.dim(1)) ;

  double [[-,-]] a = new double[[x, y]] on p ;
  double [[-]]   b = a [[0, :]] ;
  foo(a [[0 : N / 2 - 1, 0 : N - 1 : 2]]) ;
}
```

The first array section expression appears on the right hand side of the definition of `b`. It specifies `b` as an alias for the first row of `a` (Figure 3.16). In an array section expression, *unlike* in an array element reference, a scalar subscript is always allowed to be an integer expression[5]. The second array section expression appears as an argument to the method `foo`. It represents a two-dimensional, $N/2 \times N/2$, subset of the elements of `a`, visualized in Figure 3.17.

What are the ranges of `b` from the above example? That is, what does the `rng()` inquiry of `b` return? The ranges of an array section are called *subranges*, as a matter of fact. These are different from ranges considered so far. For the completeness, the language has a special syntax for writing subranges directly. Ranges identical to those of the argument of `foo` from the above could be generated by

```
Range u = x [[0 : N / 2 - 1]] ;
Range v = y [[0 : N - 1 : 2]] ;
```

[4]The syntax for triplets is directly from Fortran 90.
[5]Distributed indices are allowed as well, if the location is in the proper range.

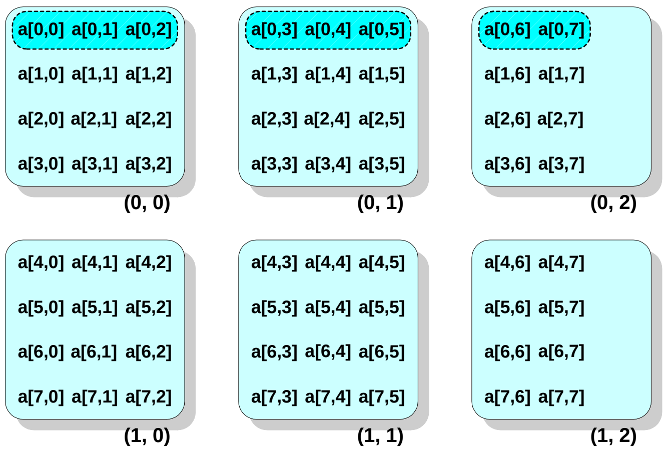

Figure 3.16. A one-dimensional section of a two-dimensional array (shaded area).

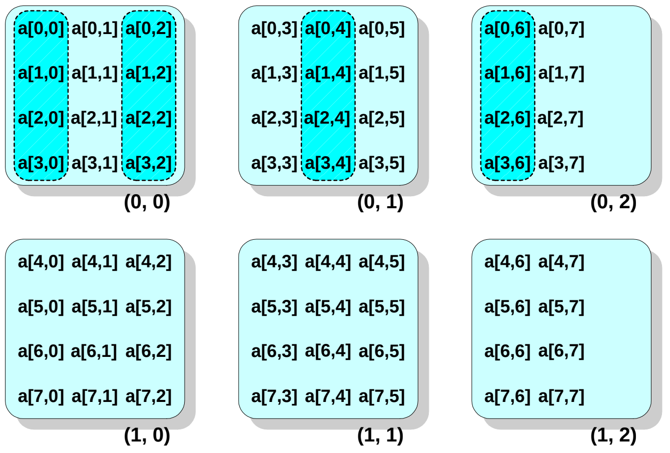

Figure 3.17. A two-dimensional section of a two-dimensional array (shaded area).

which look quite natural and similar to the subscripts for location expressions. But, the subscript is a triplet. The global indices related with the subrange `v` are in the range $0, \ldots,$ `v.size()` $- 1$. A subrange inherits locations from its parent range, but it specifically does not inherit global indices from the parent range.

As triplet section subscripts motivated us to define subranges as a new kind range, scalar section subscripts will drive us to define a new kind of group. A *restricted group* is the subset of processes in some parent group to which a specific location is mapped. From the above example, the distribution group of `b` is the subset of processes in `p` to which the location `x[0]` is mapped. In order to represent the restricted group, the division operator is overloaded. Thus, the distribution group of `b` is equal to `q` as follws;

```
Group q = p / x[0] ;
```

In a sense the definition of a restricted group is tacit in the definition of an abstract location. In Figure 3.6 section the set of processes with coordinates $(0,0)$, $(0,1)$ and $(0,2)$, to which location `x[1]` is mapped, can now be written as

```
p / x[1] ;
```

and the set with coordinates $(0,1)$ and $(1,1)$, to which `y[4]` is mapped, can be written as

```
p / y[4] ;
```

The intersection of these two—the group containing the single process with coordinates $(0,1)$—can be written as

```
p / x[1] / y[4] ;
```

or as

```
p / y[4] / x[1] ;
```

We didn't restrict that the subscripts in an array section expression *must* include some triplets (or ranges). It is legitimate for all the subscripts to be "scalar". In this case the resulting "array" has rank 0. There is nothing pathological about rank-0 arrays. They logically maintain a single element, bundled with the distribution group over which this element is replicated. Because they are logically multiarrays they can be passed as arguments to Adlib functions such as `remap`. If `a` and `b` are multiarrays, we cannot usually write a statement like

```
a [4, 5] = b [19] ;
```

since the elements involved are generally held on different processors. As we have seen, HPJava imposes constraints that forbid this kind of direct assignment between array element references. However, if it is really needed, we can usually achieve the same effect by writing

```
Adlib.remap (a [[4, 5]], b [[19]]) ;
```

The arguments are rank-0 sections holding just the destination and source elements.

The type signature for a rank-0 array with element of type T is written T`[[]]`[6]. Rank-0 multiarrays, which we will also call simply "scalars", can be created as follows:

```
double [[-,-]] a = new double [[x, y]] ;
double [[]] c = new double [[]] ;
Adlib.remap (c, a [[4, 5]]) ;
double d = c[];
```

This example illustrates one way to broadcast an element of a multiarray: remap it to a scalar replicated over the active process group. The element of the scalar is extracted through a multiarray element reference with an empty subscript list. (Like any other multiarray, scalar constructors can have `on` clauses, specifying a non-default distribution group.)

3.4.7 Low Level SPMD Programming

Dimension splitting is defined as an extension of the array section. The extra syntax is minimal, but the effect is useful. So far, we have identified a specific element by giving a *global subscript* in the distributed range. Alternatively, dimension splitting supports for accessing an element using its process coordinate and local subscript[7], by allowing a distributed dimension of an array to be temporarily treated as *two dimensions.*

If a particular dimension of an array section has the symbol `<>` as a subscript, the dimension of the array section is supposed to be split. Splitting a dimension creates two dimensions, a distributed dimension and a sequential dimension. The range of the distributed dimension is the process dimension over which the original array dimension was distributed. The local block of the original array is embedded in the sequential dimension of the result array. The two new dimensions appear consecutively in the signature of the result array, distributed dimension first.

[6]In constrast with a one-dimensional multiarray, which looks like either double `[[-]]` or double `[[*]]`.
[7]A subscript within the array segment held on the associated process.

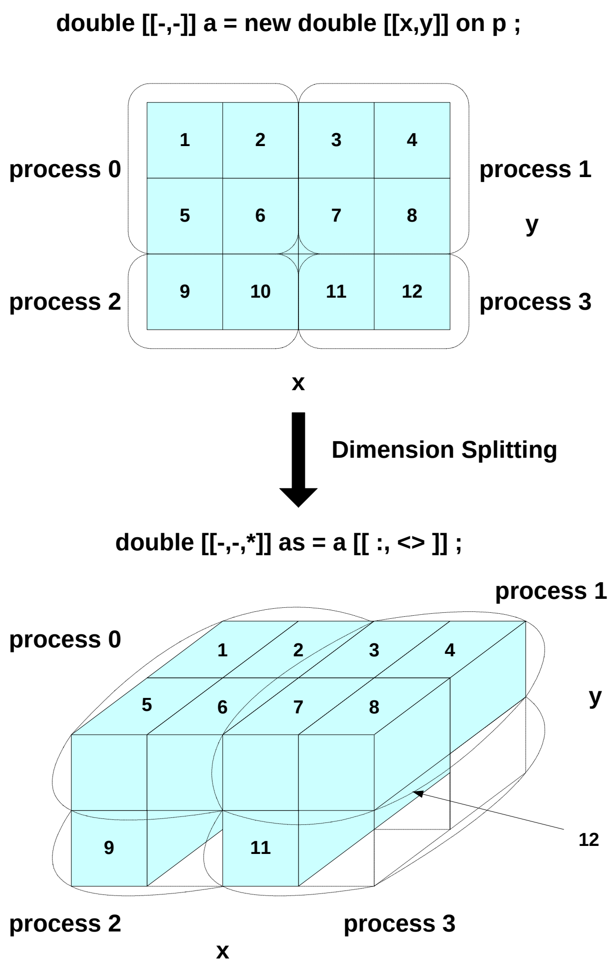

Figure 3.18. The dimension splitting in HPJava.

```
Procs2 p = new Procs2(4, 3) ;
on(p) {
  Range x = new BlockRange(N, q.dim(0)) ;
  Range y = new BlockRange(N, q.dim(1)) ;

  double [[-,-]]    a = new double [[x, y]] on p ;
  double [[-,-,*]] as = a [[:, <>]] on p ;
  ...
}
```

Figure 3.18 illustrated the dimension splitting for the above example.

The N-Body problem in Figure 3.19 on the page 51 initially creates the arrays as distributed, one-dimensional arrays, and uses this convenient form to initialize them. It then

```
Procs1    p = new Procs1(P) ;
Dimension d = p.dim(0) ;
on(p) {
  Range x = new BlockRange(N, d) ;

  double [[-]] f = new double[[x]] on p ;
  double [[-]] a = new double[[x]] on p ;
  double [[-]] b = new double[[x]] on p ;

  ... initialize ‘a’ ...

  overall(i = x for :) {
    f [i] = 0.0 ;
    b [i] = a [i] ;
  }

  // split ‘x’ dimensions:
  double [[-,*]] fs = f [[<>]], as = a [[<>]], bs = b [[<>]] ;

  for(int s = 0 ; s < P ; s++) {
    overall(i = d for :)
      for(int j = 0 ; j < B ; j++)
        for(int k = 0 ; k < B ; k++)
          fs [i, j] += force(as [i, j], bs [i, k]) ;

    // cyclically shift ‘bs’ in ‘d’ dim...

    Adlib.cshift(tmp, bs, 1, 0) ;
    HPutil.copy(bs, tmp) ;
  }
}
```

Figure 3.19. N-body force computation using dimension splitting in HPJava.

uses a split representation of the same arrays to compute the force array. Note that, because `as` and `fs` are semantically *sections* of `a` and `f`, they share common elements—they provide aliases through which the same element variables can be accessed. So when the computation loop is complete, the vector of forces can be accessed again through the one-dimensional array `f`. This is likely to be what is needed in this case. In Figure 3.19, B is the local block size N/P (P must exactly divide N for this example to work correctly).

3.5 Adlib for HPJava

One HPJava communication library is called *Adlib* [11]. This library is not supposed to have a uniquely special status so far as the HPJava language is concerned. Eventually HPJava bindings for other communication libraries will probably be needed. For example, the Adlib library does not provide the one-sided communication functionality of libraries like the Global Arrays toolkit [35]. It doesn't provide optimized numerical operations on distributed arrays like those in ScaLAPACK [4]. Moreover, it doesn't provide highly optimized collective operations for irregular patterns of array access, like those in CHAOS [16]. These libraries (and others) work with distributed arrays more or less similar in concept to HPJava distributed arrays. Bindings to these or functionally similar libraries might one day be made available for HPJava.

All collective operations in Adlib are based on *communication schedule* objects. Each kind of operation has a related class of schedules. Specific instances of the schedules involving particular array data arrays and other parameters are constructed by the class constructors. The communications needed to effect the operation are initiated by *executing* a schedule object. A single schedule may be executed and reused many times if the same pattern of accesses is repeated. The effective cost of computations involved in constructing a schedule can be decreased in this way. This paradigm was championed in PARTI.

There are three categories of collective operation in Adlib; regular communications, reduction operations, and irregular communications. It also provides a few I/O operations.

The regular communication operations are `remap()`, `writeHalo()`, `shift()`, and `cshift()`. For example, The `remap()` method takes two distributed array arguments—a source array and a destination. These two arrays must have the same size and shape, but they can have any, unrelated, distribution formats.

Let's see an example from the Adlib communication library. `Remap` is a communication schedule for copying the elements of one distributed array (or section) to another. The effect of the operation is to copy the values of the elements in the source array to the corresponding elements in the destination array, performing any communications necessary to do that. If the destination array has *replicated* mapping, the `remap()` operation will broadcast source values to all copies of the destination array elements.

As currently implemented, we can summarize the available `remap()` signatures in the notation:

```
void remap(T # destination, T # source)
```

where the variable T runs over all primitive types and `Object`, and the notation T `#` means a multiarray of arbitrary rank, with elements of type T.

Reduction operations take one or more distributed arrays as input. They combine the elements to produce one or more scalar values, or arrays of lower rank. Adlib provides a fairly large set of reduction operations, mimicking the reductions available as "intrinsic functions" in Fortran. The operations are `maxval()`, `minval()`, `sum()`, `product()`, `dotProduct()`, etc.

Adlib has some support for irregular communications in the form of collective `gather()` and `scatter()` operations. Moreover, the current HPJava version of Adlib has simple and rudimentary I/O functions such as `aprintf()` and `gprintln()`.

3.6 Discussion

We have reviewed the HPspmd programming model. It is a flexible hybrid of HPF-like data-parallel language features and the popular, library-oriented, low-level SPMD style, omitting some basic assumptions of the HPF model. Moreover, we covered the importance of integration of high-level libraries, such as numerical libraries, parallel programming libraries with distributed arrays, and irregular problem solving libraries, in the HPspmd programming model.

HPJava is an implementation of our HPspmd programming model. It extends the Java language with some additional syntax and with some pre-defined classes for handling multiarrays. By providing true multidimensional arrays, multiarrays, array sections, control constructs (e.g. `on`, `overall`, and `at`), HPJava becomes a flexible and SPMD programming language. For example with ghost regions the inner loop of algorithms for stencil updates can be written in a simple way. Moreover, with dimension splitting, parallel programming developers can implement low level SPMD programs in relatively simple ways.

Adlib is an HPJava communication library and its collective operations are regular communications, reduction operations, and irregular communications, based on the communication schedule object.

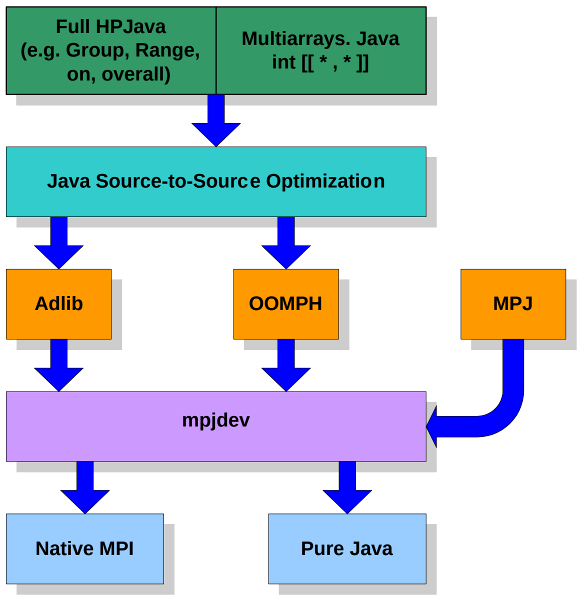

Figure 3.20. The HPJava Architecture.

Figure 3.20 shows the hierarchy of the HPJava system we have seen in this chapter. In the next chapter, we will review the HPJava compilation system.

CHAPTER 4

COMPILATION STRATEGIES FOR HPJAVA

In this chapter, we will describe the overall picture of compilation strategies for HPJava. First, we will cover the design philosophy for general multiarray types and HPspmd classes. Secondly, we will overview the HPJava asbtract syntax tree and its nodes from a design pattern point of view. Thirdly, we will represent HPJava front-end: type-analysis, reachability, and definite (un)assignment. Fourthly, we will see the pre-translator for the HPJava system.

Finally, we will introduce the basic translation scheme where we start researching efficient compilation strategies for the HPJava system. The translation scheme is the initial version we adopted to the current HPJava system. There are many places where it may re-designed for efficient compilation in the future.

4.1 Multiarray Types and HPspmd Classes

A multiarray type is not treated as a class type. If we said, "multiarrays have a class", it would probably commit us to either extending the definition of class in the Java base language, or creating genuine Java classes for each type of HPJava array that might be needed.

Since class is so fundamental in Java, extending its definition seems impractical. On the other hand, creating Java classes for each multiarray type has its own semantic problems. Presumably the associated class types would have to embed all the information for multiarray types we have described for HPJava. Since T can be any Java type, there is an infinite number of multiarray types. Thus, whenever needed at compiler-time, the associated classes would have to be created on demand by the translator. Now, we can ask ourselves a question, "Does the translator have to generate distributed class files whenever we need them?" If the answer is "yes", how are we supposed to manage all the generated class files?

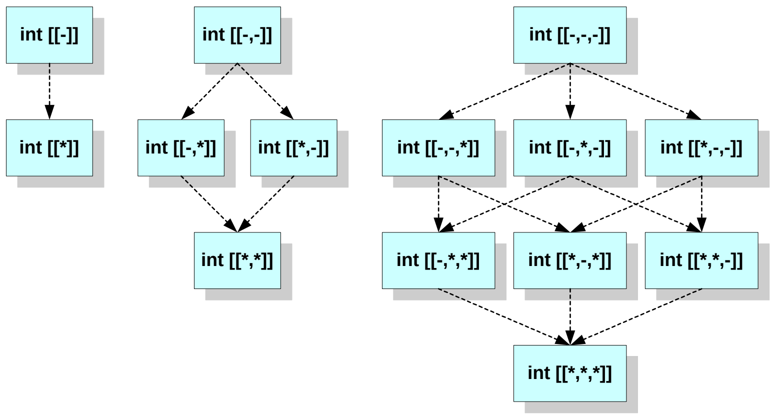

Figure 4.1. Part of the lattice of types for multiarrays of `int`.

Figure 4.1 illustrates that multiarray types have a quite complex, multiple-inheritance-like lattice. This kind of type lattice *can* be recreated in Java by using interface types. But then, when we generate a new array class, we have to make sure all the interfaces it implements directly and indirectly are also generated.

This seems too complicated. So we decided that a multiarray type is *not* a class type. The fact that a multiarray is not a member of any Java class or primitive type means that a multiarray *cannot* be an element of an ordinary Java array, nor can a multiarray reference be stored in a standard library class like `Vector`, which expects an `Object`. In practise, this is not such a big restriction as it sounds. We *do* allow multiarrays to be fields of classes. So, the programmer can make wrapper classes for specific types of multiarray. For example, suppose we need a "stack" of two-dimensional multiarrays of floating point numbers. We can make a wrapper class, `Level`, as follows:

```
class Level implements hpjava.lang.HPspmd {
  public Level(int [[-,-]] arr) { this.arr = arr ; }
  public int [[-,-]] arr ;
}

Range x, y ;

Level [] stack = new Level [S] ;
for (int l = 0 ; l < S ; l++)
  stack [l] = new Level(new int [[x, y]]) ;
```

Thus, it is minor inconvenience that multiarrays are not treated as normal objects, not a fundamental limitation[1].

The HPJava translator must tell parallel code from sequential code. It introduces a special interface, `hpjava.lang.HPspmd`, which must be implemented by any class that uses the parallel syntax.

An *HPspmd class* is a class that implements the `hpjava.lang.HPspmd` interface. Any other class is a *non-HPspmd class*. Also, an interface that extends the `hpjava.lang.HPspmd` interface is called an *HPspmd interface*, and any other interface is a *non-HPspmd interface*. The parallel syntax of HPJava can only be used in methods and constructors declared in HPspmd classes and interfaces.

To further discourage invocation of HPspmd code from non-HPspmd code, the HPJava translator imposes the following limitations:

1. If a class or interface inherits a method with the same signature from more than one of its superclasses and superinterfaces, either all declarations of the method must be in HPspmd classes and interfaces, or all declarations of the method must be in non-HPspmd classes and interfaces. So an inherited method can always be unambiguously classified as an HPspmd method or a non-HPspmd method.

2. An HPspmd class or interface may not override a non-HPspmd method.

The details of HPspmd class restrictions can be found in [9].

Many of the special operations in HPJava rely on the knowledge of the currently active process group—the *APG*. This is a context value that will change during the course of the program as distributed control constructs limit control to different subsets of processors. In the current HPJava translator the value of the APG is passed as a hidden argument to methods and constructors of HPspmd classes (so it is handled something like the `this` reference in typical object-oriented languages).

[1]Note that Fortran 90 also does not allow arrays of arrays to be declared directly. If they are needed they have to be simulated by introducing a derived data-type wrapper, just as we introduced a class here.

4.2 Abstract Syntax Tree

The HPJava grammar was originally extended from a grammar freely supported by WebGain's Java Compiler Compiler (JavaCC) [25]. JavaCC is a parser generator and a tree builder with an add-on application, JJTree. Since HPJava is a superset of Java, the new constructs, multiarrays, array sections, etc, are added into the original Java grammar. With this new grammar, JavaCC generates the HPJava parser generator and the HPJava Abstract Syntax Tree (AST).

In order to traverse the HPJava AST, the compiler adopts the *visitor* design pattern [20]. It would be confusing to have type-checking code mixed with code-generator code or flow analysis code. Moreover, adding a new operation usually requires recompiling all of these classes. It would be better if each new operation could be added separately, and the syntax node classes were independent of the operation that applies to them. We can have both by packing related operations from each class in a separate object, called `Visitor`, and passing it to elements of AST as it is traversed. When an element "accepts" the `Visitor`, it sends a request to the `Visitor` that encodes the element's class.

We see how to traverse AST for an expression, $a + b$ with some examples of Figure 4.2. AST has a visitor class, `ASTVisitor`, another visitor class, `HPJavaTypeChecker`, the binary expression syntax node, `BinExprNode`, and the identifier syntax node, `IdentifierNode`.

`ASTVisitor` is the super class of actual traversal classes such as `HPJavaTypeChecker`, `Translator`, etc, that we will see in the following sections. When the `HPJavaTypeChecker` meets a binary expression, $a + b$, the `accept` method for the left-handed node, a, is called. This method calls the visit method for `IdentifierNode` in `HPJavaTypeChecker`. After actual operations for type-checking are done, the `accept` method for the right-handed node, b, is called. Using the visitor pattern, the actual syntax tree, `ASTVisitor`, doesn't have to be modified and recompiled whenever some operations are added on the actual traversal classes such as `HPJavaTypeChecker`. This means that all visitors are well isolated from each other. Moreover, the visitor pattern naturally makes the HPJava compiler system traverse in top-down way.

The HPJava AST consists of 128 syntax nodes. Figure 4.3, Figure 4.4, and Figure 4.5 partly cover the hierarchy of the HPJava AST nodes.

```
class ASTVisitor {
  ...
  void visit (BinExprNode n) {
    n.left.accept(this);
    n.right.accept(this);
  }
  ...
  void visit (IdentifierNode n) { }
}
class HPJavaTypeChecker extends ASTVisitor {
  ...
  void visit(BinExprNode n) {
    ...  // type-checking code
    n.left.accept(this);
    ...  // type-checking code
    n.right.accept(this);
    ...  // type-checking code
  }
  void visit (IdentifierNode n) {
    ...  // type-checking code
  }
}
class IdentifierNode extends ASTNode {
  ...
  public void accept(ASTVisitor v) {
    v.visit(this);
  }
}
```

Figure 4.2. Example of HPJava AST and its nodes.

4.3 Type-Checker

The current version of the HPJava type-checker (front-end) has three main phases; *type-analysis*, *reachability analysis*, and *definite (un)assignment analysis.* Development of this type-checker has been one of the most time-consuming parts during the implementation of the whole HPJava compiler.

The three phases are basically inspired by carefully organizing "The Java Language Specification Second Edition" (JLS)[26]. The current HPJava type-checker system only

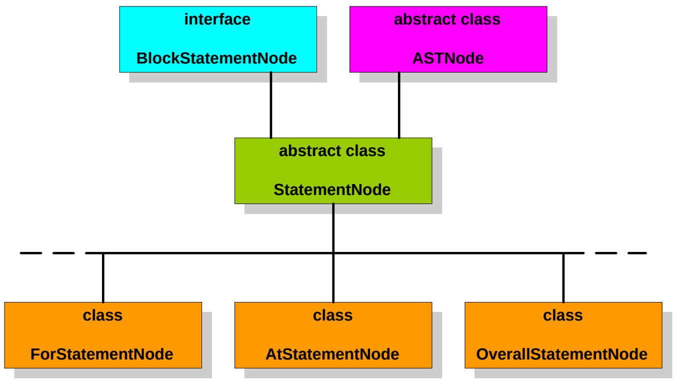

Figure 4.3. Hierarchy for statements.

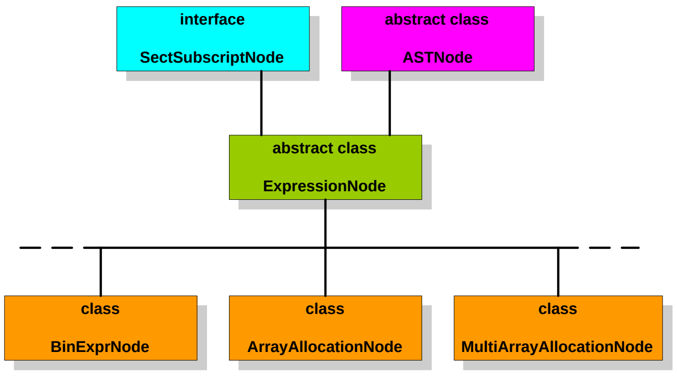

Figure 4.4. Hierarchy for expressions.

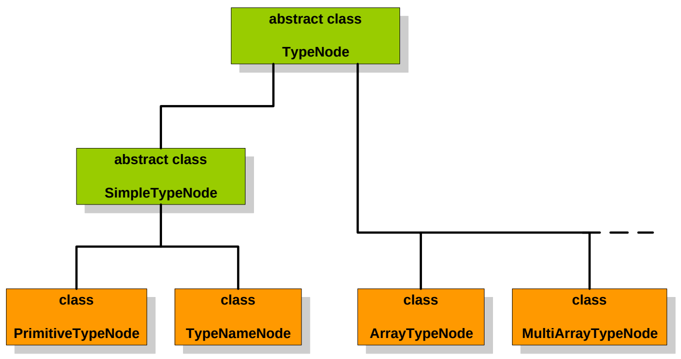

Figure 4.5. Hierarchy for types.

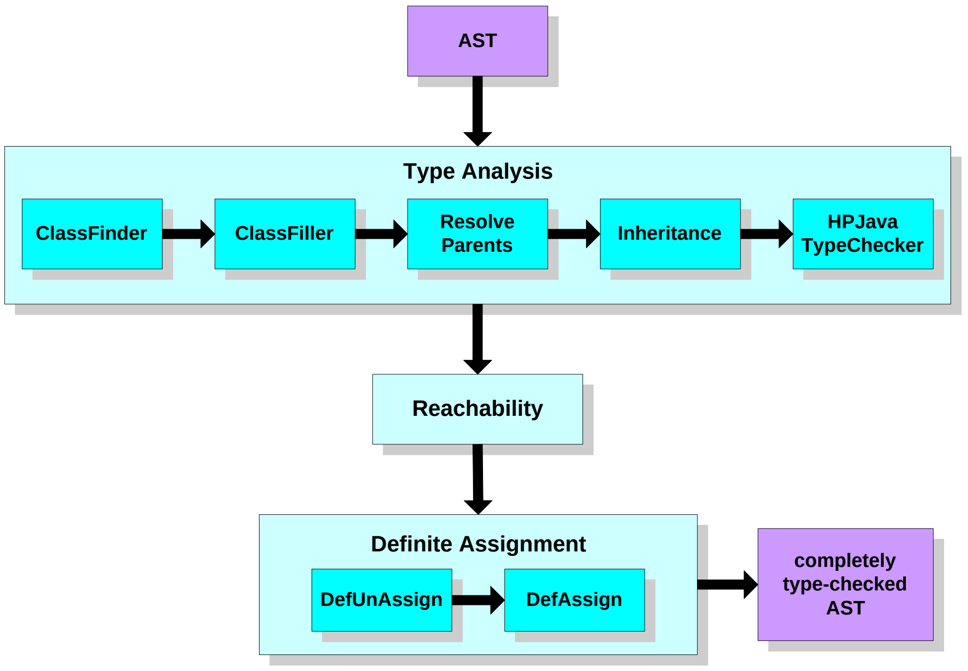

Figure 4.6. The architecture of HPJava Front-End.

supports exactly what JLS says. However, The Java language itself keeps evolving in a slow manner. For example, Sun's SDK Java compiler supports some changes to JLS. For the moment, the HPJava type-checker system will not be changed until the new edition of JLS is released.

Figure 4.6 shows the complete architecture of HPJava front-end.

4.3.1 Type Analysis

The first phase is *type-analysis*. It has five subordinate phases: *ClassFinder*, *ResolveParents*, *ClassFiller*, *Inheritance*, and *HPJavaTypeChecker*.

1. `ClassFinder` collects some simple information about top-level and nested class or interface declarations, such as names of the classes, the names of super class, and the names of super interfaces.

2. `ResolveParents` resolves class's proper super class and super interfaces using the information from ClassFinder.

3. `ClassFiller` fulfills a more complicated missions. It collects all of the rest information about top-level and nested class or interface declarations, such as field declarations, method declarations, constructor declarations, anonymous class allocations, and so on. `ClassFiller` also collects and resolves single-type-import declarations and type-import-on-demand declarations.

4. `Inheritance` collects and resolves the method inheritance, overriding, and hiding information to be used in `HPJavaTypeChecker`.

5. `HPJavaTypeChecker` does type-checking on statements, statement expressions, and expressions, including all ordinary Java and newly introduced HPJava constructs, multiarrays, etc.

4.3.2 Reachability and Definite (Un)assignment Analysis

The second phase is *reachability analysis*, carrying out a conservative flow analysis to make sure all statements are reachable. The idea of `Reachability` is that there must be some possible execution path from the beginning of the constructor, method, instance initializer or static initializer that contains the statement to the statement itself. The analysis takes into account the structure of statements. Except for the special treatment of while, do, and for statements whose condition expression has the constant value true, the values of expressions are not taken into account in the flow analysis.

The third phase is *definite (un)assignment analysis*. It consists of two parts, *DefAssign* and *DefUnassign*. Each local variable and every blank final must have a definitely assigned value when any access of its value occurs. Similarly, every blank final variable must be assigned at most once; it must be definitely unassigned when an assignment to it occurs. The HPJava compiler must carry out a specific conservative flow analysis to make sure that, for every access of a local variable or blank final field *f*, *f* is definitely assigned before the access, and, for every assignment to a blank final variable, the variable is definitely unassigned before the assignment.

`DefUnassign` checks the definite unassignment rules for Java, implementing the flow analysis of JLS. This applies three checks that are essentially "static" (i.e. independent of the dataflow analysis):

1. A final field or final local variable cannot appear as the left-hand operand of an assignment operator, or as an operand of ++ or −−, unless the variable is blank final.

2. Moreover, a blank final field can only appear as the left-hand operand of an assignment operator, or as an operand of ++ or −−, if it is referenced through its simple name, or has unqualified 'this' as prefix (this is a slightly ad hoc restriction, but we follow JDK).

3. A blank final field that appears as the left-hand operand of an assignment operator, or as an operand of ++ or −−, must not be declared as a member of a superclass (of "the current class"). Superclass fields are considered to be previously assigned.

These are prerequisite to the main business:

- Any blank final variable appearing as the left-hand operand of an assignment operator, or as an operand of ++ or −−, must be definitely unassigned at the point where it appears (after evaluation of the right-hand operand, in the case of an assignment).

`DefAssign` checks the definite assignment rules for Java, implementing the flow analysis of JLS. Main checks implemented here are:

1. All class variables (static fields) in a class are definitely assigned after the static initialization code in the class (initializers for static fields and static initializer blocks) has been executed.

2. All instance variables (non-static fields) in the class are definitely assigned after the instance initialization code (initializers of non-static fields and instance initializer blocks, followed by the body of any single constructor in the class) has been executed.

3. A blank final field declared in a class is definitely assigned before every use of its value anywhere in the class, provided those uses takes the form of a simple, unqualified name. (If the use has a more complicated form, the unassigned, default value may be visible. The restriction to simple names is slightly ad hoc, but we follow JDK).

4. A local variable that is not initialized in its declaration is definitely assigned before every use of its value.

4.4 Pre-translation

Currently, the HPJava translator has two phases, *pre-translation* and *translation.* Pre-translation reduces the source HPJava program to an equivalent HPJava program in a *restricted form* of the full HPJava language. The translation phase transforms the pre-translated program to a standard Java program. The main reason the current HPJava translator has two phases is to make a basic translation scheme easy and clear by transforming certain complex expressions involving multiarrays into simple expressions in the pre-translation phase. Thus, the restricted form generated by the pre-translator should be suitable for being processed in the translation phase. The basic translation schema in section 4.5 will be applied to the pre-translated HPJava programs.

The rules for restricted forms are as follows;

Definition 1 *Composite multiarray expressions* are multiarray creation expressions, multiarray-valued method invocation expressions, multiarray section expressions, and multiarray restriction expressions.

Definition 2 *Simple expressions are* constant expressions, Local variable references, Fields referenced using the syntax of a simple identifier, f, assuming the referenced field is not *volatile*, and the keyword `this`.

Rule 1 A composite multiarray expression may *only* appear as the right-hand-side of a top-level assignment (or, in the case of multiarray-valued method invocation, as a standalone statement expression). Here, a *top-level assignment* is defined to be an assignment expression appearing as a *statement expression* in a program block. This specifically excludes assignments appearing in for-loop headers. Also note that a variable declarator with an initializer is *not* an assignment expression, so the restricted form does not allow composite multiarray expressions to appear as initializers in declarations.

Rule 2 All of the following *must* be simple expressions:

a) the target object in accesses to multiarray-valued fields,

b) the boolean expression in a multiarray-valued conditional expressions, and

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

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

  ... initialize 'a'

  double [[-]] b = a [[0, :]] ;

  foo(a [[0 : N / 2 - 1, 0 : N - 1 : 2]]) ;
}
```

Figure 4.7. Example source program before pre-translation.

c) the array operand in array-restriction expression,

d) the array operand of multiarray inquiry operations (`rng()`, `grp()`, etc),

e) the range and group parameters of multiarray creation expressions,

f) the range expression in the header of `overall` and `at` constructs.

Figures 4.7 and 4.8 illustrate the process of pre-translation. The source code involves an array section. In this example pre-translation shifts two kinds of composite array expression—multiarray creation and array section—into top-level assignments. In the case of the array creation expressions it simply has to split the array declarations with initializers to blank declarations plus an assignment. In the case of the section, it has to introduce temporaries.

Besides simplifying variables, the pre-translation phase also breaks up multiarray declarations that declare several variables, outputting a series of declarations of *individual* multiarray variables. This slightly simplifies the work of the translation phase.

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

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

  ... initialize 'a'

  double [[-]] b ;
  b = a [[0, :]] ;

  double [[-, -]] __$t0 ;
  __$t0 = a [[0 : N / 2 - 1, 0 : N - 1 : 2]] ;
  foo(__$t0) ;
}
```

Figure 4.8. Example source program after pre-translation.

4.5 Basic Translation Scheme

After the pre-translator converts an HPJava program to a suitable restricted form, the translator will start its actual functionality. This section will introduce basic, but unoptimized translation strategies that the current HPJava translator adopts.

Prior to describing the actual translation strategies, we will introduce some *translation functions.* The detailed and specific definitions for each will be described in the following subsections.

A function, $\mathbf{T}\,[e]$, on expression terms returns the result of translating an expression e, assuming that the expression is *not* a multiarray.

Translation functions for multiarray-expressions are more complicated. In the previous section, we defined a subset of *composite* multiarray-valued expressions. The remaining *non-composite* multiarray-valued expressions are:

- multiarray-valued local variable access,
- multiarray-valued field access,

- conditional expression, in which the second or third operands are multiarrays.

- assignment expression, in which the left-hand operand is a multiarray-valued variable.

The composite expressions only appear in restricted contexts and do not have translation functions in their own right (instead they are effectively handled as part of the translation of a top-level assignment statements). For *non-composite* multiarray-valued expressions there are $2 + R$ separate parts of the evaluation: $\mathbf{T}_{\mathrm{dat}}\,[e]$, $\mathbf{T}_{\mathrm{bas}}\,[e]$ and $\mathbf{T}_0\,[e]$, ..., $\mathbf{T}_{R-1}\,[e]$, where R is the rank of the array. The interpretation of these separate terms will be given in the following sections.

Finally the translation function for *statements*, $\mathbf{T}\,[S\,|p]$, translates the statement or block S in the context of p as active process group. In the scheme given below for translation of statements we will just use the name *apg* to refer to the effective active process group. Hence a schema of the form

SOURCE:
S

TRANSLATION:
S'

should be read more precisely as

SOURCE:
$s \quad \equiv \quad S$

TRANSLATION:
$\mathbf{T}\,[s\,|apg] \quad \equiv \quad S'$

Here we will describe important translation schemata. The full version of translation scheme for HPJava in [9] involves perhaps a couple of dozen such schemata of varying complexity.

4.5.1 Translation of a multiarray declaration

Figure 4.9 represents a schema for translating a multiarray declaration in a source HPJava program. T is some Java type, and a'_{dat}, a'_{bas} and $a'_0 \ldots a'_{R-1}$ are new identifiers, typically

SOURCE:

T `[[`$attr_0$`,  ...,  `$attr_{R-1}$`]]` a `;`

TRANSLATION:

T `[]` a'_{dat} `;`

`ArrayBase` a'_{bas} `;`

$DIMENSION\_TYPE(attr_0)$ a'_0 `;`
...
$DIMENSION\_TYPE(attr_{R-1})$ a'_{R-1} `;`

Figure 4.9. Translation of a multiarray-valued variable declaration.

derived from a by adding some suffixes, the strings $attr_r$ are each either a hyphen, -, or an asterisk, *, and the "macro" $DIMENSION\_TYPE$ is defined as:

$$DIMENSION\_TYPE(attr_r) \equiv \texttt{ArrayDim}$$

if the term $attr_r$ is a hyphen, or

$$DIMENSION\_TYPE(attr_r) \equiv \texttt{SeqArrayDim}$$

if the term $attr_r$ is an asterisk.

For instance, if a class in the source program has a field:

```
float [[-,*]] var ;
```

the translated class may be supposed to have the four fields as:

```
float [] var__$DDS ;

ArrayBase var__$bas ;

ArrayDim    var__$0 ;
SeqArrayDim var__$1 ;
```

Generally a rank-r multiarray in the source program is translated to $2 + r$ variables in the ordinary Java program. The first variable is an ordinary, one-dimensional, Java array

SOURCE:

$$a \texttt{ = new } T \texttt{ [[}e_0\texttt{, } \ldots\texttt{, } e_{R-1}\texttt{]] } bras \texttt{ on } p \texttt{ ;}$$

TRANSLATION:

```
int s = 1 ;
int b = 0 ;
p.checkContained(apg) ;
DimSet t = p.getDimSet() ;
DEFINE_DIMENSION(T_{R-1} [a] , e_{R-1}, s, b, t)
...
DEFINE_DIMENSION(T_0 [a] , e_0, s, b, t)
T_dat [a] = p.amMember() ? new T [s] bras : null ;
T_bas [a] = new ArrayBase(p, b) ;
```

where:

T is a Java type,
R is the rank of the created array,
each e_r is either a range-valued or an integer-valued simple expression in the source program,
the term *bras* is a string of zero or more bracket pairs, `[]`,
p is a simple expression in the source program,
the expression a is the assigned array variable in the source program,
s and b are names of new temporaries, and
the macro *DEFINE_DIMENSION* is defined in the text.

Figure 4.10. Translation of multiarray creation expression.

holding local elements. A struct-like object of type `ArrayBase` holds a base offset in this array and an HPJava `Group` object—the distribution group of the array. Simple `ArrayDim` and `SeqArrayDim`[2] objects each contain an integer stride in the local array and an HPJava `Range` object describing the dimensions of the distributed array.

[2]The class `SeqArrayDim` is a subclass of `ArrayDim`, specialized to parameterize sequential dimensions conveniently.

4.5.2 Translation of multiarray creation

Figure 4.10 represents a schema for translating the most general form of distributed array creation with an *on* clause. The pre-translator makes sure that multiarray creation only appears on the right-hand-side of a top-level assignment.

If the expression e_r is a range the macro *DEFINE_DIMENSION* is defined as follows:

$$\begin{aligned}
&\mathit{DEFINE\_DIMENSION}(a'_r, e_r, s, b, t) \equiv \\
&\qquad a'_r \texttt{ = } e_r\texttt{.arrayDim(}s\texttt{) ;} \\
&\qquad b \texttt{ += } s \texttt{ * } e_r\texttt{.loExtension() ;} \\
&\qquad s \texttt{ *= } e_r\texttt{.volume() ;} \\
&\qquad t\texttt{.remove(}e_r\texttt{.dim()) ;}
\end{aligned}$$

As each dimension is processed, the memory stride for the next dimension is computed by multiplying the variable s by the number of locally held range elements in the current dimension. The variable b is incremented to allow space for lower ghost regions, below the base of the physical array, if this is required for the ranges involved.

If the expression e_r is an integer then *DEFINE_DIMENSION*(a'_r, e_r, s, b, t) is evaluated the same as *DEFINE_DIMENSION*(a'_r, e_r, s) in the previous subsection.

The call to `checkContained()` throws a `GroupNotContainedException` runtime exception if p is not contained in the current APG. The variable t is also introduced for run-time checking of correct usage. The calls to `remove()` will throw a `DimensionNotInGroupException` runtime exception if any range specified for the array is not distributed over a dimension in p, *or* if any two specified ranges are distributed over the same dimension.

The method `arrayDim()` on the `Range` class creates an instance of `ArrayDim`, with the memory stride specified in its argument. It is used in place of a call to the `ArrayDim` constructor because `arrayDim()` has the property that if the range is actually a collapsed range, the returned object will be an instance of the `SeqArrayDim` subclass. This allows a new array created with a collapsed range to be cast to an array with a sequential dimension, should it prove necessary at a later stage.

4.5.3 Translation of the `on` construct

Figure 4.11 represents a schema for translating the `on` construct. This `on` construct establishes `p` as the *active process group* within its body. That is, after creating e_{grp}, we

SOURCE:

```
on ($e_{\text{grp}}$) $S$
```

TRANSLATION:

```
final APGGroup $p$ = $\mathbf{T}\left[e_{\text{grp}}\right]$.toAPG() ;
if ($p$ != null) {
    $\mathbf{T}\left[S\,|p\right]$
}
```

where:

e_{grp} is an expression in the source, and
p is the name of a new variable, and
S is a statement in the source program.

Figure 4.11. Translation of `on` construct.

will want to run some code within the process group. The `on` construct limits control to process in its parameter group. The code in the on construct is only executed by processes taht belong to e_{grp}.

4.5.4 Translation of the `at` construct

Figure 4.12 represents a schema for translating the `at` construct. Note that pre-translation will have reduced x to a simple expression.

The coordinate and local subscript associated with the specified location is returned by the method, `location()`, which is a member of the `Range` class. It takes one argument, the global subscript, and returns an object of class `Location` .

The *global index* for the index i is the value of *glb.* This value is used in evaluating the global index expression i‘.

The *local subscript* for the index i is the value of *sub.* This value is used in computation of offsets generated when i is used as an array subscript.

The *shift step* for the index i is the value of *shf.* This value is used in computation of offsets associated with shifted index subscripts.

SOURCE:

```
at ($i$ = $x$[$e_{\text{glb}}$]) $S$
```

TRANSLATION:

```
int $glb$ = $\mathbf{T}\left[e_{\text{glb}}\right]$ ;
Location $l$ =  $x$.location($glb$) ;

int $sub$ = $l$.sub ;
int $shf$ = $x$.str() ;

Dimension $dim$ = $l$.dim ;
if ($dim$.crd() == $l$.crd) {
    final APGGroup $p$ = $apg$.restrict($dim$) ;

    $\mathbf{T}\left[S\,|p\right]$
}
```

where:

i is an index name in the source program,
x is a simple expression in the source program,
e_{glb} is an expression in the source program,
S is a statement in the source program, and
glb, *l*, *sub*, *shf*, *dim* and *p* are the names of new variables.

Figure 4.12. Translation of `at` construct.

The *dimension* for the index i is the value of *dim*. One possible later use of this value is in computation the distribution group of a section subscripted with i.

4.5.5 Translation of the `overall` construct

Figure 4.13 represents a schema for translating the `overall` construct. The `localBlock()` method on the `Range` class returns parameters of the locally held block of index values associated with a range. These parameters are returned in another simple struct-like object of class `Block`. $\mathbf{T}[e]$ represents the translated form of expression e.

SOURCE:

```
overall (i = x for e_lo : e_hi : e_stp) S
```

TRANSLATION:

```
Block b =  x.localBlock(T[e_lo], T[e_hi], T[e_stp]) ;

int shf = x.str() ;

Dimension dim = x.dim() ;
final APGGroup p = apg.restrict(dim) ;

for (int l = 0 ; l < b.count ; l++) {
    int sub = b.sub_bas + b.sub_stp * l ;
    int glb = b.glb_bas + b.glb_stp * l ;

    T[S |p]
}
```

where:

i is an index name in the source program,
x is a simple expression in the source program,
e_{lo}, e_{hi}, and e_{stp} are expressions in the source,
S is a statement in the source program, and
b, *shf*, *dim*, p, l, *sub* and *glb* are names of new variables.

Figure 4.13. Translation of `overall` construct.

The *local subscript* for the index i is the value of *sub*. This value is used in subscripting multiarrays. The *global index* for the index i is the value of *glb*. This value is used in evaluating the global index expression i`.

Since we use the `localBlock()` method to compute parameters of the local loop, this translation is identical for every distribution format—block-distribution, simple-cyclic distribution, aligned subranges, and so on — supported by the language. Of course there is an overhead related to abstracting this local-block parameter computation into a method call; but the method call is made at most once at the start of each loop, and we expect that in

SOURCE:

$$e \;\equiv\; a\ \texttt{[}e_0\texttt{,}\ \ldots\texttt{,}\ e_{R-1}\texttt{]}$$

TRANSLATION:

$$\mathbf{T}\left[e\right] \;\equiv\; \mathbf{T}_{\text{dat}}\left[a\right]\ \texttt{[}\mathit{OFFSET}(a, e_0, \ldots, e_{R-1})\texttt{]}$$

where:

The expression a is the subscripted array,
each term e_r is either an integer, a distributed index name,
or a shifted index expression, and
the macro *OFFSET* is defined in the text.

Figure 4.14. Translation of multiarray element access.

many cases optimizing translators will recognize repeat calls to these methods, or recognize the distribution format and inline the computations, reducing the overhead further.

$\mathbf{T}\left[S\,|p\right]$ means the translation of S in the context of p as active process group.

4.5.6 Translation of element access in multiarrays

Figure 4.14 represents a general schema for translating element access. Here we only need to consider the case where the array reference is a multiarray. The macro *OFFSET* is defined as

$$\begin{aligned}
&\mathit{OFFSET}(a, e_0, \ldots, e_{R-1}) \equiv \\
&\qquad \mathbf{T}_{\text{bas}}\left[a\right]\texttt{.base} \;\texttt{+}\; \mathit{OFFSET\_DIM}(\mathbf{T}_0\left[a\right], e_0) \\
&\qquad\qquad\qquad\qquad \ldots \\
&\qquad\qquad\qquad\quad \texttt{+}\; \mathit{OFFSET\_DIM}(\mathbf{T}_{R-1}\left[a\right], e_{R-1})
\end{aligned}$$

There are three cases for the macro *OFFSET_DIM* depending on whether the subscript argument is a distributed index, a shifted index, or an integer subscripts (in a sequential dimension). We will only illustrate the case where e_r is a distributed index i. Then

$$\mathit{OFFSET\_DIM}(a'_r, e_r) \equiv \quad a'_r\texttt{.stride * } \mathit{sub}$$

where *sub* is the local subscript variable for this index (see the last section).

SOURCE:

v `=` a `[[`$subs_0$`, ...,` $subs_{R-1}$`]] ;`

TRANSLATION:

int b = $\mathbf{T}_{\text{bas}}[a]$.base ;
$PROCESS\_SUBSCRIPTS(v, 0, a, 0)$
$\mathbf{T}_{\text{dat}}[v]$ = $\mathbf{T}_{\text{dat}}[a]$
$\mathbf{T}_{\text{bas}}[v]$ = new ArrayBase($\mathbf{T}_{\text{bas}}[a]$.group, b) ;

where:

The expression v is the assigned array variable in the source program,
the simple expression a is the subscripted array in the source program,
each term $subs_s$ is either an integer, a distributed index, a shifted index, a triplet, or <>,
b is the name of a new temporary, and
the macro $PROCESS\_SUBSCRIPTS$ is defined in the text.

Figure 4.15. Translation of array section with no scalar subscripts.

4.5.7 Translation of array sections

The schema for translating array section is more complicated than any other translation schemata. We will break it down into three cases: the case where there are no scalar subscripts—integer or distributed index; the case where integer scalar subscripts appear in *sequential* dimensions only; and the general case where scalar subscripts may appear in distributed dimensions.

4.5.7.1 Simple array section – no scalar subscripts

Figure 4.16 represents a schema for translating array section with no scalar subscripts. The macro *PROCESS_SUBSCRIPTS* will be defined here in a tail-recursive way. The intention is that it should be expanded to a compile-time loop over s.

Let R be the rank of the subscripted array. If $s = R$, then the macro

$$PROCESS\_SUBSCRIPTS(v, r, a, s) \equiv \emptyset$$

Otherwise, if $subs_s$ is the degenerate triplet, :, then

```
$PROCESS\_SUBSCRIPTS(v, r, a, s) \equiv$
        $\mathbf{T}_r[v]$ = $\mathbf{T}_s[a]$ ;
        $PROCESS\_SUBSCRIPTS(v, r+1, a, s+1)$
```

Otherwise, if $subs_s$ is the triplet, $e_{\mathrm{lo}}:e_{\mathrm{hi}}:e_{\mathrm{stp}}$, and the sth dimension of a is distributed, then

```
$PROCESS\_SUBSCRIPTS(v, r, a, s) \equiv$
    $\mathbf{T}_r[v]$ = $a'_s$.range.subrng($e'_{\mathrm{lo}}$, $e'_{\mathrm{hi}}$, $e'_{\mathrm{stp}}$).arrayDim($a'_s$.stride) ;
    $PROCESS\_SUBSCRIPTS(v, r+1, a, s+1)$
```

where $a'_s = \mathbf{T}_s[a]$, $e'_{\mathrm{lo}} = \mathbf{T}[e_{\mathrm{lo}}]$, $e'_{\mathrm{hi}} = \mathbf{T}[e_{\mathrm{hi}}]$, and $e'_{\mathrm{stp}} = \mathbf{T}\left[e_{\mathrm{stp}}\right]$. Otherwise, if $subs_s$ is the triplet, $e_{\mathrm{lo}}:e_{\mathrm{hi}}:e_{\mathrm{stp}}$, and the sth dimension of a is sequential, then

```
$PROCESS\_SUBSCRIPTS(v, r, a, s) \equiv$
  $\mathbf{T}_r[v]$ = new SeqArrayDim($a'_s$.range.subrng($e'_{\mathrm{lo}}$, $e'_{\mathrm{hi}}$, $e'_{\mathrm{stp}}$), $a'_s$.stride) ;
  $PROCESS\_SUBSCRIPTS(v, r+1, a, s+1)$
```

with definitions as above. Two similar cases using the two-argument form of `subrng()` take care of triplets of the form $e_{\mathrm{lo}}:e_{\mathrm{hi}}$. Otherwise, if $subs_s$ is the splitting subscript, `<>`, then

```
$PROCESS\_SUBSCRIPTS(v, r, a, s) \equiv$
        Range $x$ = $a'_s$.range ;
        int   $u$ = $a'_s$.stride ;
        Range $z$ = $x$.shell() ;
        $\mathbf{T}_r[v]$ = new ArrayDim($x$.dim(), $u$ * $z$.volume()) ;
        $\mathbf{T}_{r+1}[v]$ = new SeqArrayDim($z$, $u$) ;
        $PROCESS\_SUBSCRIPTS(v, r+2, a, s+1)$
```

where x, u, and z are the names of new temporaries.

4.5.7.2 Integer subscripts in sequential dimensions

To handle the case where integer subscripts may appear in sequential dimensions (Figure 4.16) we add one new case for the definition of the macro *PROCESS_SUBSCRIPTS*.

If $subs_s$ is an integer expression, and the sth dimension of a is sequential, then

```
$PROCESS\_SUBSCRIPTS(v, r, a, s) \equiv$
    $b$ += $OFFSET\_DIM(\mathbf{T}_s[a], subs_s)$ ;
    $PROCESS\_SUBSCRIPTS(v, r, a, s+1)$
```

where the macro *OFFSET_DIM* is defined in section 4.5.6.

SOURCE:

```
v = a [[subs_0, ..., subs_{R-1}]] ;
```

TRANSLATION:

```
int b = T_bas[a].base ;
Group p = T_bas[a].group ;
PROCESS_SUBSCRIPTS(v, 0, a, 0)
if(p != null)
    T_dat[v] = p.amMember() ? T_dat[a] : null ;
else
    T_dat[v] = T_dat[a] ;
T_bas[v] = new ArrayBase(p, b) ;
```

where:

> The expression v is the assigned array variable in the source program,
> the simple expression a is the subscripted array in the source program,
> each term $subs_s$ is either an integer, a distributed index, a shifted index, a triplet, or <>,
> b and p are the names of new temporaries, and
> the macro *PROCESS_SUBSCRIPTS* is defined in the text.

Figure 4.16. Translation of array section without any scalar subscripts in *distributed* dimensions.

4.5.7.3 Scalar subscripts in distributed dimensions

The scheme for translating array sections when scalar subscripts appear in some distributed dimension is illustrated in Figure 4.17.

We add two new cases for the definition of the macro *PROCESS_SUBSCRIPTS*. If $subs_s$ is the integer expression n, and the sth dimension of a is distributed, then

```
PROCESS_SUBSCRIPTS(v, r, a, s) ≡
    Range x = a'_s.range ;
    Location l = x.location(n') ;
    b += l.sub * a'_s.stride ;
    if(p != null) p = p.restrict(x.dim(), l.crd) ;
    PROCESS_SUBSCRIPTS(v, r, a, s + 1)
```

where x and l are the names of new temporaries, $a'_s = \mathbf{T}_s[a]$, and $n' = \mathbf{T}[n]$. Otherwise, if $subs_s$ is a distributed index i or a shifted index $i \pm d$, then

SOURCE:

$$v \texttt{ = } a \texttt{ [[}subs_0\texttt{, ..., }subs_{R-1}\texttt{]] ;}$$

TRANSLATION:

b `=` $\mathbf{T}_{\text{bas}}[a]$`.base ;`
p `= (Group)` $\mathbf{T}_{\text{bas}}[a]$`.group.clone() ;`
$PROCESS\_SUBSCRIPTS(v, 0, a, 0)$
$\mathbf{T}_{\text{dat}}[v]$ `=` p`.amMember() ?` $\mathbf{T}_{\text{dat}}[a]$ `: null ;`
$\mathbf{T}_{\text{bas}}[v]$ `= new ArrayBase(`p`,` b`) ;`

where:

The expression v is the assigned array variable in the source program,
the simple expression a is the subscripted array in the source program,
each term $subs_s$ is either an integer, a triplet, or `<>`,
b and p are the names of new temporaries, and
the macro *PROCESS_SUBSCRIPTS* is defined in the text.

Figure 4.17. Translation of array section allowing scalar subscripts in distributed dimensions.

$PROCESS\_SUBSCRIPTS(v, r, a, s) \equiv$
  b `+=` $OFFSET\_DIM(a'_s, subs_s)$ `;`
  `if(`p `!= null)` p `=` p`.restrict(`dim`) ;`
  $PROCESS\_SUBSCRIPTS(v, r, a, s+1)$

where the macro *OFFSET_DIM* is defined in section 4.5.6 and *dim* is the dimension associated with i.

4.6 Test Suites

HPJava 1.0 is fully operational and can be downloaded from `www.hpjava.org` [23]. The system fully supports the Java Language Specification (JLS) [26], and has been tested and debugged against the HPJava test suites and *jacks* [24], an Automated Compiler Killing Suite. The current score is comparable to that of Sun JDK 1.4 and IBM Developer Kit 1.3.1. This means that the HPJava front-end is very conformant with Java. The HPJava test suites includes simple HPJava programs, and complex scientific algorithms and applications such as a multigrid solver, adapted from an existing Fortran program (called PDE2), taken from the Genesis parallel benchmark suite [1]. The whole of this program has been ported to HPJava (it is about 800 lines). Also, Computational Fluid Dynamics research application for fluid flow problems, (CFD)[3] has been ported to HPJava (it is about 1300 lines). An applet version of this application can be viewed at `www.hpjava.org`.

4.7 Discussion

We have reviewed the compilation strategies the current HPJava adopts in this chapter. For multiarray types and HPspmd classes, we have explained in detail the philosophy of design for multiarray types as well as the programming restrictions on HPspmd classes.

We have seen the whole picture of the current HPJava type-checking system with some explanations for the functionality of each phase such as type analysis, reachability nalysis, definite (un)assignment analysis. The current HPJava type-checker is totally based on the specification from "The Java Language Specification" [26]. Moreover, we reviewed the rules and restriction of the pre-translation phase. Through the isolation of the pre-translator from the translator itself, we have seen that the translator phase becomes simpler and more efficient.

The main issue of this chapter is to create logically well-defined and well-designed "basic" translation schemes for the HPJava system. We have reviewed the basic translation schemes we adopted to the current system. Because of the simplicity of the current HPJava translator, we need to keep researching to create better translation schemes for the system. Moreover, the naivety leads our research to a new direction; optimization. With simple and efficient translation schemes and effective optimization we will see in the next chapter, we can say the future of the HPJava system will be promising.

[3]The program simulates a 2-D inviscid flow through an axisymmetric nozzle. The simulation yields contour plots of all flow variables, including velocity components, pressure, mach number, density and entropy, and temperature. The plots show the location of any shock wave that would reside in the nozzle. Also, the code finds the steady state solution to the 2-D Euler equations.

CHAPTER 5

OPTIMIZATION STRATEGIES FOR HPJAVA

In the previous chapters, we have reviewed the HPspmd programming model, the HPJava language, and its compilation strategies. In the section 4.5, we have seen the basic translation scheme for the current HPJava compiler. We will sometimes call the basic translation scheme, *naive translation*. That is, any optimizations are not yet applied. Thus in this chapter we will introduce some optimization strategies suitable for speeding up the current naive translation of HPJava.

Based on observations for parallel algorithms such as direct matrix multiplication from Figure 3.13, Laplace equation using red-black relaxation from Figure 3.11, Laplace equation using Jacobi relaxation from Figure 3.7, etc, the majority of multiarray element accesses are generally located inside innermost `overall` loops. The main target of our optimization strategies is the complexity of the associated terms in the subscript expression of a multiarray element access, and the cost of HPJava compiler-generated method calls (`localBlock()`, etc). The following optimization strategies should remove overheads of the naive translation scheme (especially for the `overall` construct), and speed up HPJava.

5.1 Partial Redundancy Elimination

Partial Redundancy Elimination (PRE) is a global, powerful optimization technique first developed by Morel and Renvoise [34] and discussed in [27, 6]. PRE generally results in removing partially redundant expressions, and hoisting loop-invariant expressions. Unlike many other optimization algorithms, PRE doesn't need to distinguish between global and local optimizations. The same algorithm can handle both global and local versions of an optimization simultaneously. Moreover, PRE never lengthens an execution path.

Here, we need to review PRE a bit more deeply in order to know what features of PRE are suitable to our optimization plans. PRE eliminates redundant computations of expressions

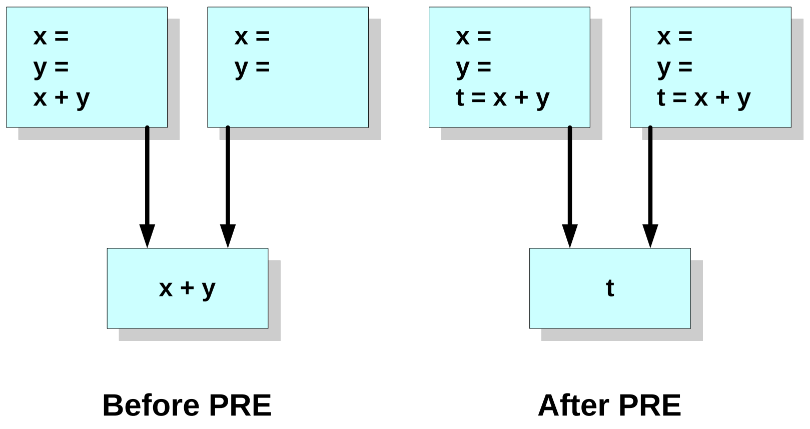

Figure 5.1. Partial Redundancy Elimination

that do not necessarily occur on all control flow paths leading to a given redundant computation. It combines two other techniques: *common subexpression elimination* and *loop-invariant code motion*. We need a couple of definitions before discussing these two.

Definition 3 An expression, `e` is *redundant* at some point `p` if and only if it is computed along every path leading to `p` and none of its constituent subexpressions has been redefined.

Definition 4 An expression, `e` is *loop-invariant* if it is computed inside a loop and its value is identical in all the iterations of the loop.

Common subexpression elimination says that if an expression `e` is redundant at `p`, then the evaluation of `e` at `p` can be replaced with a reference. Loop-invariant code motion says that if an expression `e` is loop-invariant in a loop, then it can be computed before the loop and referenced, rather than evaluated inside the loop.

Definition 5 An expression `e` is *partially redundant* at a point `p` if it is redundant along some, but not all, paths that reach `p`.

PRE is a complicated algorithm to understand and to implement, compared with other well-known optimization alogithms. However, the basic idea of PRE is simple. Basically, PRE converts partially redundant expressions into redundant expressions. The key steps of PRE are:

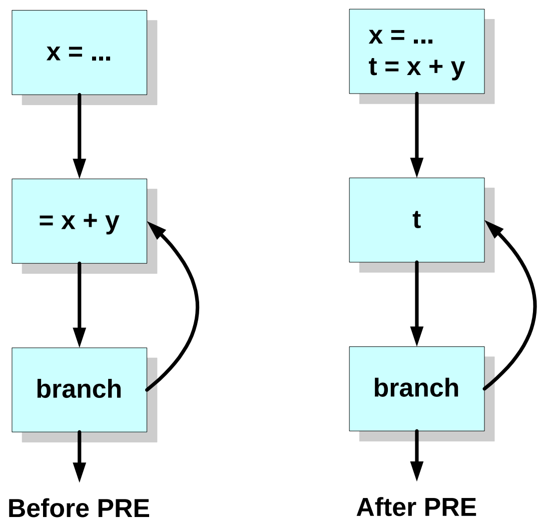

Figure 5.2. Simple example of Partial Redundancy of loop-invariant

Step 1: *Discover where expressions are partially redundant using data-flow analysis.*

Step 2: *Solve a data-flow problem that shows where inserting copies of a computation would convert a partial redundancy into a full redundancy.*

Step 3: *Insert the appropriate code and delete the redundant copy of the expression.*

The PRE we plan to apply to the HPJava System is originally described in [27]. It is a PRE using *Static Single Assignment* (SSA) form [15], called `SSAPRE`.

In Figure 5.1, the expression $x + y$ is redundant along the left-hand control flow path, but not the right. PRE inserts a computation of $x + y$ along the right-hand path. Thus, the computation of $x + y$ at the merge point is fully redundant and can be replaced with a temporary variable. Moreover, loop-invariant expressions are partially redundant as seen in Figure 5.2. On the left, $x+y$ is partially redundant since it is available from one predecessor (along the backedge of the loop), but not the other. Inserting an evaluation of $x + y$ before the loop allows it to be eliminated from the loop body.

We have reviewed the key features of PRE and its algorithm briefly. Since PRE is a global and powerful optimization algorithm to eliminate partial redundancy, (especially to get rid of loop-invariants generated by the current HPJava translator), we expect the optimized code

```
    s = 0                              s = 0
    i = 0                              k = a
L1: if i < n goto L2                   b = n * 4
    j = i * 4                          c = a + b
    k = j + d                      L1: if k < c goto L2
    x = M[k]                           x = M[k]
    s = s + x                          s = s + x
    i = i + 1                          k = k + 4
    goto L1                            goto L1

L2                                 L2

    (a) Before                         (b) After
```

Figure 5.3. Before and After Strength Reduction

to be very competitive and faster than the naively translated one. We will see experimental studies, applying PRE to some HPJava applications in section 5.5.

5.2 Strength Reduction

Strength Reduction (SR) [2] replaces expensive operations by equivalent cheaper ones from the target machine language. Some machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For instance, x^2 is invariably cheaper to implement as $x * x$ than as a call to an exponentiation routine. Moreover, an additive operator is generally cheaper than a multiplicative operator.

Some loops have a variable i that is incremented or decremented, and a variable j that is set in the loop to $i * c + d$ where c and d are loop-invariant. Then we can calculate j's value without reference to i; whenever i is incremented by a we can increment j by $c * a$

The program (a) from Figure 5.3 is a program to sum the elements of an array. Using *induction variable analysis* to find that i and j are related induction variables, *strength reduction* to replace a multiplication by 4 with an addition, then *induction variable elimination to replace* $i < n$ by $k < 4 * n + a$, we get the program (b).

We have reviewed the key features of SR and its algorithm briefly. SR is a very powerful algorithm to transform computations involving multiplications and additions together into computations only using additions. Thus, it could be a very competitive candidate to

SOURCE:

```
overall (i = x for e_lo : e_hi : e_stp) S
```

TRANSLATION:

```
Block b =  x.localBlock(T[e_lo], T[e_hi], T[e_stp]) ;

int shf = x.str() ;

Dimension dim = x.dim() ;
final APGGroup p = apg.restrict(dim) ;

int sub = b.sub_bas ;
int glb = b.glb_bas ;

for (int l = 0 ; l < b.count ; l++) {

    T[S |p]

    sub += b.sub_stp ;
    glb += b.glb_stp ;
}
```

where:

i is an index name in the source program,
x is a simple expression in the source program,
e_{lo}, e_{hi}, and e_{stp} are expressions in the source,
S is a statement in the source program, and
b, *shf*, *dim*, p, l, *sub* and *glb* are names of new variables.

Figure 5.4. Translation of `overall` construct, applying Strength Reduction.

optimize the current HPJava compiler because of the complexity of the associated terms in the subscript expression of a multiarray element access.

Before applying PRE and SR to some scientific algorithms, we can apply this SR technique on the basic translation of `overall` construct in the subsection 4.5.5. The naive translation generates a local subscript, *sub*, and a global subscript, *glb* for the index i. We call these two variables, *control subscripts* for the index i. The computations for the control subscripts are obviously amenable to SR. Thus, Figure 5.4 shows the translation scheme for `overall` after applying Strength Reduction.

5.3 Dead Code Elimination

Dead Code Elimination (DCE) [2] is an optimization technique to eliminate some variables not used at all. A variable is *useless* in a program P if it is dead at all exits from P, and its only use is in a definition of itself. All definitions of a useless variable may be eliminated.

Some instructions might have implicit side effects with carelessly applying DCE for high-level languages such as Java. For example,

```
int a = foo() ;
```

and assume that a only appears in this definition. That is, a is useless and may be deleted. But, in general, we can't know what happens inside the method call, `foo()`.

An `overall` construct generates 6 variables outside the loop according to the naive translation scheme. The first 4 variables are called *control variables* for `overall` construct. These variables are initialized by methods such as `localBlock()`, `str()`, `dim()`, and `restrict()`. Since these control variables are often unused, and the methods are specialized methods known to the compiler—side effect free—we don't have any side effects from applying DCE with data flow analysis to them.

The next 2 variables are control subscripts. They are useless if the `overall` has neither integer global index values (e.g. i') nor shifted indices. The compiler can easily know the uselessness of the control subscripts through applying data flow analysis, and can eliminate them without any side effects.

We assume that DCE should be applied after all optimization techniques we discussed earlier. Moreover, we assume we narrow the target of DCE for the HPJava optimization strategy. That is, for the moment, DCE will target only control variables and control scripts for `overall` constructs.

5.4 Loop Unrolling

In conventional programming languages some loops have such a small body that most of the time is spent to increment the loop-counter variables and to test the loop-exit condition.

We can make these loops more efficient by *unrolling* them, putting two or more copies of the loop body in a row. We call this technique, *Loop Unrolling* (LU) [2].

Our experiments [29] didn't suggest that this is a very useful source-level optimization for HPJava, except in one special but important case. LU will be applied for `overall` constructs in an HPJava Program of Laplace equation using red-black relaxation. For example,

```
overall (i = x for 1 + (i` + iter) % 2 : N-2 : 2) { ... }
```

This `overall` generates a relatively expensive method call in the translation:

```
x.localBlock(1 + (__$t + iter) % 2, N-2, 2) ;
```

The `overall` is repeated in every iteration of the loop associated with the outer `overall`. This will be discussed again in section 5.5.2. Because red-black iteration is a common pattern, HPJava compiler should probably try to recognize this idiom. If a nested overall includes expressions of the form

```
(i` + expr) % 2
```

where `expr` is invariant with respect to the outer loop, this should be taken as an indication to unroll the outer loop by 2. The modulo expressions then become loop invariant and arguments of the call to `localBlock()`, the whole invocation is a candidate to be lifted out of the outer loop.

5.5 Applying PRE, SR, DCE, and LU

The main issue of the current HPJava optimization is to eliminate the complexity of the associated terms in the subscript expression of a multiarray element access, generally located in inner `overall` loops, and to lift out HPJava compiler generated expensive method calls to outer loops. Thus, in this section, we will apply PRE, Strength Reduction, DCE, and LU optimization strategies to some naively translated HPJava programs such as *direct matrix multiplication* from Figure 3.13 and Laplace equation using red-black relaxation from Figure 3.11.

```
double sum = 0 ;

for (int k = 0 ; k < N ; k ++) {
  sum += a__$DS [a__$bas.base  + a__$0.stride  * __$t47 +
                 a__$1.off_bas + a__$1.off_stp * k] *
         b__$SD [b__$bas.base + b__$0.off_bas + b__$0.off_stp * k +
                 b__$1.stride * __$t54] ;
}
```

Figure 5.5. The innermost for loop of naively translated direct matrix multiplication

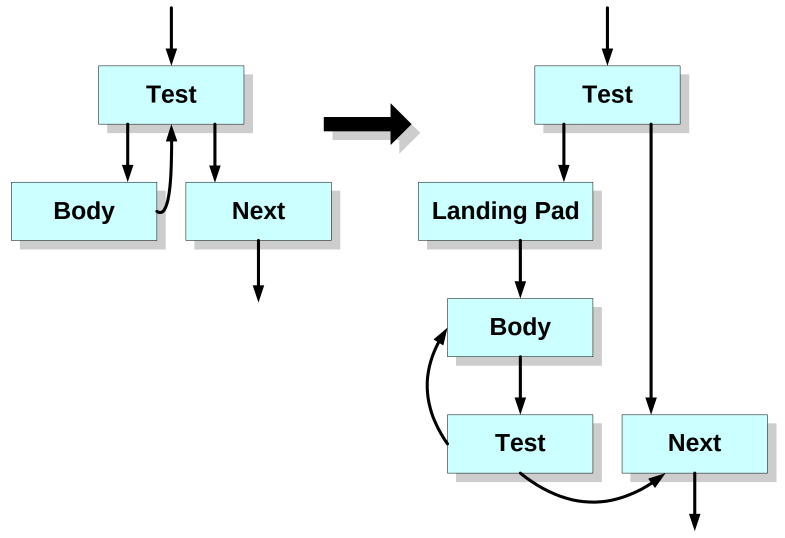

Figure 5.6. Inserting a landing pad to a loop

5.5.1 Case Study: Direct Matrix Multiplication

First, we start applying PRE and SR to direct matrix multiplication. Figure 5.5 shows the innermost `for` loop of the naively translated direct matrix multiplication program in HPJava, using the basic translation schemes of the section 4.5.

In the loop, we find complex subscript expressions of multiarray element accesses, involving final variables such as `a__$bas.base`, `a__$0.stride`, `a__$1.off_bas`, `a__$1.off_stp`, `b__$bas.base`, `b__$0.off_bas`, `b__$0.off_stp`, and `b__$1.stride`, `__$t47`, `__$t54`. We need to recall the main idea of PRE: eliminating redundant computations of expressions that

```
double sum = 0 ;
int k = 0 ;
if (k < N) {
    ///////////////////
    //  Landing Pad  //
    ///////////////////
    int a1 = a__$bas.base,  a2 = a__$0.stride, a3 = a__$1.off_bas,
        a4 = a__$1.off_stp, a5 = b__$bas.base, a6 = b__$0.off_bas,
        a7 = b__$0.off_stp, a8 = b__$1.stride ;

    int a9  = a1 + a2 * __$t47 + a3 ;
    int a10 = a5 + a6 ;
    int a11 = a8 * __$t54 ;
    ///////////////////
    do {
        sum += a__$DS [a9 + a4 * k] * b__$SD [a10 + a7 * k + a11] ;
        k ++ ;
    } while (k < N) ;
}
```

Figure 5.7. After applying PRE to direct matrix multiplication program

do not necessarily occur on all control flow paths that lead to a given redundant computation. So they variables do not necessarily occur on all control flow paths that lead to a given redundant computation since they are final (i.e. in general they are constant variables.). Thus, these variables can be replaced with temporary variables, declared outside the loop.

Before applying PRE, we need to introduce a *landing pad* [42]. To make PRE operative, we give each loop a landing pad representing entry to the loop from outside. Figure 5.6 shows how to insert a landing pad to a loop. Thus, all the newly introduced temporary variables by PRE, can be declared and initialized in the landing pad for the for loop. Figure 5.7 is the optimized code using PRE. Finally applying SR, which eliminates *induction variables* in the loop, the optimized HPJava program of direct matrix multiplication is shown in Figure 5.8.

Applying PRE and SR makes this simple HPJava program well-optimized. Moreover, we expect the optimized code to be more competitive and faster in performance than the naively translated one. In the next section, we will investigate more complicated examples.

```
double sum = 0 ;
int k = 0 ;
if (k < N) {
    ///////////////////
    //  Landing Pad  //
    ///////////////////
    int a1 = a__$bas.base,  a2 = a__$0.stride, a3 = a__$1.off_bas,
        a4 = a__$1.off_stp, a5 = b__$bas.base, a6 = b__$0.off_bas,
        a7 = b__$0.off_stp, a8 = b__$1.stride ;

    int a9  = a1 + a2 * __$t47 + a3 ;
    int a10 = a5 + a6 ;
    int a11 = a8 * __$t54 ;
    ///////////////////
    int aa1 = a9, aa2 = a10 + a11 ;
    ///////////////////
    do {
        sum += a__$DS [aa1] * b__$SD [aa2] ;
        k ++ ;
        ///////////////////
        aa1 += a4 ; aa2 += a7 ;
        ///////////////////
    } while (k < N) ;
}
```

Figure 5.8. Optimized HPJava program of direct matrix multiplication program by PRE and SR

5.5.2 Case Study: Laplace Equation Using Red-Black Relaxation

Next, we apply PRE, SR, and DCE to Laplace equation using red-black relaxation from Figure 3.11. Figure 5.9 shows the innermost `overall` loop of the naively translated direct matrix multiplication program in HPJava, using the new translation schemes from Figure 5.4. Here, let's focus on the control variables. Suppose that there is a nested `overall` loop. Then, after the translation, control variables generated by the inner loop reside inside the outer loop. That is, these expensive method calls are repeatedly called in every iteration of the outer loop. Since in the real life HPJava is aimed at large scale scientific and engineering applications, the number of iterations are likely to be large. Using information available to the compiler, moreover, we know in advance that the return values for `str()`, `dim()`, and

```
Block __$t42 = y.localBlock (1 + (__$t40 + iter) % 2, N - 2, 2) ;
int   __$t43 = y.str () ; Dimension __$t44 = y.dim () ;
APGGroup  __$t45 = __$t38.restrict (__$t44) ;

int __$t47 = __$t42.glb_bas, __$t48 = __$t42.sub_bas ;

for (int __$t46 = 0 ; __$t46 < __$t42.count ; __$t46 ++) {
    a__$DD [a__$bas.base + a__$0.stride * __$t41 +
                           a__$1.stride * __$t48] = (double) 0.25 *
        (a__$DD [a__$bas.base + a__$0.stride * (__$t41 - __$t36 * 1) +
                                a__$1.stride * __$t48] +
         a__$DD [a__$bas.base + a__$0.stride * (__$t41 + __$t36 * 1) +
                                a__$1.stride * __$t48] +
         a__$DD [a__$bas.base + a__$0.stride * __$t41 +
                                a__$1.stride * (__$t48 - __$t43 * 1)] +
         a__$DD [a__$bas.base + a__$0.stride * __$t41 +
                                a__$1.stride * (__$t48 + __$t43 * 1)]) ;

    __$t47 += __$t42.glb_stp ; __$t48 += __$t42.sub_stp ;
}
```

Figure 5.9. The innermost overall loop of naively translated Laplace equation using red-black relaxation

`restrict()` are *not* changed no matter what computations occur in the inner loop, except for `localBlock()`. This means that these control variables are partially redundant for the outer `overall`. Moreover, `__$t44` and `__$t45` are absolutely useless because they are not used anywhere, and `__$t47` is useless because it is only used in an useless statement, `__$t47 += __$t42.glb_stp`.

LU is a quite useful optimization technique for this experimental study since the innermost `overall` looks as follows:

```
overall (j = y for 1 + (i‘ + iter) % 2 : N-2 : 2) { ... }
```

which generates an expensive method call in the translation:

```
y.localBlock(1 + (__$t40 + iter) % 2, N-2, 2) ;
```

where `iter` is invariant with respect to the outer loop. This means that the outer loop can be unrolled by 2. Then, the arguments of `localBlock()` become loop invariant, and the whole invocation is a candidate to be hoisted outside the outer loop. The following simplified code is the optimized code by applying LU to the outer loop:

```
for(int __$t39 = 0; __$t39 < __$t35.count; __$t39 += 2) {
  Block __$t42_1 = y.localBlock(1 + (__$t40 + iter) % 2, N-2, 2) ;
  ...
  for (int __$t46 = 0; __$t46 < __$t42.count; __$t46 ++) { ... }

  Block __$t42_2 = y.localBlock(1 + (__$t40 + iter + __$t35.glb_stp) % 2, N-2, 2) ;
  ...
  for (int __$t46 = 0; __$t46 < __$t42.count; __$t46 ++) { ... }
  __$t40 += 2 * __$t35.glb_stp ;
}
```

Because `(__$t40 + iter) % 2` does not change when `__$t40` is incremented by a multiple of 2, this expression (for example) is loop-invariant. Now both invocations of `localBlock()` becomes invariant, and are ready to be hoisted outside the outer loop.

To eliminate complicated distributed index subscript expressions and to hoist control variables in the inner loops, we will adopt the following algorithm;

Step 1: *(Optional) Apply Loop Unrolling.*

Step 2: *Hoist control variables to the outermost loop by using compiler information if loop invariant.*

Step 3: *Apply Partial Redundancy Elimination and Strength Reduction.*

Step 4: *Apply Dead Code Elimination.*

We will call this algorithm *HPJOPT2* (**HPJ**ava **OPT**imization Level **2**)[1] . Applying LU is optional since it is only useful when a nested `overall` loop involves the pattern, `(i‘ + expr) % 2`. We don't treat Step 3 of HPJOPT2 as a part of PRE. It is feasible for control variables and control subscripts to be hoisted by applying PRE. But using information available to the compiler, we often know in advance they are loop invariant without applying PRE. Thus, without requiring PRE, the compiler hoists them if they are loop invariant.

Figure 5.10 shows a complete example of the optimized Laplace equation using red-black relaxation by HPJOPT2. All control variables are successfully lifted out of the outer loop, and complicated distributed index subscript expressions are well optimized.

[1]In later benchmarks we will take PRE alone as our "Level 1" optimization.

```
// Hoisted variables
Block __$t35 = x.localBlock (1, N - 2) ; int __$t36 = x.str () ;
int __$t40 = __$t35.glb_bas, __$t41 = __$t35.sub_bas ;
int __$t43 = y.str () ;
int glb_iter = __$t35.glb_bas + iter ;
Block __$t42_1 = y.localBlock (1 + (glb_iter), N - 2, 2) ;
Block __$t42_2 = y.localBlock (1 + (glb_iter + __$t35.glb_stp), N - 2, 2) ;
int a3 = a__$bas.base, a4 = a__$0.stride, a5 = a__$1.stride ;
int a6 = a3 + a4 * __$t41, a7 = a4 * __$t36, a9 = a5 * __$t43 ;

// Main body of Laplace equation using red-black relaxation
int __$t39 = 0 ;
if (__$t39 < __$t35.count) {
  int b1 = __$t35.glb_stp, b2 = __$t35.sub_stp ;
  do {
    int __$t48 = __$t42.sub_bas ;
    int __$t46 = 0 ;
    if (__$t46 < __$t42.count) {
      int a1 = __$t42.glb_stp, a2 = __$t42.sub_stp, ;
      int a8 = a5 * __$t48, a10 = a6 + a8 ;
      do {
        a__$DD [a10] =
            (double) 0.25 * (a__$DD [a10 - a7] + a__$DD [a10 + a7] +
                             a__$DD [a10 - a9] + a__$DD [a10 + a9]) ;
        __$t48 += a2 ; __$t46 ++ ;
      } while (__$t46 < __$t42.count) ;
    }
    __$t48 += a2 ; __$t46 = 0 ;
    if (__$t46 < __$t42.count) {
      int a1 = __$t42.glb_stp, a2 = __$t42.sub_stp, ;
      int a8 = a5 * __$t48, a10 = a6 + a8 ;
      do {
        a__$DD [a10] =
            (double) 0.25 * (a__$DD [a10 - a7] + a__$DD [a10 + a7] +
                             a__$DD [a10 - a9] + a__$DD [a10 + a9]) ;
        __$t48 += a2 ; __$t46 ++ ;
      } while (__$t46 < __$t42.count) ;
    }
    __$t40 += 2 * b1 ; __$t41 += 2 * b2 ; __$t39 += 2 ;
  } while (__$t39 < __$t35.count) ;
}
```

Figure 5.10. Optimized HPJava program of Laplace equation using red-black relaxation by HPJOPT2

In the mean time, we also need to observe what effect was produced on the subscript expression of a multiarray element access by PRE and SR. In the innermost `overall` loop, we have a multiarray element access in 3.11,

```
a [i - 1, j]
```

After the naive translation, in Figure 5.9, this access transforms into

```
a__$DD [a__$bas.base +
        a__$0.stride * (__$t41 - __$t36 * 1) + a__$1.stride * __$t48]
```

PRE knows that `__$t41`, and `__$t36` are constant values using data flow analysis. Thus, computations involving `__$t41`, and `__$t36` can be hoisted outside the loop where the multiarray element access resides. An interesting one is a variable, `__$t48`, being an induction variable candidate for SR. The multiplicative operation involving `__$t48` can be replaced with some cheaper additive operations. Thus, SR can affect a part of the subscript expression of this multiarray element access. Finally, this multiarray element access is optimized as follows;

```
a__$DD [a10 - a7]
```

Since most of the generated method calls for control variables are hoisted outside the loops, and expensive multiplicative operations are replaced with cheaper additive operations using PRE and Strength Reduction, we expect the optimized Laplace equation using red-black relaxation program to be faster in performance than the naive translation. This will be confirmed in the following two chapters

5.6 Discussion

We have reviewed some optimization schemes such as Partial Redundancy Elimination (PRE), Strength Reduction (SR), Dead Code Elimination (DCE), and Loop Unrolling (LU) algorithms to optimize the subscript expression of a multiarray element access and control variables for `overall` constructs.

PRE and SR are good candidates to make the natural complexity of the subscript expression of a multiarray element access simpler. Moreover, with some compiler information and the help of the unrolling technique, control variables generated by the naive translation for the `overall` construct can be hoisted ouside the outermost `overall` construct. Since the control variables are often dead, the Dead Code Elimination algorithm can be applied as well.

We have introduced the HPJava optimization strategy, *HPJOPT2* we plan to apply. It hoists control variables to the outermost loop by using compiler information if partially redundant, applies Dead Code Elimination, and finally applies Partially Redundancy Elimination and Strength Reduction. HPJOPT2 originally targets the subscript expressions of a multiarray element access and control variables for `overall` constructs. However, it will also generally optimize complex expressions, eliminate some dead codes, and reduce expressions in strength in HPJava programs.

CHAPTER 6

BENCHMARKING HPJAVA, PART I: NODE PERFORMANCE

Now we have reviewed compilation and optimization strategies for HPJava, it is time to see the performance of the HPJava system.

Compared to competitors, the HPJava system has various advantages. For example, HPJava is totally implemented in Java. This means that once coded, it can be run anywhere. However, this could turn to a disadvantage of HPJava, since many applications are designed and implemented for specific systems in order to gain performance advantage. Even though JDK 1.4 introduced performance features[1], do we expect Java to be the most people's favorite in terms of performance?

One of the main goals of this dissertation is to benchmark the HPJava system, and to prove its promising performance. In this chapter, we will benchmark scientific and engineering applications (e.g. a direct matrix multiplication program, and partial differential equations, such as the Laplace equation using red-black relaxation and the 3-dimensional diffusion equation) and a local dependency index application, called Q3 index, with a large set of data and iterations.

In this chapter, for each benchmark program, we will benchmark on a Linux machine. The performance of HPJava on a single processor node is as critical as the performance on multi-processors. Moreover, benchmarking sequential Java programs is as important as benchmarking HPJava programs because we can't ignore the close relation of HPJava with Java. We will be particularly interested in the effectiveness of the optimization schemes introduced in the last chapter. Typically we experiment with two different optimization levels described there: simple PRE and the augmented HPJOPT2 scheme.

The benchmarks in this chapter will be performed on the following machine:

[1]Such as the new IO package, called `java.nio`.

Table 6.1. Compilers and optimization options lists used on Linux machine.

HPJava	Java	C	Fortran
IBM Developer kit 1.4 (JIT) with -O	IBM Developer kit 1.4 (JIT) with -O	gcc -O5	g77 -O5

- **Linux Machine** – Red Hat 7.3 on PentiumIV 1.5 GHz CPU with 512 MB memory and 256 KB physical cache.

In addition to HPJava and sequential Java programs, we also benchmark C and Fortran programs on each machine to compare the performance. The table 6.1 lists the compilers and optimization options used on each machine. This chapter is based on our earlier publication [29].

6.1 Importance of Node Performance

Our HPJava language incorporates some ideas from High Performance Fortran (HPF) [22], and also from lower level library-based SPMD approaches, including MPI. All principles of HPF are *not* adopted wholesale in HPJava. Instead we import just a subset of basic HPF language primitives into HPJava: in particular *distributed arrays*. Library-based SPMD programming has been very successful in its domain of applicability, and is well-established. We can assume fairly confidently that the lessons and successes of SPMD programming will carry over HPJava. What it is less obvious is that HPJava can provide acceptable performance at each computing node.

There are two reasons why HPJava node performance is uncertain. The first one is that the base language is Java. We believe that Java is a good choice for implementing our HPspmd model. But, due to finite development resources, we can only reasonably hope to use the available commercial Java Virtual Machines (JVMs) to run HPJava node code. HPJava is, for the moment, a source-to-source translator. Thus, HPJava node performance depends heavily upon the third party JVMs. Are they good enough?

The second reason is related to nature of the HPspmd model itself. The data-distribution directives of HPF are most effective if the distribution format of arrays ("block" distribution format, "cyclic" distribution and so on) is known at compile time. This extra static information contributes to the generation of efficient node code[2]. HPJava starts from a slightly different point of view. It is primarily intended as a framework for development of—and use of—libraries that operate on distributed data. Even though distributed arrays may be clearly distinguished from sequential arrays by type signatures, their *distribution format* is often *not* known in advance (at compile time). Can we still expect the kind of node performance possible when one does have detailed compile-time information about array distribution?

We begin to address these questions in this chapter with benchmarks for some applications mentioned earlier. These applications will be benchmarked, comparing the output of current HPJava compiler. In this chapter these applications are executed on a single processor. But—because of the way the translation scheme works—we are reasonably confident that the similar results will carry over to node code on multiple processors. In fact this will be confirmed in the next chapter.

6.2 Direct Matrix Multiplication

Matrix multiplication is one of the most basic mathematical and scientific algorithms. Several matrix multiplication algorithms have been described in HPJava, such as general matrix multiplication from the Figure 3.15, pipelined matrix multiplication from the HPJava manual [9], and direct matrix multiplication from the Figure 3.13.

The "direct" matrix multiplication algorithm is relatively easier and potentially more efficient since the operand arrays have carefully chosen replicated/collapsed distributions. Moreover, as illustrated in Figure 3.14, the rows of `a` are replicated in the process dimension associated with `y`. Similarly the columns of `b` are replicated in the dimension associated with `x`. Hence all arguments for the inner scalar product are already in place for the computation—no run-time communication is needed.

Figure 6.1 shows the performance of the direct matrix multiplication programs in `Mflops/sec` with the sizes of $50\times$ 50, 80×80, 100×100, 128×128, and 150×150 in HPJava, Java, and C on the Linux machine.

[2]It is true that HPF has *transcriptive mappings* that allow code to be developed when the distribution format is not known at compile time, but arguably these are an add-on to the basic language model rather than a central feature.

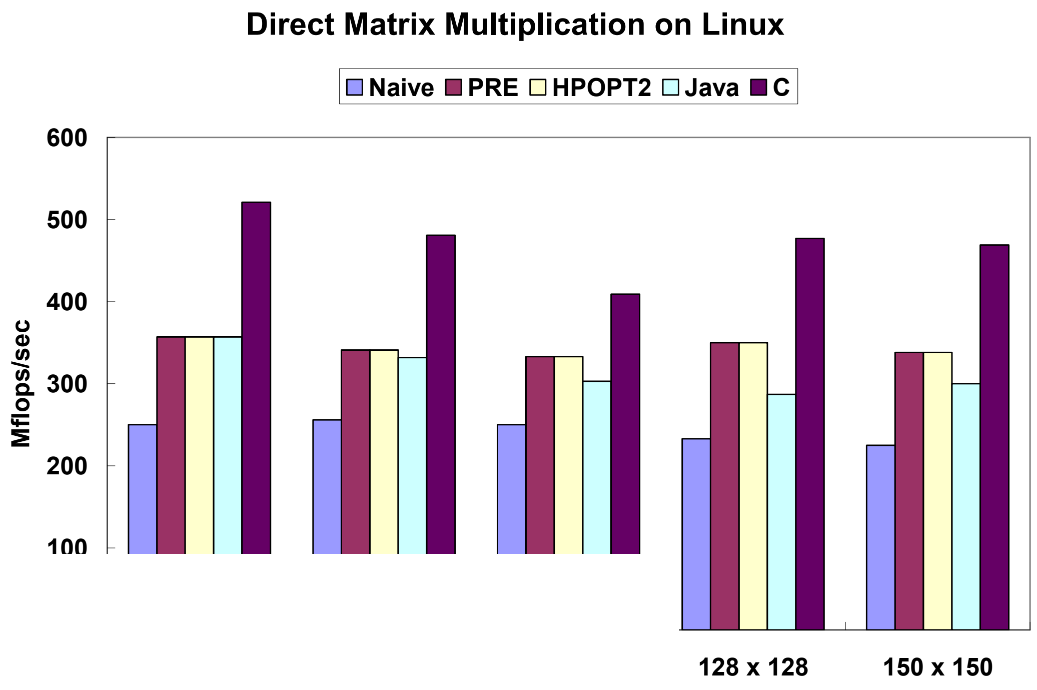

Figure 6.1. Performance for Direct Matrix Multiplication in HPJOPT2, PRE, Naive HPJava, Java, and C on the Linux machine

First, we need to see the Java performance from 6.1. The performance of Java over the C performance is up to 74%, and the average is 67%. This means that with a favorable choice of the matrix size, we can expect the Java implementation to be competitive with the C implementation. Secondly, we see the HPJava performance. The performance of naive translation over Java is up to 82%, and the average is 77%. It is quite acceptable. An interesting result we have expected is the PRE performance over the Java performance. It is up to 122%, and the average is 112%. HPJava with PRE optimization can get the same performance as Java or better on one processor. Thirdly, we need to see the speedup of PRE over the naive translation. It is up to 150%, and the average is 140%. The optimization gives dramatic speedup to the HPJava translation.

We can see that HPJOPT2 has no advantage over simple PRE in the Figure 6.1. The reason is the innermost loop for the direct matrix multiplication algorithm in HPJava is "for" loop, i.e. "sequential" loop, in the Figure 5.5. This means HPJOPT2 scheme has nothing to optimize (e.g. hoisting control variables to the outermost overall construct) for this algorithm.

6.3 Partial Differential Equation

Partial Differential Equations (PDEs) are at the heart of many computer analyses or simulations of continuous physical systems such as fluids, electromagnetic fields, the human body, etc [40].

Since the HPJava system aims at scientific and engineering computing applications, benchmarking PDEs is an important issue for the system. In this section we will briefly review the mathematical background of PDEs such as Laplace equations and diffusion equations, and will benchmark the PDE programs in HPJava.

6.3.1 Background on Partial Differential Equations

There are many phenomena in nature, which, even though occurring over finite regions of space and time, can be described in terms of properties that prevail at each point of space and time separately. This description originated with Newton, who with the aid of his differential calculus showed how to grasp a global phenomenon—for example, the elliptic orbit of a planet, by means of a locally applied law, for example $F = ma$.

This approach has been extended from the motion of single point particles to the behavior of other forms of matter and energy, including fluids, light, heat, electricity, signals traveling along optical fibers, neurons, and even gravitation. This extension consists of formulating or stating a partial differential equation governing the phenomenon, and then solving that differential equation for the purpose of predicting measurable properties of the phenomenon.

PDEs are classified into three categories, hyperbolic (e.g. wave) equations, parabolic (e.g. diffusion) equations, and elliptic (e.g. Laplace) equations, on the basis of their characteristics, or the curves of information propagation.

Suppose there exists an unknown function u of x. We can denote u's partial derivative with respect to x as follows;

$$u_x = \frac{\partial u}{\partial x} \ , \quad u_{xy} = \frac{\partial^2}{\partial x \partial y} \tag{6.1}$$

Then,

1. **Laplace Equation** – An equation for unknown function u, of x, y, and z, as follows;

$$u_{xx} + u_{yy} + u_{zz} = 0 \tag{6.2}$$

Solutions to this equation, known as harmonic functions, serve as the potentials of vector field in physics, such as the gravitational or electrostatic fields.

A generalization of Laplace's equation is Poisson's equation:

$$u_{xx} + u_{yy} + u_{zz} = f \tag{6.3}$$

where $f(x, y, z)$ is a given function. The solutions to this equation describe potentials of gravitational and electrostatic fields in the presence of masses or electrical charges, respectively.

2. **Diffusion Equation** – An equation describing the temperature in a given region over time for unknown function u, with respect to x, y, and z, as follows;

$$u_t = k(u_{xx} + u_{yy} + u_{zz}) \tag{6.4}$$

Solutions will typically "even out" over time. The number k describes the heat conductivity of the material.

3. **Wave Equation** – An equation for unknown function u, with respect to x, y, z, and t, where t is a time variable, as follows;

$$u_{tt} = c^2(u_{xx} + u_{yy} + u_{zz}) \tag{6.5}$$

Its solutions describe waves such as sound or light waves; c is a number which represents the speed of the wave. In lower dimensions, this equation describes the vibration of a string or drum. Solutions will typically be combinations of oscillating sine waves.

6.3.2 Laplace Equation Using Red-Black Relaxation

In this section, we will benchmark the well-known Laplace equation using red-black relaxation. Figure 3.11 is an implementation of the 'red-black' scheme, in which all even sites are stencil-updated, then all odd sites are stencil-updated in a separate phase.

There are two things we want to point out for PDE benchmark applications before we start: overhead of run-time communication library calls and relation between problem size and cache size.

First, an HPJava program such as Laplace equation using red-black relaxation from Figure 3.11 introduces a run-time communication library call, `Adlib.writeHalo()`, updating the cached values in the ghost regions with proper element values from neighboring processes. Because our concern here is with optimization of node code, not the communication library, we will ignore (delete) communication library calls when benchmarking node performance of HPJava through this chapter.

We do two experiments, one with all data in the cache, and the other not, and see what we learn from the experiments.

In the first experiment, in order to maximize performance, we will choose a right size of each matrix for this benchmark. When the size of a matrix on PDE applications is larger than the size of physical cache, the cache hit ratio will be decreased. Thus, we assume the we choose a proper size of matrix for Laplace equation using red-black relaxation to maximize the efficiency of code in the machine. Because the size of physical cache for the Linux machine is 256 KB and the data type is `double`, the proper size of matrix should be less than $\sqrt{256000/8} \cong 179$. So, we choose a 150×150 matrix for benchmarking an HPJava Laplace equation using red-black relaxation program.

In section 5.4, we discussed that a relatively expensive method call, `localBlock()`, can't be lifted out of the outer loop since it is not an invariant. We produced the loop unrolling (LU) technique to handle this problem. Then, is LU the best choice to make the method call an invariant and to hoist outside the outer loop?

Compared to LU, we introduce another programming technique, called *splitting*. This is a programming technique rather than a compiler optimization scheme. The main body of the HPJava Laplace equation using red-black relaxation from Figure 3.11 looks like:

```
overall(i = x for 1 : N - 2)
   overall(j = y for 1 + (i` + iter) % 2 : N - 2 : 2) { ... }
```

In the naive translation, the inner `overall` generates a method call:

```
x.localBlock(1 + (__$t + iter) % 2, N-2, 2) ;
```

which is not an invariant and can't be hoisted outside the outer loop. This problem can be solved if the index triplet of the inner `overall` is invariant with respect to the outer loop.

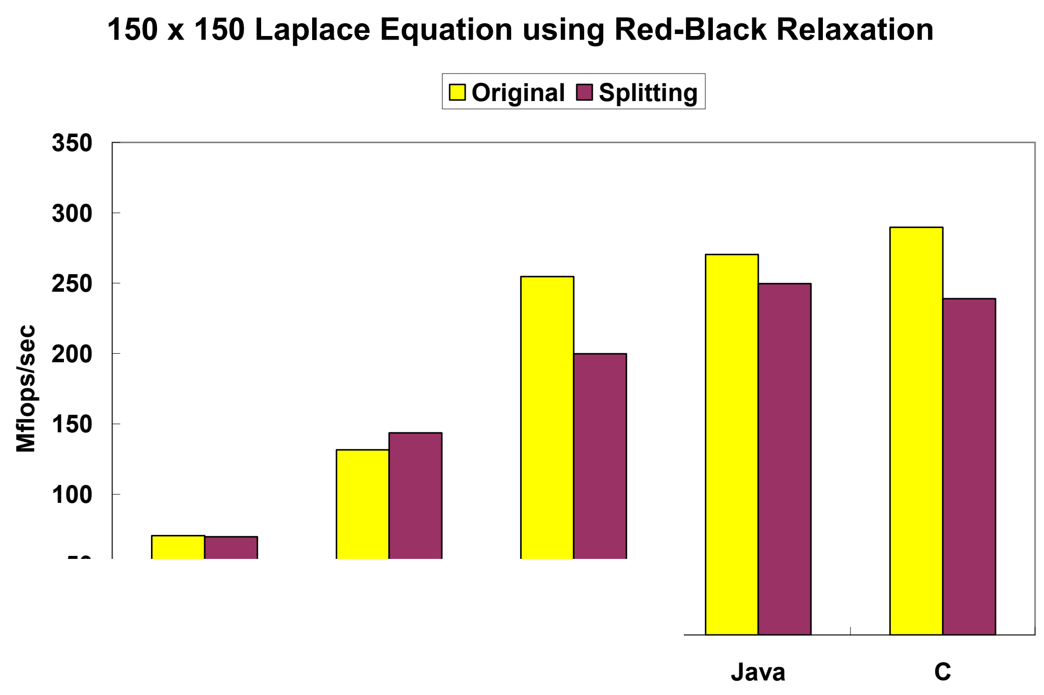

Figure 6.2. Performance comparison between 150 × 150 original Laplace equation from Figure 3.11 and split version of Laplace equation.

We can split the inner `overall` in the main body splitting each nested `overall` above into two nested `overall`, one for updating 'red' and the other for updating 'black' like:

```
// Updating 'red'
overall(i = x for 1 : N - 2)
   overall(j = y for 1 : N - 2 : 2) { ... }
overall(i = x for 2 : N - 2)
   overall(j = y for 2 : N - 2 : 2) { ... }
// Updating 'black'
overall(i = x for 1 : N - 2)
   overall(j = y for 2 : N - 2 : 2) { ... }
overall(i = x for 2 : N - 2)
   overall(j = y for 1 : N - 2 : 2) { ... }
```

Figure 6.2 shows the performance comparison of Laplace equation using red-black relaxation with the problem size of 150 × 150. The original Laplace equation using red-black relaxation is from Figure 3.11. It is optimized by HPJOPT2 with loop unrolling since the pattern, `(i` + expr) % 2` is detected. The split version is the newly introduced program to avoid the pattern, `(i` + expr) % 2` we can immediately hoist the heavy method call, `localBlock()`, outside the outer loop (without loop unrolling). Moreover, the original

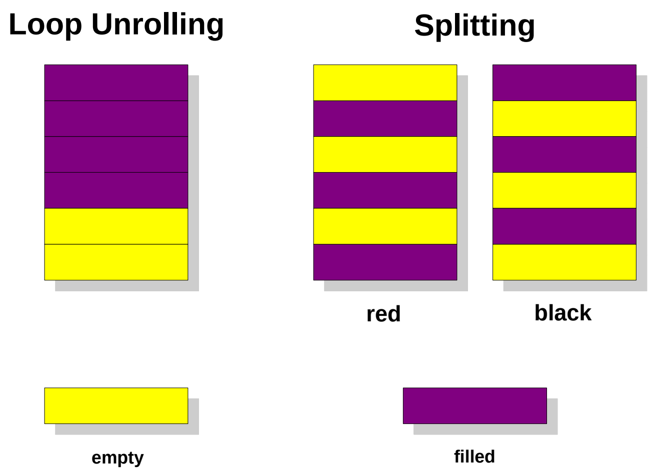

Figure 6.3. Memory locality for loop unrolling and split versions.

version of Java and C programs is not loop-unrolled. Rather, it is maximally optimized by `O` and `-O5` options, respectively.

The performance of naive translations for both programs are almost identical to each other and quite behind Java and C performance (about 30%). For the program only PRE applied, the split version is slightly better (10%) than the original. Both are still quite behind Java and C performance (about 49%). The program with only PRE applied has performance improvement over the naive translation. The original is improved by 186% and the split version is improved by 205%. Moreover, we got more dramatic performance improvement from HPJOPT2 over the naive translation. The original is improved by 361% as well as the split version is improved by 287%.

The interesting thing is the performance of programs where HPJOPT2 is applied. The speed of the original with HPJOPT2 (and loop-unrolled) is 127% over the split version with HPJOPT2 because of memory locality. Data for the original with HPJOPT2 are locally held for each inner loop, compared to data for the split version like Figure 6.3. This means that applying the HPJOPT2 compiler optimization (unrolling the outer loop by 2) is the better choice than programmatically splitting the `overall` nest.

One of the most interesting results is that the performance of the original with HPJOPT2 (and loop-unrolled) is almost identical with Java and C performance (the original is 6% and

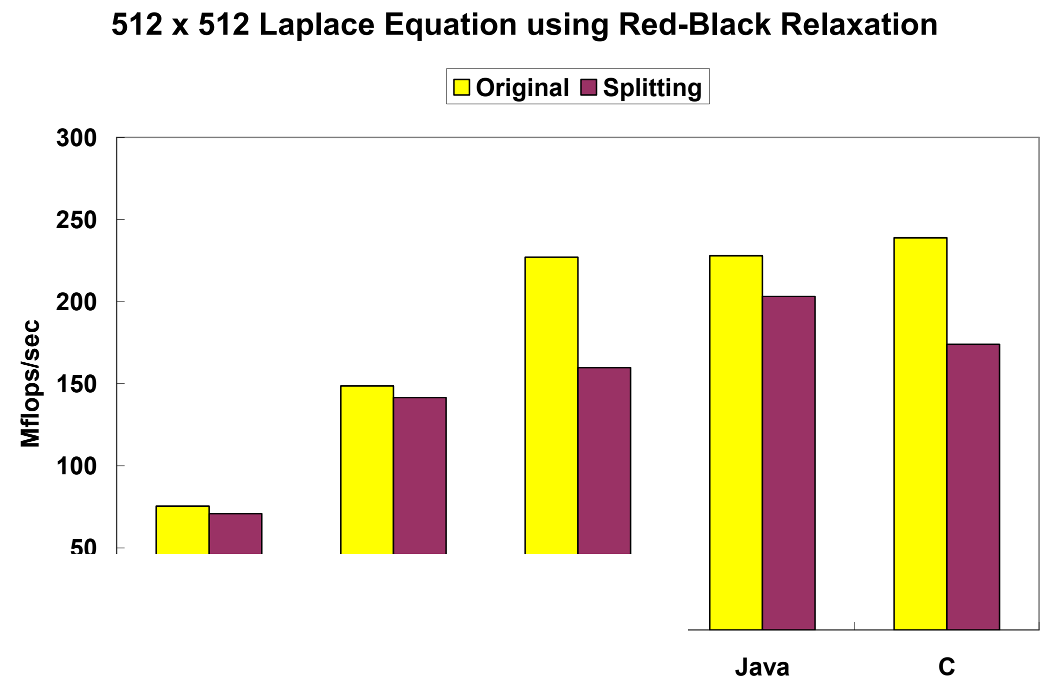

Figure 6.4. Performance comparison between 512 × 512 original Laplace equation from Figure 3.11 and split version of Laplace equation.

10% behind, respectively). That is, an HPJava node code of Laplace equation using red-black relaxation optimized by HPJOPT2 is quite competitive with Java and C implementations.

For our second experiment, Figure 6.4 shows the performance comparison of Laplace equation using red-black relaxation with the problem size of 512 × 512. The pattern of performance is virtually identical with that of the problem size of 150 × 150 except that if the version for all data in cache has 12% better performance than the large problem size.

6.3.3 3-Dimensional Diffusion Equation

Diffusion is an everyday experience; a hot cup of tea distributes its thermal energy or an uncorked perfume bottle distributes its scent throughout a room.

Figure 6.5 shows the performance comparison of the 3D diffusion equation with the sizes of 32 × 32 × 32, 64 × 64 × 64, 128 × 128 × 128 in HPJava, Java, and C on the Linux machine.

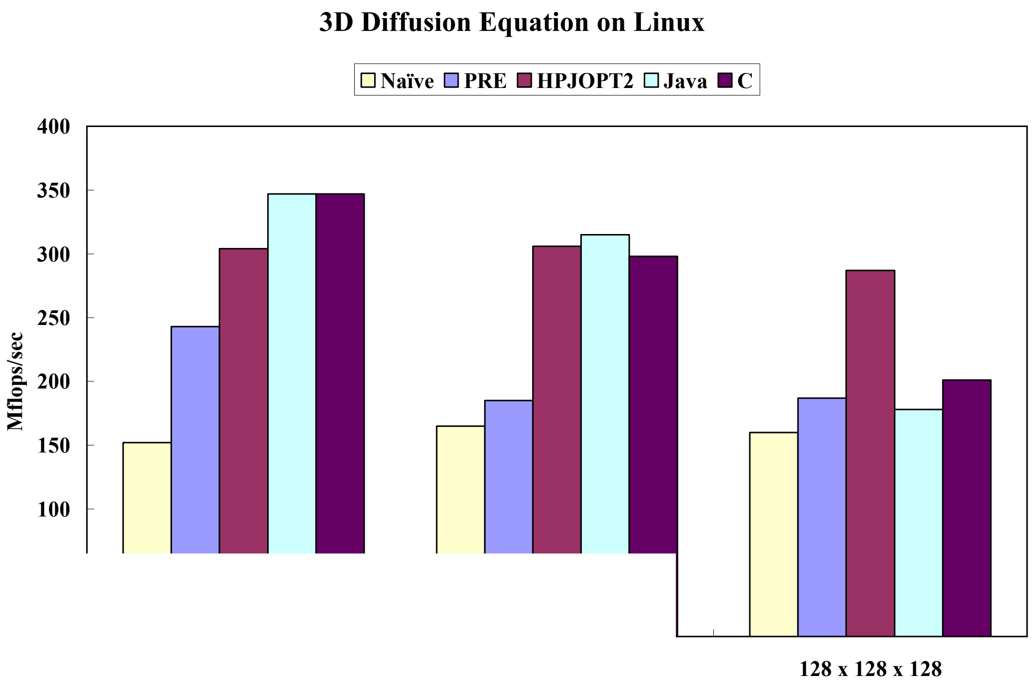

Figure 6.5. 3D Diffusion Equation in Naive HPJava, PRE, HPJOPT2, Java, and C on Linux machine

First, we need to see the Java performance from 6.5. The performance of Java over the C performance is up to 106%, and the average is 98%. This means that we can expect the Java implementation to be competitive with the C implementation.

The performance of naive translation over Java is up to 90%, and the average is 62%. We meet a situation very similar to the Laplace equation. That is, we expected the performance of the naively translated 3D diffusion equation to be poor because of the nature of naive translation schemes for overall construct and the subscript expression of a multiarray element access. The program with only PRE applied has performance improvement over the naive translation. It is improved by up to 160%, and the average is 130%. The program optimized by HPJOPT2 is improved by up to 200%, and the average is 188%. The HPJOPT2 optimization gives a dramatic speedup to the HPJava translation.

As expected, an HPJava program for the 3D diffusion equation maximally optimized by HPJOPT2 is very competitive with Java and C implementations. When the problem size is increased like 128 $\times$ 128 $\times$ 128, the HPJava program is tremendously better than Java and

C (161% and 141%). This is a very good result since the HPJava system aims at scientific and engineering computing applications with the large problem size.

6.4 Q3 – Local Dependence Index

The *Q3* index is an essential analysis for observing examinees' behaviour in terms of "ability" and "speed." The result of the index will be used to evaluate if the test is fair for all students without the level of their abilities. In this section, we will review an educational application, Q3 – local dependence index, and will implement and benchmark the application in HPJava.

6.4.1 Background on Q3

Test specialists need to assign item parameters to each item and an examinee parameter to each examinee so that they can probabilistically predict the response of any examinee to any item, even if similar examinees have never taken similar items before. For such a purpose, *Item Response Theory* (IRT) has been developed and has popularly used for item analysis, test construction, and test equating. Most commonly used 3-parameter logistic IRT model takes the mathematical form:

$$\mathbf{P_i}(\theta) = \mathbf{c_i} + (\mathbf{1} - \mathbf{c_i})\frac{\mathbf{e}^{\mathbf{1.7a_i}(\theta - \mathbf{b_i})}}{\mathbf{1} - \mathbf{e}^{\mathbf{1.7a_i}(\theta - \mathbf{b_i})}} \qquad (\mathbf{i} = \mathbf{1}, \mathbf{2}, \ldots, \mathbf{n}) \tag{6.6}$$

where a_i is the item discrimination parameter, b_i is the item difficulty parameter, c_i is the guessing parameter, and $P_i(\theta)$ is the probability that an examinee with ability level θ answers item i correctly.

When an IRT model is used to analyze the data, ignoring the violation of these assumptions, the estimated values of item parameters and the examinee parameter may not be accurate. Moreover, the major usage of the most standardized achievement test is selection, prediction and placement. When the use of a test results can affect the life chances or educational opportunities of examinees, The analysis of test data should be done carefully.

Therefore, Several methods for assessing local dependence in the test data have been developed. Yen [49] proposed that Q3 index is useful to detect local dependence. Q3 is the correlation of a pair of items with the trait estimate partialled out. Yen illustrated that the residual is calculated as

$$\mathbf{d_{ik}} = \mathbf{u_{ik}} - \hat{\mathbf{P}}_\mathbf{i}(\hat{\theta_\mathbf{k}}) \tag{6.7}$$

where u_{ik} is the score of the k^{th} examinee on the i^{th} item. $\hat{P}_i(\hat{\theta_k})$ can be estimated by equation 6.6. Then Q3 is the correlation of deviation scores across all examinees and can be expressed as

$$\mathbf{Q3_{ij}} = \mathbf{r_{d_i d_j}} \tag{6.8}$$

for item i and j, and it provides a pair-wise index of item dependence. If local independence is held between any pair of items, expected value of Q3 is supposed to be close to $\frac{-1}{(n-1)}$, where n is the number of items on the test [50]. In [12], Q3 index has successfully used to provide an evidence of local dependence.

6.4.2 Data Description

One English test form from a national standardized examination administered in 1999 was examined using Q3 index. The test consists of 5 passages and there are 15 items associated with each passage. The number of examinee who has taken the test is 2551. Since, items are nested in particular passage in the test, we cannot be certain that local independence assumption is held for this data. So the check-up procedure is needed such as calculating Q3 index. IRT item parameters, a_i, b_i, and c_i, and the examinee parameter, θ_k for 3-PL model of test data were estimated using computer software from [33]. Once the item parameters and the examinee parameter are estimated, $\hat{P}_i(\hat{\theta_k})$ can be estimated using equation 6.6. Then Q3 statistic can be calculated.

Thus, we have a 75×3 matrix for the item parameters, a_i, b_i, and c_i, called `PAR`, a 2551×1 matrix for the examinee parameter, θ_k, called `THT`, and a 2551×75 matrix for the answers of each student, called `RAW`.

6.4.3 Experimental Study – Q3

Q3 is a real world application with a large number of computations, rather than a scientific benchmarking program. Figure 6.6 is a Q3 implementation in HPJava. Here, rather than benchmarking the whole program, we need to know which parts dominates the

```
overall (i = x for : )                // Compute probability
  for (int j = 0; j<NUM_RESPONSES; j++) {
    double exp = Math.exp(1.7 * par[j, 0] * (tht[i] - par[j, 1])) ;
    prob[i, j] = par[j, 2] + (1.0 - par[j, 2]) * exp / (1.0 + exp) ;
  }

overall (i = x for : )                // Calculate the difference scores
  for (int j = 0; j<NUM_RESPONSES; j++)
    diff[i, j] = raw[i, j] - prob[i, j] ;

Adlib.sumDim(sum_diff, diff, 0) ;     // sum(di) / n
for (int i = 0; i<NUM_RESPONSES; i++)
  sum_diff[i] /= NUM_STUDENTS ;

overall (i = x for : )                // sum(power(di, 2)) / n
  for (int j = 0; j<NUM_RESPONSES; j++)
    power_diff_2[i, j] = diff[i, j] * diff[i, j] / NUM_STUDENTS ;
Adlib.sumDim(sum_power_diff_2, power_diff_2, 0) ;

overall (k = x for : )                // calculating didj
  for (int i = 0; i<NUM_RESPONSES; i++)
    for (int j = 0; j<NUM_RESPONSES; j++)
      if (i != j)
        didj[k, i*NUM_RESPONSES + j] = diff[k, i] * diff[k, j] / NUM_STUDENTS ;

Adlib.sumDim(sum_didj, didj, 0) ;     // sum(di * dj) / n

for (int i=0; i<NUM_RESPONSES; i++)   // covariance
  for (int j=0; j<NUM_RESPONSES; j++)
    if (i != j)
      cov[i][j] = sum_didj[i*NUM_RESPONSES + j] - sum_diff[i] * sum_diff[j] ;

for (int i=0; i<NUM_RESPONSES; i++)   // variance
  var[i] = sum_power_diff_2[i] - sum_diff[i] * sum_diff[i] ;

for (int i = 0; i<NUM_RESPONSES; i++) // Calculate Q3
  for (int j = 0; j<NUM_RESPONSES; j++)
    if (i != j) {
      q3[i][j] = cov[i][j] / Math.sqrt(var[i] * var[j]) ;
      avg += q3[i][j] ;
    }
```

Figure 6.6. Q3 Index Algorithm in HPJava

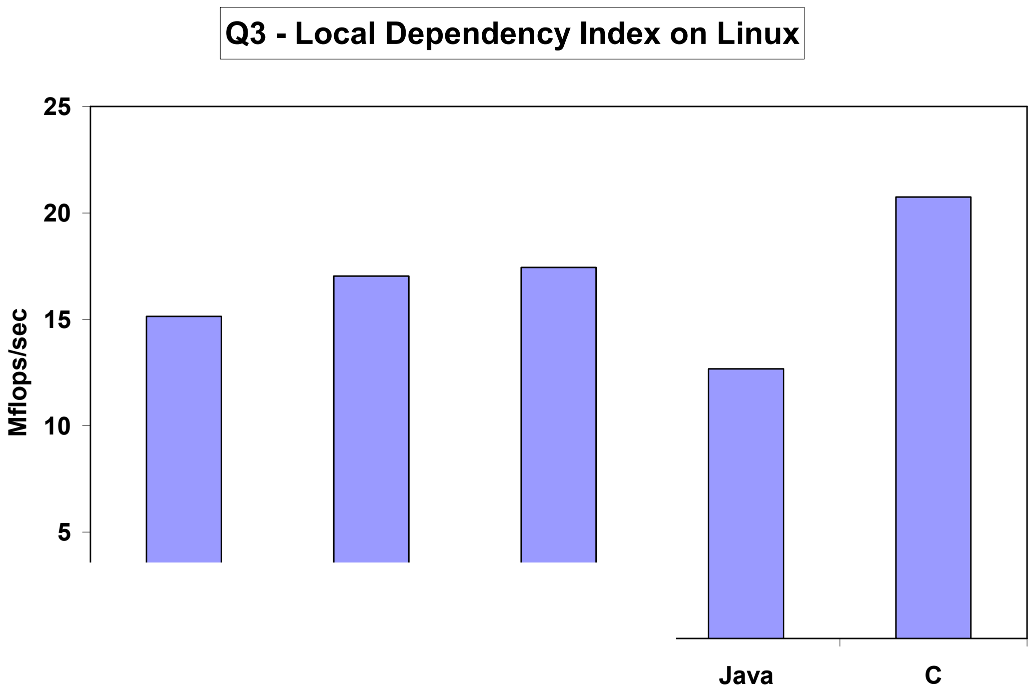

Figure 6.7. Performance for Q3 on Linux machine

whole algorithm. Obviously, Q3 application spend its most time in the overall construct for calculating difference of every pair of `diff` elements, and the summation of the difference. The kernel of Q3 dominates 97% of the whole program according to the time measuring.[3] Thus, we will benchmark only the kernel of Q3 algorithm.

Figure 6.7 shows the performance comparisons of Q3 in HPJava, Java, and C on the Linux machine.

The Java performance is 61% over the C performance. Here, what we have to focus on is the performance of HPJava over that of Java. The HPJava performance is 119% over the Java performance. This is the ideal result of HPJava performance we always have expected. The reason HPJava is faster than Java is that, in the HPJava implementation of Q3, in order to calculate $\frac{1}{n}\sum d_i d_j$ where $i = 1, \ldots, 2551$ and $j = 1, \ldots, 5625$, we use a run-time communication library, `Adlib.sumDim()` instead of loops used in the sequential Java implementation of Q3.

[3]In the first place, we intuitively expected the most dominant parts of Q3 might be the first overall construct for calculating probability since the construct has a Java math function call, `Math.exp(...)`.

In addition, the design decision leads the HPJava implementation of Q3 to an efficient implementation. To calculate $d_i * d_j$, we use nested for loops instead of overall constructs. If a matrix with a small size, such as 75 $\times$ 75, is distributed, the cost of the communications is higher than that of the computing time. When a multiarray is created, we generally suggest that a dimension of a small size (e.g. < 100) be a sequential dimension rather than distributed one to avoid unnecessary communication times.

The optimized codes by PRE and HPJOPT2 performs up to 113% and 115%, respectively, over the naive translation. Especially, the result of HPJOPT2 is up to 84% of the C performance. Moreover, the result of HPJOPT2 is up to 138% of the Java performance. That is, an HPJava program of Q3 index maximally optimized by HPJOPT2 is very competitive with C implementations.

As discussed in section 6.3.2, because the problem size of Q3 is too large to fit in the physical cache of the Linux machine, the net speed of Q3 is slower (17.5 Mflops/sec with HPJOPT2 optimization) than other applications ($\geq$ 250 Mflops/sec with HPJOPT2 optimization). However, the net speed of Java and C programs is slower than other applications as well. Even though Q3 is relatively slower than others on a single processor, we expect performance on multi-processors to be better because data will be divided, with the right choice of the number of processors, each part may fit in the cache of the parallel machines.

6.5 Discussion

In this chapter, we have experimented on and benchmarked the HPJava language with scientific and engineering applications on a Linux machine (Red Hat 7.3) with a single processor (Pentium IV 1.5 GHz CPU with 512 MB memory and 256 KB cache).

The main purpose we concentrated on benchmarking on the Linux machine in this chapter is to verify if the HPJava system and its optimization strategies produces efficient *node code*. Without confidence of producing efficient node code, there is no prospect of high-performance for the HPJava system on parallel machines. Moreover, through these benchmarks, we studied the behaviour of the overall construct and the subscript expression of a multiarray element access in HPJava programs and experimented with the effect of optimization strategies to the HPJava system.

Table 6.2. Speedup of each application over naive translation and sequential Java after applying HPJOPT2.

	Direct Matrix Multiplication	Laplace equation red-black relaxation	3D Diffusion	Q3
HPJOPT2 over naive translation	150%	361%	200%	115%
HPJOPT2 over sequential Java	122%	94%	161%	138%

Unlike direct matrix multiplication and Q3 index, the index triplets of overall constructs of Laplace equation using red-black relaxation and 3D diffusion equation in HPJava are not using the default value (e.g. `overall(x = i for :)`). If the index triplet depends on variables, then one of control variables, whose type is `localBlock()`, is not loop invariant. This means that it can't be hoisted outside the most outer overall construct. To eliminate this problem in a common case, we adopted a Loop Unrolling optimization in our HPJOPT2. Moreover, when creating multiarrays, all dimensions are distributed in these PDE examples. In contrast, some of dimensions in direct matrix multiplication and Q3 index are *sequential*. The translation scheme for the subscript expression of a distributed dimension is obviously more complicated than that of a sequential dimension.

As we see from table 6.2, HPJava with HPJOPT2 optimization can maximally increase performance of scientific and engineering applications with large problem size and more distributed dimensions. It proves that the HPJava system should be able to produce efficient node code and the potential performance of HPJava on multi-processors looks very promising.

CHAPTER 7

BENCHMARKING HPJAVA, PART II: PERFORMANCE ON PARALLEL MACHINES

In this chapter we will benchmark the HPJava programs introduced in chapter 6 on parallel machines. In chapter 6 we have seen that HPJava produces quite efficient node code, and also that our HPJOPT2 optimization scheme dramatically improves performance on a single processor. With successful benchmarks on one processor, we expect the HPJava system to achieve high-performance on parallel machines.

The main issue of this chapter is to observe performance of HPJava programs on the parallel machines. It won't be necessary to critically analyze behaviour and performance of HPJava programs on these machines. We are reasonably confident that good results for node code will carry over to multi-processors because of the way that translation and optimization works.

In addition to HPJava and sequential Java programs, we also benchmark C and Fortran programs on each machine to compare to the performance. For now the C and Fortran programs are sequential programs, not parallel programs.

To assess performance on multi-processors, the benchmarks will be performed on the following machines:

- **Shared Memory Machine** – Sun Solaris 9 with 8 UltraSPARC III Cu 900 MHz Processors and 16 GB of main memory.

- **Distributed Memory Machine** – IBM SP3 running with four Power3 375 MHz CPUs and 2 GB of memory on each node.

The table 7.1 lists the compilers and optimization options used on each machine. This chapter is based on our recent publications such as [31, 30]. The benchmarks on SP3 are from [32]. It experimented with only naively translated HPJava programs.

Table 7.1. Compilers and optimization options lists used on parallel machines.

	Shared Memory	Distributed Memory
HPJava	Sun JDK 1.4.1 (JIT) with -server -O	IBM Developer kit 1.3.1 (JIT) with -O
Java	Sun JDK 1.4.1 (JIT) with -server -O	IBM Developer kit 1.3.1 (JIT) with -O
C	gcc -O5	
Fortran		F90 -O5

Table 7.2. Speedup of the naive translation over sequential Java and C programs for the direct matrix multiplication on SMP.

Number of Processors	1	2	3	4	5	6	7	8
Naive translation over Java	0.39	0.76	1.15	1.54	1.92	2.31	2.67	3.10
PRE over Java	0.76	1.52	2.29	3.07	3.81	4.59	5.30	6.12
PRE over Naive translation	1.96	2.02	2.00	2.00	1.98	1.99	1.99	1.98

7.1 Direct Matrix Multiplication

Figure 7.1 shows the performance of the direct matrix multiplication on the shared memory machine. In the figure, Java and C indicate the performance of sequential Java and C programs on the shared memory machine. Naive and PRE indicates the parallel performance of the naive translation and PRE on the shared memory machine. Since we observed in section 6.2 that HPJOPT2 takes no advantage over PRE for this algorithm, we benchmark only naive translation, PRE, Java, and C programs in this section.

First, we need to see the Java performance over the C performance on the shared memory machine. It is 86%. With this Java performance, we can expect the performance of HPJava to be promising on the shared memory machine. The table 7.2 shows the speedup of the naive translation over sequential Java program. Moreover, it shows the speedup of PRE over the naive translation.

The speedup of the naive translation with 8 processors over sequential Java is up to 310%. The speedup of PRE with 8 processors over sequential Java is up to 612%. The speedup

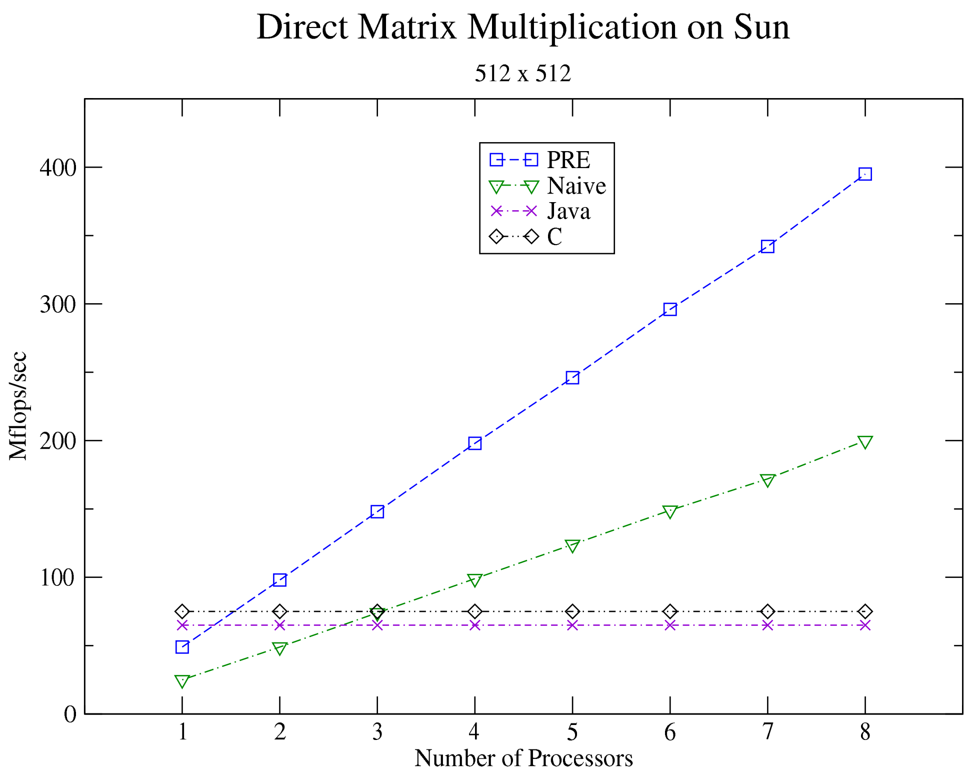

Figure 7.1. Performance for Direct Matrix Multiplication on SMP.

Table 7.3. Speedup of the naive translation and PRE for each number of processors over the performance with one processor for the direct matrix multiplication on SMP.

Number of Processors	2	3	4	5	6	7	8
Naive translation	1.96	2.96	3.96	4.96	5.96	6.88	8.00
PRE	2.02	3.02	4.04	5.02	6.04	6.98	8.06

of PRE over the naive translation is up to 202%. Performance of PRE overtakes that of sequential Java on 2 processors.

The table 7.3 shows the speedup of the naive translation and PRE for each number of processors over the performance with one processor on the shared memory machine. The naive translation gets up to 800% speedup using 8 processors, compared to performance with a single processor on the shared memory machine. Moreover, PRE gets up to 806% speedup.

The direct matrix multiplication doesn't have any run-time communications. But, this is reasonable since we focus on benchmarking the HPJava compiler in this dissertation, not communication libraries. In the next section, we will benchmark examples with run-time communications.

7.2 Laplace Equation Using Red-Black Relaxation

In Figure 3.11, we have introduced a run-time communication library call, `Adlib.writeHalo()`, updating the cached values in the ghost regions with proper element values from neighboring processes. With ghost regions, the inner loop of algorithms for stencil updates can be written in a simple way, since the edges of the block don't need special treatment in accessing neighboring elements. Shifted indices can locate the proper values cached in the ghost region. This is a very important feature in real codes. Thus, the main issue for this subsection is to assess the performance bottleneck in an HPJava program from run-time communication library calls. Through benchmarking both Laplace equations with and without `Adlib.writeHalo()`, we will analyze the effect and latency of the library call in real codes[1].

Figure 7.2 shows the performance of the Laplace equation using red-black relaxation without `Adlib.writeHalo()` on the shared memory machine. Again, we need to see the Java performance over the C performance on the shared memory machine. It is 98% over C. It is quite a satisfactory performance achievement for Java.

The table 7.4 shows the speedup of the naive translation over sequential Java program. Moreover, it shows the speedup of HPJOPT2 over the naive translation. The speedup of the naive translation over sequential Java is up to 360% with 8 processors. The speedup of HPJOPT2 over sequential Java is up to 487% with 8 processors. The speedup of HPJOPT2 over the naive translation is up to 137%.

The table 7.5 shows the speedup of the naive translation and HPJOPT2 for each number of processors over the performance with one processor. The naive translation gets up to

[1]Without `Adlib.writeHalo()`, the algorithm will give incorrect answers. But, we will disregard this, since the purpose of this experiment is to benchmark the performance of the HPJava *compiler*, and to verify the effect of the optimization strategies.

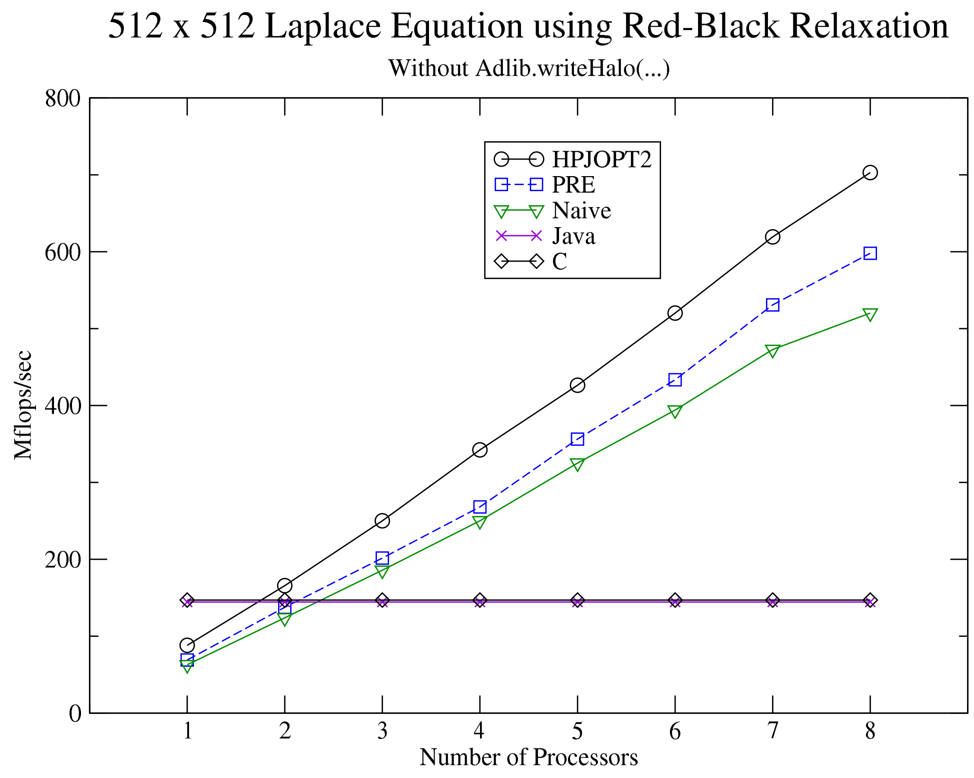

Figure 7.2. 512 × 512 Laplace Equation using Red-Black Relaxation without `Adlib.writeHalo()` on shared memory machine

Table 7.4. Speedup of the naive translation over sequential Java and C programs for the Laplace equation using red-black relaxation without `Adlib.writeHalo()` on SMP.

Number of Processors	1	2	3	4	5	6	7	8
Naive translation over Java	0.43	0.86	1.29	1.73	2.25	2.73	3.27	3.60
HPJOPT2 over Java	0.59	1.15	1.73	2.35	2.95	3.60	4.29	4.87
HPJOPT2 over Naive translation	1.36	1.34	1.35	1.37	1.31	1.32	1.31	1.35

Table 7.5. Speedup of the naive translation and HPJOPT2 for each number of processors over the performance with one processor for the Laplace equation using red-black relaxation without `Adlib.writeHalo()` on SMP.

Number of Processors	2	3	4	5	6	7	8
Naive translation	1.97	2.96	3.98	5.17	6.27	7.53	8.28
HPJOPT2	1.94	2.92	4.00	4.98	6.08	7.24	8.21

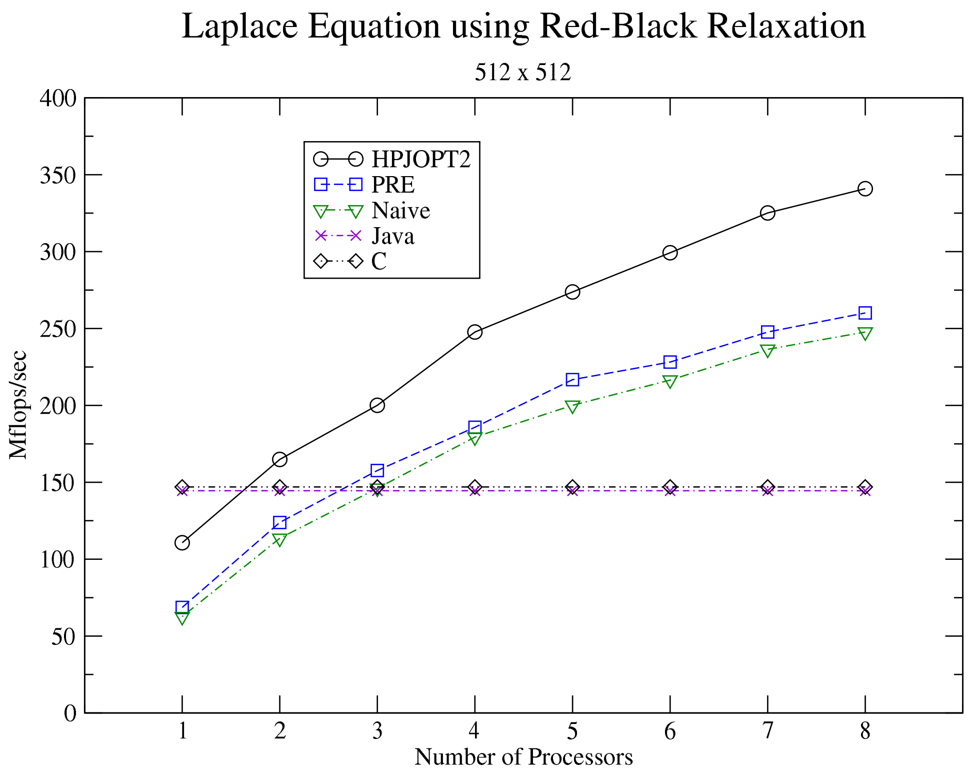

Figure 7.3. Laplace Equation using Red-Black Relaxation on shared memory machine

828% speedup using 8 processors on the shared memory machine. Moreover, HPJOPT2 gets up to 821% speedup.

Thus, if we could ignore `Adlib.writeHalo()`, the HPJava system would give a tremendous performance improvement on the shared memory machine. Moreover, HPJOPT2 optimization scheme works quite well.

Figure 7.3 shows the performance of the Laplace equation using red-black relaxation with `Adlib.writeHalo()` on the shared memory machine. The table 7.6 shows the speedup of the

Table 7.6. Speedup of the naive translation over sequential Java and C programs for the Laplace equation using red-black relaxation on SMP.

Number of Processors	1	2	3	4	5	6	7	8
Naive translation over Java	0.43	0.79	1.01	1.24	1.38	1.50	1.64	1.71
HPJOPT2 over Java	0.77	1.14	1.38	1.71	1.89	2.07	2.25	2.36
HPJOPT2 over Naive translation	1.77	1.45	1.37	1.38	1.37	1.38	1.37	1.38

Table 7.7. Speedup of the naive translation and HPJOPT2 for each number of processors over the performance with one processor for the Laplace equation using red-black relaxation on SMP.

Number of Processors	2	3	4	5	6	7	8
Naive translation	1.81	2.33	2.86	3.19	3.46	3.77	3.95
HPJOPT2	1.49	1.80	2.23	2.47	2.69	2.93	3.07

naive translation over sequential Java and C programs. Moreover, it shows the speedup of HPJOPT2 over the naive translation. The speedup of the naive translation with 8 processors over sequential Java and C is up to 171%. The speedup of HPJOPT2 with 8 processors over sequential Java and C is up to 236%. The speedups of HPJOPT2 over the naive translation is up to 177%.

The table 7.7 shows the speedup of the naive translation and HPJOPT2 for each number of processors over the performance with one processor. The naive translation gets up to 395% speedup using 8 processors on the shared memory machine. Moreover, HPJOPT2 gets up to 307% speedup. Compared with the performance without `Adlib.writeHalo()`, it is up to 200% slower. However, in both programs with and without the library call, the optimization achievement is quite consistent. It means obviously optimization strategies are not affected by the run-time communication library calls. They only make the running time of HPJava programs slow down. Thus, when using run-time communication libraries, we need to carefully design HPJava programs since the latency can't be just disregarded. We want to note the shared memory version of Adlib was implemented very naively by porting the transport layer, `mpjdev`, to exchange messages between Java and threads. A more careful port for the shared memory version would probably give much better performance.

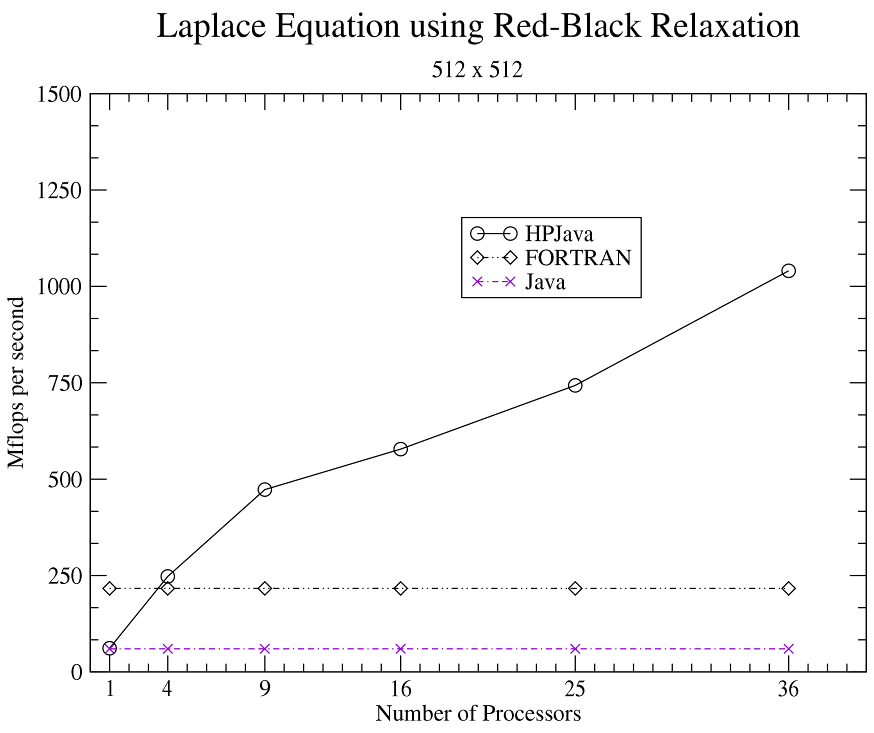

Figure 7.4. Laplace Equation using Red-Black Relaxation on distributed memory machine

Table 7.8. Speedup of the naive translation for each number of processors over the performance with one processor for the Laplace equation using red-black relaxation on the distributed memory machine.

Number of Processors	4	9	16	25	36
Naive translation	4.06	7.75	9.47	12.18	17.04

Figure 7.4 shows performance for the Laplace equation using red-black relaxation on the distributed memory machine. Table 7.8 also shows the speedup of the naive translation for each number of processors over the performance with one processor for the Laplace equation using red-black relaxation on the machine.

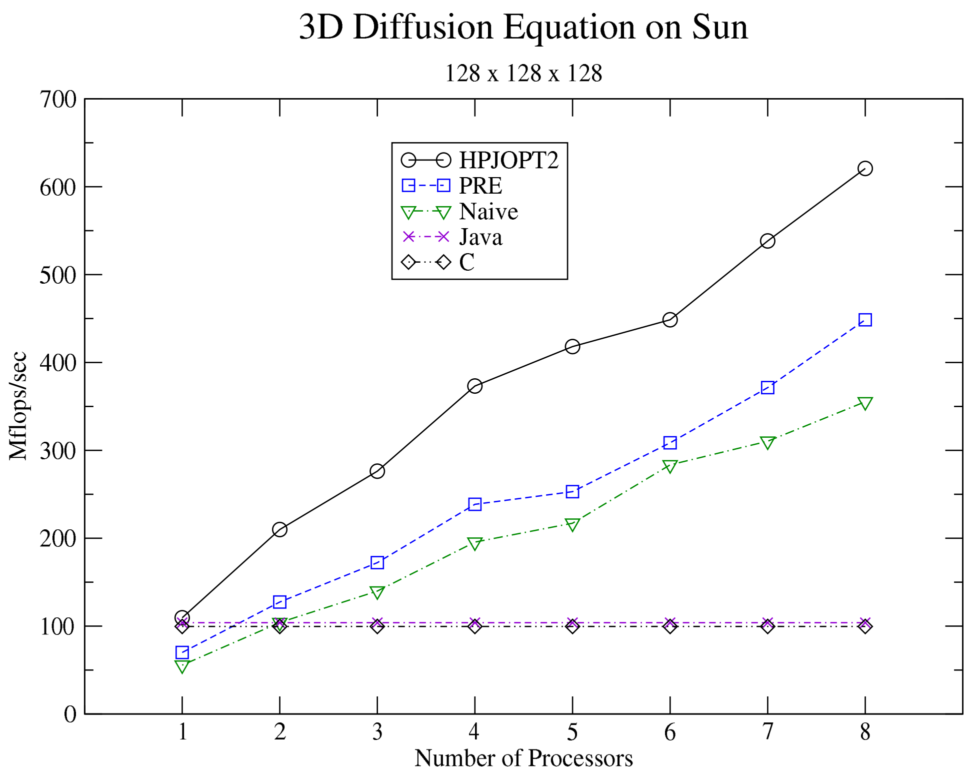

Figure 7.5. 3D Diffusion Equation on shared memory machine

7.3 3-Dimensional Diffusion Equation

Figure 7.5 shows the performance of the 3D diffusion on the shared memory machine. Again, we need to see the Java performance over the C performance on the shared memory machine. It is 104.33% over C.

The table 7.9 shows the speedup of the HPJava naive translation over sequential Java and C programs. Moreover, it shows the speedup of HPJOPT2 over the naive translation.

The speedup of the naive translation with 8 processors over sequential Java is up to 342%. The speedup of HPJOPT2 with 8 processors over sequential Java is up to 598%. The speedup of HPJOPT2 over the naive translation is up to 175%.

The table 7.10 shows the speedup of the naive translation and HPJOPT2 for each number of processors over the performance with one processor. The naive translation gets up to 639% speedup using 8 processors on the shared memory machine. Moreover, HPJOPT2 gets up to 567%.

Table 7.9. Speedup of the naive translation over sequential Java and C programs for the 3D Diffusionequation on the shared memory machine.

Number of Processors	1	2	3	4	5	6	7	8
Naive translation over Java	0.54	1.00	1.35	1.88	2.09	2.73	2.99	3.42
HPJOPT2 over Java	1.05	2.02	2.66	3.59	4.02	4.32	5.18	5.98
HPJOPT2 over Naive translation	1.97	2.02	1.97	1.91	1.93	1.58	1.73	1.75

Table 7.10. Speedup of the naive translation and HPJOPT2 for each number of processors over the performance with one processor for the 3D Diffusion equation on the shared memory machine.

Number of Processors	2	3	4	5	6	7	8
Naive translation	1.87	2.52	3.52	3.90	5.10	5.58	6.39
HPJOPT2	1.92	2.52	3.41	3.82	4.10	4.92	5.67

Table 7.11. Speedup of the naive translation for each number of processors over the performance with one processor for the 3D diffusion equation on the distributed memory machine.

Number of Processors	4	9	16	32
Naive translation	3.31	5.76	9.98	13.88

Figure 7.6 shows performance for the 3D diffusion equation on the distributed memory machine. Table 7.11 also shows the speedup of the naive translation for each number of processors over the performance with one processor for the 3D diffusion equation on the machine.

7.4 Q3 – Local Dependence Index

Figure 7.7 shows the performance of Q3 on the shared memory machine. Again, we need to see the Java performance over the C performance on the shared memory machine. It is 55% over C.

The table 7.12 shows the speedup of the HPJava naive translation over sequential Java and C programs. Moreover, it shows the speedup of HPJOPT2 over the naive translation.

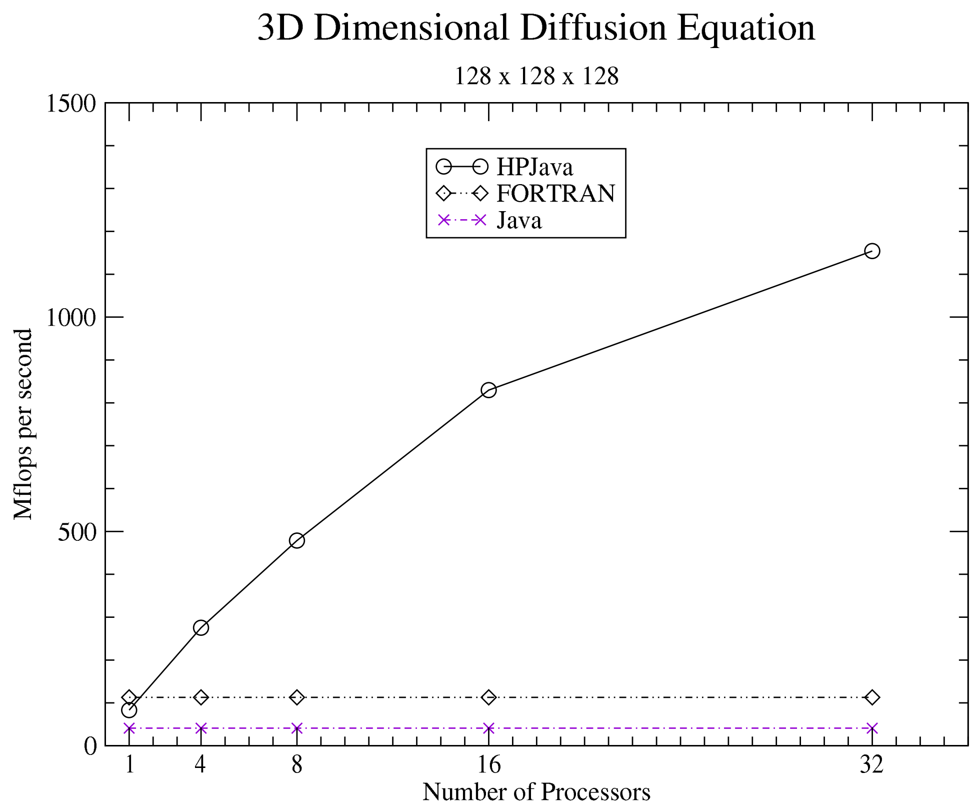

Figure 7.6. 3D Diffusion Equation on distributed memory machine

The speedups of the naive translation with 8 processors over sequential Java and C is up to 1802%. The speedups of HPJOPT2 with 8 processors over sequential Java and C is up to 3200%. The speedup of HPJOPT2 over the naive translation is up to 181%. We recall that performance of Q3 is slow compared to other applications on the Linux machine. As expected, with multi-processors, performance of Q3 is excellent even without any optimizations. It illustrates that performance of HPJava can be outstanding for applications with large problem sizes.

The table 7.13 shows the speedup of the naive translation and HPJOPT2 for each number of processors over the performance with one processor. The naive translation gets up to 1226% speedup using 8 processors on the shared memory machine. Moreover, HPJOPT2 gets up to 1729% speedup. Unlike traditional benchmark programs, Q3 gives a tremendous speedup with a moderate number (= 8) of processors.

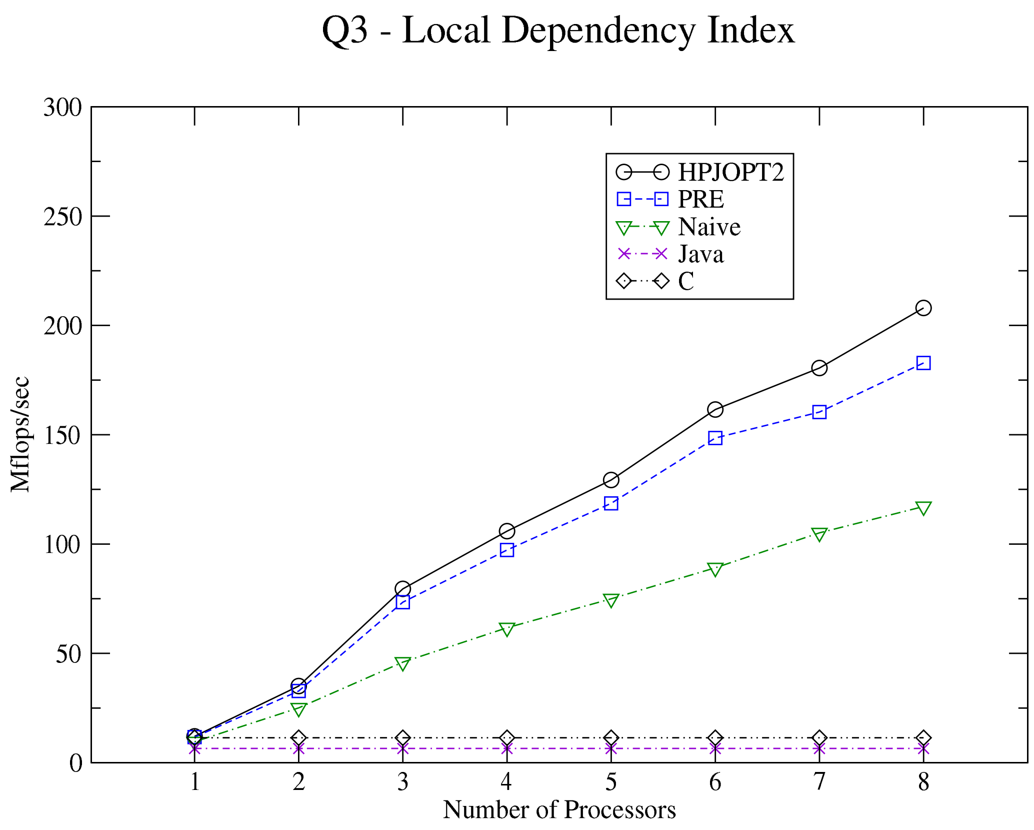

Figure 7.7. Q3 on shared memory machine

Table 7.12. Speedup of the naive translation over sequential Java and C programs for Q3 on the shared memory machine.

Number of Processors	1	2	3	4	5	6	7	8
Naive translation over Java	1.47	3.85	7.08	9.50	11.53	13.71	16.18	18.02
Naive translation over C	0.84	2.20	4.04	5.42	6.58	7.83	9.23	10.29
HPJOPT2 over Java	1.85	5.39	12.23	16.30	19.89	24.81	27.77	32.00
HPJOPT2 over Naive translation	1.26	1.40	1.73	1.72	1.73	1.81	1.72	1.76

Table 7.13. Speedup of the naive translation and HPJOPT2 for each number of processors over the performance with one processor for Q3 on the shared memory machine.

Number of Processors	2	3	4	5	6	7	8
Naive translation	2.62	4.81	6.46	7.84	9.32	11.00	12.26
HPJOPT2	2.91	6.61	8.80	10.75	13.40	15.01	17.29

7.5 Discussion

In this chapter, we have experimented with and benchmarked the HPJava language on scientific and engineering applications on a shared memory machine (Sun Solaris 9 with 8 Ultra SPARC III Cu 900 MHz Processors and 16 GB of main memory) and a distributed memory machine (IBM SP3 running with four Power3 375 MHz CPUs and 2 GB of memory on each node).

We have explored the performance of the HPJava system on both machines using the efficient node codes we have benchmarked in chapter 6. The speedup of each HPJava application is very satisfactory even with expensive run-time communication libraries such as `Adlib.writeHalo()` and `Adlib.sumDim()`. Moreover, performances on both machines shows consistent and similar behaviour as we have seen on the Linux machine. One machine doesn't have a big advantage over others. Performance of HPJava is good on all machines we have benchmarked.

When program architects design and build high-performance computing environments, they have to think about what system they should choose to build and deploy the environments. There rarely exists machine-independent software, and performance on each machine is quite inconsistent. HPJava has an advantage over some systems because performance of HPJava on Linux machines, shared memory machines, and distributed memory machines are consistent and promising. Thus, we hope that HPJava has a promising future, and can be used anywhere to achieve high-performance parallel computing.

CHAPTER 8

RELATED SYSTEMS

There are several language projects related to our HPspmd programming model. Examples are Co-Array Fortran, ZPL, JavaParty, Timber, and Titanium. Co-Array Fortran is an extended Fortran dialect for SPMD programming. ZPL is a new array language for scientific and engineering computations. JavaParty is a cluster-based parallel programming in Java. Timber is a parallelizing static compiler for a superset of Java. Titanium is a language and system for high-performance parallel computing, and its base language is Java. In this chapter we will discuss these related languages and compare them with our HPspmd model.

8.1 Co-Array Fortran

Co-Array Fortran [36], formerly called F−−, is a simple and small set of extensions to Fortran 95 for Single Program Multiple Data, SPMD, parallel processing. It feels and look similar to Fortran and has a few new extensions associated with *work distribution* and *data distribution.*

We review the work distribution first. A single program is replicated a fixed number of times. Each replication, called an *image*[1], has its own data objects. Each image is asynchronously executed with the normal rules of Fortran, so the execution path may differ from image to image. The programmer decides the actual path for the image through a unique image index, by using normal Fortran control constructs and by explicit synchronizations.

For data distribution, Co-Array Fortran lets the programmer choose data distribution in a syntax very much like normal Fortran array syntax. Array indices in *parentheses* follow the normal Fortran rules within one memory image. Array indices in *square brackets* provide convenient notation for accessing objects *across* images and follow similar rules. The

[1]An image is what HPJava calls a *process*.

programmer uses co-array syntax only where it is needed. A reference to a co-array with no square brackets attached to it is a reference to the object in the local memory of the executing image. Since most references to data objects in a parallel code should be to the local part, co-array syntax should appear only in isolated parts of the code. If not, the syntax acts as a visual flag to the programmer that too much communication among images may be taking place. It also acts as a flag to the compiler to generate code that avoids latency whenever possible.

Co-arrays provides a way of expressing remote memory operations. For example:

```
X       = Y[PE]   ! get from Y[PE]
Y[PE]   = X       ! put into Y[PE]
Y[:]    = X       ! broadcast X
Y[LIST] = X       ! broadcast X over subset of PE's in array LIST
Z(:)    = Y[:]    ! collect all Y
S = MINVAL(Y[:])  ! min (reduce) all Y
B(1:M)[1:N] = S   ! S scalar, promoted to array of shape (1:M,1:N)
```

The approach is different to HPJava. In Co-Array Fortran, 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 Co-Array Fortran the logical model of communication is built into the language—remote memory access with intrinsics for synchronization. In HPJava, there are no communication primitives in the language itself. We follow the MPI philosophy of providing communication through separate libraries. While Co-Array Fortran and HPJava share an underlying programming model, we claim that our framework may offer greater opportunities for exploiting established software technologies.

8.2 ZPL

ZPL [45] is an array programming language designed from first principles for fast execution on both sequential and parallel computers for scientific and engineering computation.

In scalar languages such as Fortran, C, Pascal, Ada, and so on, $a + b$ represents the addition of two numbers, i.e. operations apply only to single values. In such languages we need looping and indexing in order to add two arrays:

```
DO I = 1, N
  DO J = 1, N
    A(I,J) = A(I,J) + B(I,J)
  ENDDO
ENDDO
```

In HPJava we might use `overall` constructs. In ZPL operations are generalized to apply to both scalars and arrays. Thus, $a + b$ expresses the sum of two scalars if a and b were declared as scalars, or arrays if they were declared as arrays. When applied to arrays, the operations act on corresponding elements as illustrated in the loops above. When the ZPL compiler encounters the statement

```
A := A + B;
```

and A and B are two dimensional arrays, it generates code that is effectively the same as the Fortran loops shown above. An array language, therefore, simplifies programming. Of course this is very similar to Fortran 90.

ZPL 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 similarities to our `overall` construct. In ZPL, parallelism and communication are more implicit than in HPJava. The connection between ZPL programming and SPMD programming is not explicit. While there are certainly attractions to the more abstract point of view, HPJava deliberately provides lower-level access to the parallel machine.

Figure 8.1 illustrates a ZPL program for Jacobi. Compared to an HPJava program for Jacobi in Figure 3.7, HPJava uses a nested `overall` loop for this computation, and ZPL uses a loop in the main body because of the generalized array operations.

8.3 JavaParty

JavaParty [38] allows easy ports of multi-threaded Java programs to distributed environments such as clusters. Regular Java already supports parallel applications with threads and synchronization mechanisms. While multi-threaded Java programs are limited to a single address space, JavaParty extends the capabilities of Java to distributed computing environments. It adds remote objects to Java purely by declaration, avoiding disadvantages of explicit socket communication and the programming overhead of RMI.

```
/*                      Jacobi                       */
program jacobi;

config var n     : integer = 5;       -- Declarations
           delta : float   = 0.0001;
region     R = [1..n, 1..n];
direction  north = [-1, 0]; south = [ 1, 0];
           east  = [ 0, 1]; west  = [ 0,-1];

procedure jacobi();                   -- Entry point
var A, Temp : [R] float;
    err     :     float;

     begin
[R]            A := 0.0;              -- Initialization
[north of R]   A := 0.0;
[east  of R]   A := 0.0;
[west  of R]   A := 0.0;
[south of R]   A := 1.0;

[R]  repeat                           -- Main body
         Temp := (A@north+A@east+A@west+A@south) / 4.0;
         err  := max<< abs(A-Temp);
         A    := Temp;
     until err < delta;

[R]  writeln(A);                      -- Output result
     end;
```

Figure 8.1. A simple Jacobi program in ZPL.

The basic approach of JavaParty is that a multi-threaded Java program can be easily transformed to a distributed JavaParty program by identifying the classes and threads that should be spread across the distributed system. A programmer can indicate this by a newly introduced class modifier, `remote`, which is the only extension of Java.

As a parallel language, the most important feature of JavaParty is its "location transparency." Similar to HPF, programmers don't have to distribute remote objects and remote threads to specific nodes because the compiler and run-time system deal with locality and communication optimization.

JavaParty is implemented as a pre-processor phase to a Java compiler. It currently uses RMI as a target and thus inherits some of RMI's advantages such as distributed garbage collection. JavaParty codes is translated into regular Java code with RMI hooks. For example,

```
remote class B extends A implements C {
    T x = I ;          // instance variable
    void foo() { Q }   // method
}
```

is translated into

```
interface B_class extends RemoteClassIntf { ... }

class B_class_impl extends RemoteClass implements B_class {
    T x = I ;                                // instance variable
    public void foo() throws RemoteException { // method
        toRemote(Q) ;
    }
}
```

JavaParty is for parallel cluster programming in Java. It has important contribution to research on Java-based parallel computing. Moreover, because the only extension is the `remote` keyword, it is systematically simple and easy to be used and to be implemented. However, HPJava provides lower-level access to the parallel machine. Compared to the HPJava system, the basic approach of JavaParty, remote objects with RMI hooks, might become its bottleneck because of unavoidable overhead of RMI and limited evidence that the remote procedure call approach is good for SPMD programming.

8.4 Timber

Timber [47] is a Java-based programming language for semi-automatic array-parallel programming, in particular for programming array-based applications. The language has been designed as part of the Automap project, in which a compiler and run-time system are being developed for distributed-memory systems. Apart from a few minor modifications, Timber is still a superset of Java.

We see some array declarations of array variables in Timber, for example:

```
int [*] a = new int[4] ;
real [*,*] ident = {{1,0,0}, {0,1,0}, {0,0,1}} ;
real [*][*] vv   = {{1,0,0}, {0,1,0}, {0,0,1}} ;

ident [1,2] = 1 ;      // An array access
```

Even if `ident` and `vv` have the same initialization expression, they are *different* arrays. `ident` is a two-dimensional array, and `vv` is a one-dimensional array of one-dimensional arrays. Array accessing is very similar to that in other languages.

Timber provides the `each` and `foreach` constructs, which specify that the iterations can be executed in arbitrary order, but that each iteration is to be executed sequentially. For some analysis, the compiler can translate most `foreach` constructs into explicit parallel loops in SPC [46] form. This intermediate form is used for automatic task mapping.

```
each {
  s1 ;
  s2 ;
}

foreach (i = 0 : n) {
  a[i].init() ;
}
```

For the `each` construct, `s1` and `s2` are executed in arbitrary order. The `foreach` construct is a parameterized version of the `each` construct. In above example, it invokes the `init()` method on n members of array `a`.

Like HPJava, Timber introduces multidimensional arrays, array sections, and a parallel loop. They have some similarities in syntax, but semantically Timber is very different to HPJava. Although Timber supports parallel operations such as `each`, `foreach` constructs, it is difficult to say Timber targets massively parallel distributed memory computing, using HPF-like multiarrays, and supports lower-level access to the parallel machines. Moreover, for high-performance, Timber chose C++ as its target language. But it becomes its own bottleneck since Java has been improved and C++ is less portable and secure in the modern computing environment.

8.5 Titanium

Titanium [48] is another Java-based language (not a strict extension of Java) for high-performance computing. Its compiler translates Titanium into C. Moreover, it is based on a parallel SPMD model of computation.

The main new features of Titanium are immutable classes, explicit support for parallelism, and multi-dimensional arrays. Immutable classes are not extensions of any existing class. This restriction allows the compiler to pass such objects by value and to allocate them on the stack. They behave like Java primitive types or C structs.

Titanium arrays are multi-dimensional, distinct from Java arrays. They are constructed using *domains*, which specify their index sets, and are indexed by *points*. Points are tuples of integers and domains are sets of points. The following code shows an example declaration of a multi-dimensional array.

```
Point<2> l = [1, 1] ;
Point<2> u = [10, 20] ;
RectDomain<2> r = [l : u] ;
double [2d]   A = new double[r] ;
```

Moreover, a multi-dimensional iterator, `foreach` allows one to operate on the elements of A, as follows;

```
foreach (p in A.domain()) {
    A[p] = 42 ;
}
```

Titanium is originally designed for high-performance computing on both distributed memory and shared memory architectures. (Support for SMPs is not based on Java threads.) The system is a Java-based language, but, its translator finally generates C-based codes for performance reason. In the current stage development of Java, performance has almost caught up with C and Fortran, as discussed in chapter 6. Translating to C-based codes is no longer necessarily an advantage in these days. In addition, it doesn't provide any special support for distributed arrays and the programming style is quite different to HPJava.

8.6 Discussion

In this chapter we have reviewed and compared some parallel language and library projects to our HPJava system. Compared against others, each language and library has its advantages and disadvantages.

A biggest edge the HPJava system has against other systems is that HPJava not only is a Java extension but also is translated from a HPJava code to a pure Java byte code. Moreover, because HPJava is totally implemented in Java, it can be deployed any systems without any changes. Java is object-oriented and highly dynamic. That can be as valuable in scientific computing as in any other programming discipline. Moreover, it is evident that SPMD environments are successful in high-performance computing, despite that programming applications in SPMD-style is relatively difficult. Our HPspmd programming model targets SPMD environments with lower-level access to the parallel machines unlike other systems.

In the chapter 6, we have discussed Java's potential to displace established scientific programming languages. According to this argument, HPJava, an HPspmd programming model, has a high advantage over other parallel languages.

Many systems adopted implicit parallelism and simplicity for users. But this results in the difficulty of implementing the system. Some systems chose C, C++, and Fortran as their target language for high-performance. But, mentioned earlier, Java is a very competitive language for scientific computing. The choice of C or C++ makes the systems less portable and secure in the modern computing environment.

Thus, because of the *popularity*, *portability*, *high-performance* of Java and SPMD-style programming, the HPJava system gets many advantages over other systems.

CHAPTER 9

CONCLUSIONS

9.1 Conclusion

In this dissertation, we gave a historical review of data-parallel languages such as High Performance Fortran (HPF), some message-passing frameworks including p4, PARMACS, and PVM, as well as the MPI standard, and high-level libraries for multiarrays such as PARTI, the Global Array (GA) Toolkit, and Adlib.

Moreover, we introduced an HPspmd programming language model—a SPMD framework for using libraries based on multiarrays. It adopted the model of distributed arrays standardized by the HPF Forum, but relinquished the high-level single-threaded model of the HPF language. Also, we described a Java extension of HPspmd programming language model, called HPJava. Via examples, we reviewed the specific syntax and a few new control constructs of HPJava. Most importantly we discussed the compilation strategies such as type-analysis, pre-translation, basic translation schemes, and some optimizations of HPJava.

To verify our compilation schemes are appropriate for the current HPJava system, we experimented with and benchmarked some scientific and engineering applications in HPJava. The result of the experimental studies and benchmarks was quite satisfactory as an initial benchmark for a brand-new language[1]. Moreover, we proved that the performance of Java applications are almost caught up with that of C applications in several machines.

Finally, we compared and contrasted our HPspmd programming language model with several other language projects such as Co-Array Fortran, ZPL, JavaParty, Titanium, and timber, similar to our HPspmd programming language model, as well as a parallel C++ library as a superset of STL, called STAPL.

[1]In case of HPF, scientists and engineers are still developing its compilation systems for efficient performance a decade after its initial appearance.

9.2 Contributions

The primary contributions of this dissertation are to propose the potential of Java as a scientific parallel programming language and to pursue efficient compilation of the HPJava language for high-performance computing. By experimenting with and benchmarking large scientific and engineering HPJava programs on a Linux machine, a shared memory machine, and a distributed machine, we have proved that the HPJava compilation and optimization scheme generates efficient node code for parallel computing.

- The HPJava design team, composed of Bryan Carpenter, Geoffrey Fox, Guansong Zhang, and Xiaoming Li, designed and proposed the HPspmd programming Language Model. The major goal of the system we are building is to provide a programming model that is a flexible hybrid of HPF-like data-parallel language and the popular, library-oriented, SPMD style without the basic assumptions of the HPF model. When the project was proposed, the underlying run-time communication library was available [11], the translator itself was being implemented in a compiler construction framework developed in the PCRC project [13, 51]. Soon afterwards, JavaCC and JTB tools were adopted for the HPJava parser.

- The author, Han-Ku Lee, joined the HPJava development team in 1998. Since then, he has developed the HPJava front-end and back-end environments and implementations including the HPJava Translator, the Type-Analyzer for analyzing source classes' hierarchy, and the Type-Checker for safe type-checking as well as implemented the JNI interfaces of the run-time communication library, Adlib. In addition, he has maintained the HPJava Parser, implemented the Abstract Expression Node generator for intermediate codes, the Unparser to unparse translated codes. At the School of Computational Science and Information Technology (CSIT) at Florida State University (FSU), Tallahassee, FL, the most time-consuming parts such as type-checker has been firmly implemented, and the first fully functional translator were developed.

- Currently, the author and the HPJava development team keep researching and improving the HPJava system. The main concentration of the HPJava system is now to benchmark the current HPJava system on different machines such as Linux, SMPs, and

IMB SP3. Moreover, the HPJava development team is concentrating on optimization strategies for the translator and pure-Java implementation for the run-time communication libraries to make the system competitive with existing Fortran programming environments.

- The author has significantly contributed to the HPJava system in developing and maintaining the HPJava front-end and back-end environments at NPAC, CSIT, and Pervasive Technology Labs. He has developed the Translator, the Type-Checker, and the Type-Analyzer of HPJava, and has researched the HPJava optimization scheme. Some of his early work at NPAC are the Unparser and the Abstract Expression Node generator, and original implementation of the JNI interfaces of the run-time communication library, Adlib. Recently he focused on the topic "Efficient Compilation of the HPJava Language for Parallel Computing" for optimization of the HPJava system.

9.3 Future Works

9.3.1 HPJava

In a future version of HPJava, we will complete the HPJava optimization phase. Further research work will include developing simpler translation schemes and a more efficient optimizer, and developing more efficient run-time libraries for shared memory machines and a pure Java version of run-time library for distributed memory machines. In addition, we will do more benchmarks on distributed memory machines.

9.3.2 High-Performance Grid-Enabled Environments

9.3.2.1 Grid Computing Environments

Grid computing environments can be defined as computing environments that are fundamentally distributed, heterogeneous, and dynamic for resources and performance. As inspired by [19], the Grid will establish a huge environment, connected by global computer systems such as end-computers, databases, and instruments, to make a World-Wide-Web-like distributed system for science and engineering.

The majority of scientific and engineering researchers believe that the future of computing will depend heavily on the Grid for efficient and powerful computing, improving legacy technology, increasing demand-driven access to computational power, increasing utilization of idle capacity, sharing computational results, and providing new problem-solving techniques and tools. Of course, substantially powerful Grids can be established using high-performance networking, computing, and programming support regardless of the location resources and users.

What then will be the biggest potential issues in terms of programming support to simplify distributed heterogeneous computing in the same way that the World-Wide-Web simplified information sharing over the internet? *High-performance* is one possible answer since a slow system which has a clever motivation is useless. Another answer could be the thirst for *grid-enabled applications*, hiding the "heterogeneity" and "complexity" of grid environments without losing performance.

Today, grid-enabled application programmers write applications in what, in effect, is assembly language: sometimes using explicit calls to the Internet Protocol's User Datagram Protocol (UDP) or Transmission Control Protocol (TCP), explicit or no management of failure, hard-coded configuration decisions for specific computing systems. We are somewhat far from portable, efficient, high-level languages.

9.3.2.2 HPspmd Programming Model Towards Grid-Enabled Applications

To support "high-performance grid-enabled applications", the future grid computing systems will need to provide *programming models* [19]. The main thrust of programming models is to hide and simplify complexity and details of implementing the system, while focusing on the application design that have a significant impact on program performance or correctness.

Generally, we can see different programming models in sequential programming and parallel programming. For instance, in sequential programming, commonly used programming models for modern high-level languages furnish applications with inheritance, encapsulation, and scoping. In parallel programming, distributed arrays, message-passing, threads, condition variables, and so on. Thus, using each model's significant characteristics, sequential and parallel program must maximize their performance and correctness.

There is no clarity about what programming model is appropriate for a grid environment, although it seems clear that many programming models will be used.

One approach to grid programming is to adapt programming models that have already proved successful in sequential or parallel environments. For example, the data-parallel language model such as our HPspmd Programming Model might be an ideal programming model for supporting and developing high-performance grid-enabled applications, allowing programmers to specify parallelism in terms of process groups and multiarray operations. A grid-enabled MPI[2] would extend the popular message-passing models. New Java I/O API's dramatic performance improvement encourages us to focus on grids-enabled MPI implementations as well. Moreover, high-performance grid-enabled applications and run-time systems demand "adaptability", "security", and "ultra-portability", which can be simply supported by the HPJava language since it is implemented in the context of Java.

Despite tremendous potential, enthusiasm, and commitment to the Grid, few software tools and programming models exist for high-performance grid-enabled applications. Thus, to make prospective high-performance grid-enabled environments, we need nifty compilation techniques, high-performance grid-enabled programming models, applications, and components, and a better and improved base language (e.g. Java).

The HPJava language has quite acceptable performance on scientific and engineering (matrix) algorithms, which play very important roles in high-performance grid-enabled applications such as "search engines" and "parameter searching". Moreover, the most interesting Grid problem where HPJava is adoptable is "coordinating" the execution and information flow between multiple "web services" where each web service has WSDL style interface and some high level information describing capabilities. It can even help parallel computing by specifying compiler hints. In the near future, HPJava will be mainly used as a middleware to support "complexity scripts" in the project, called "BioComplexity Grid Environment" at Indiana University.

Thus, we believe that the success of HPJava would make our HPspmd Programming Model a promising candidate for constructing high-performance grid-enabled applications and components.

[2]currently our HPJava project team provides a Java implementation of MPI, called *mpiJava*

9.3.3 Java Numeric Working Group from Java Grande

The goals of the Numeric Working Group of the Java Grande Forum are (1) to assess the suitability of Java for numerical computation, (2) to work towards community consensus on actions which can be taken to overcome deficiencies of the language and its run-time environment, and (3) to encourage the development of APIs for core mathematical operations. The Group hosts open meetings and other forums to discuss these issues.

The recent efforts of Java Numeric Working Group are true multidimensional arrays, inclusion of static imports, Inclusion of facilities for automatic boxing and unboxing of primitive types, Inclusion of enhanced `for` loops (i.e., `foreach`), Improvements in java.lang.Math, Multiarray Package, and etc.

Especially, *the HPJava language contributes to true multidimensional arrays and its syntax on Java.* Jose Moreira, etc, provided an update on the progress of Java Specification Request (JSR) 83 on multiarrays, and especially adopted the same syntax as HPJava distributed arrays and true multi-dimensional arrays. A *multiarray* is a rectangular multidimensional array whose shape and extents are fixed. Such arrays admit compiler optimizations that Java arrays of arrays do not. That is, with multiarrays, Java can achieve very high-performance for numerical computing through the use of compiler techniques and efficient implementation of array operations.

As one of the Java Numeric Working Group members, I believe that the endless efforts will make Java (language specification, run-time environments, and its compiler) a better and more adaptable language for Grid Computing environments.

9.3.4 Web Service Compilation (i.e. Grid Compilation)

One of our future research interests is in developing the Web Service Compilation (i.e. Grid Compilation) technique. As a good example of Grid Computing System, web services are the fundamental building blocks in the move to distributed computing on the Internet. A *web service* is any service that is available over the Internet, uses a standard XML messaging system, and is not tied to any one operating system or programming language. Basically, a web service protocol stack consists of SOAP (i.e. how to talk to web services), WSDL (i.e. how web services are described), and UDDI (i.e. how to find web services).

There is a common feature between Parallel Computing (e.g. HPJava) and Grid Computing (e.g. Web Service). Both computing systems heavily depend upon *messaging*. Although explicit message-passing is a necessary evil for parallel computing, messaging is the natural architecture for Grid Computing. On the current scientific and engineering technique, the main difference for messaging between parallel computing and Grid Computing is the latency. For example, MPI's latency may be 10 microseconds. How can the message systems of Grid Computing catch up with MPI? There are some areas we can research to improve the messaging for performance-demand Grid Computing systems.

For a moment, we have many Web Service tools such as XML, SOAP, SAX, DOM, XPATH, XSLT, etc, which are very high-level and can be improved for developing *low latency high-performance Grid-communications*. For example, the *canonicalization* technique of the original XML documents can speed up the time to process XML documents by a factor of two. Unfortunately, so far, there is no standard XSL output method which could be used with the identity transformer to generate a canonical form of a source document. Thus, this canonicalization can be researched and accomplished by my pre-processor experience and expertise of the HPspmd programming models.

The other idea for speed-up of Grid Computing systems is that we need some different approaches for messaging, compared with legacy techniques. For instance, A/V sessions require some tricky set-up processes where the clients interested in participating, joining, and negotiating the session details. This part of the process has no significant performance issues and can be implemented with XML. The actual audio and video traffic does have performance-demands and here we can use existing faster protocol. Many applications consist of many control messages, which can be implemented in basic Web Service way and just the part of the messaging needs good performance. Thus, we end up with control ports running the basic WSDL with possible high-performance ports bound to a different protocol. This well-organized design and well-isolated messaging from the performance-demand applications can lead Grid to low latency high-performance Grid-communication environments.

We call this kind of implementation and design issues to improve Grid Computing environments, the *Web Service Compilation*, or *Grid-Compilation*. We believe that Grid-Compilation will play the most important role in the entire Grid Computing environments

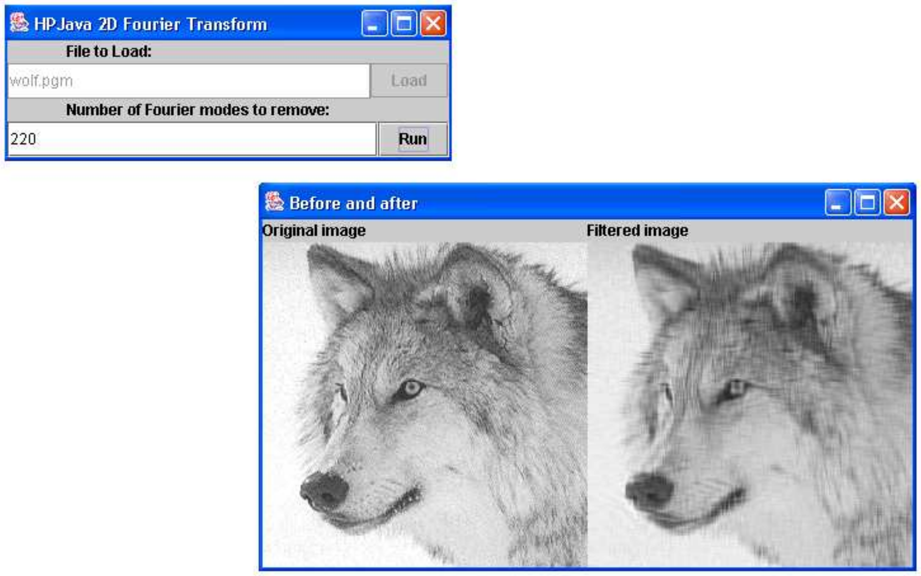

Figure 9.1. Running an HPJava Swing program, `Wolf.hpj`.

since the performance-demands on heterogeneous systems are the most key issues for computational grids.

9.4 Current Status of HPJava

The first version of the HPJava translator was released and can be downloaded from our Web site at `http://www.hpjava.org`. Full sources are available under a very liberal licensing agreement.

There are two parts to the software. The HPJava development kit, *hpjdk* contains the HPJava compiler and an implementation of the high-level communication library, *Adlib*. The only prerequisite for installing hpjdk is a standard Java development platform, like the one freely available for several operating systems from Sun Microsystems. The installation of hpjdk is very straightforward because it is a pure Java package. Sequential HPJava programs using the standard `java` command can be immediately run. Parallel HPJava programs can also be run with the `java` command, provided they follow the *multithreaded model.*

To distribute parallel HPJava programs across networks of host computers, or run them on supported distributed-memory parallel computers, you also need to install a second HPJava package—*mpiJava*. A prerequisite for installing mpiJava is the availability of an *MPI* installation on your system.

hpjdk contains a large number of programming examples. Larger examples appear in figures, and many smaller code fragments appear in the running text. All non-trivial examples have been checked using the hpjdk 1.0 compiler. Nearly all examples are available as working source code in the hpjdk release package, under the directory `hpjdk/examples/PPHPJ/`. Figure 9.1 is running an HPJava Swing program, `Wolf.hpj`.

9.5 Publications

1. Bryan Carpenter, Han-Ku Lee, Sang Boem Lim, Geoffrey Fox, and Guansong Zhang. *Parallel Programming in HPJava.* April 2003. http://www.hpjava.or.

2. Han-Ku Lee, Bryan Carpenter, Geoffrey Fox, and Sang Boem Lim. *HPJava: Programming Support for High-Performance Grid-Enabled Applications.* June 23 – 26, 2002. The 2003 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03).

3. Han-Ku Lee, Bryan Carpenter, Geoffrey Fox, and Sang Boem Lim. *HPJava: Efficient Compilation and Performance for HPC.* July 27 – 30, 2003. The 7th World Multiconference on Systemics Cybernetics, and Informatics (SCI 2003).

4. Sang Boem Lim, Bryan Carpenter, Geoffrey Fox, and Han-Ku Lee. *Collective Communication for the HPJava Programming Language.* To appear Concurrency: Practice and Experience, 2003.

5. Han-Ku Lee, Bryan Carpenter, Geoffrey Fox, and Sang Boem Lim. *Benchmarking HPJava: Prospects for Performance.* Feb 8, 2002. The Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers(LCR 2002). http://motefs.cs.umd.edu/lcr02.

6. Bryan Carpenter, Geoffrey Fox, Han-Ku Lee, and Sang Boem Lim. *Node Performance in the HPJava Parallel Programming Language.* Feb 8, 2002. Submitted to 16th Annual ACM International Conference on Super Computing(ICS2001).

7. Bryan Carpenter, Geoffrey Fox, Han-Ku Lee, and Sang Boem Lim. *Translation of the HPJava Language for Parallel Programming.* May 31, 2001. The 14th annual workshop on Languages and Compilers for Parallel Computing(LCPC2001). http://www.lcpcworkshop.org/LCPC2001/

REFERENCES

[1] C.A. Addison, V.S. Getov, A.J.G. Hey, R.W. Hockney, and I.C. Wolton. *The Genesis Distributed-Memory Benchmarks.* Elsevier Science B.V., North-Holland, Amsterdam, 1993.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Pub Co, 1986.

[3] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[4] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide.* SIAM, 1997.

[5] J. Boyle, R. Butler, T. Disz, BGlickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors.* Winston, 1987.

[6] Preston Briggs and Keith D. Cooper. Effective Partial Redundancy Elimination. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 159–170, 1994.

[7] R. Butler and E Lusk. Monitors, messages, and clusters: The P4 parallel programming system. volume 20, pages 547–564, 1994.

[8] Bryan Carpenter, Geoffrey Fox, and Guansong Zhang. An HPspmd Programming Model Extended Abstract. 1999. http://www.hpjava.org.

[9] Bryan Carpenter, Han-Ku Lee, Sang Lim, Geofrrey Fox, and Guansong Zhang. *Parallel Programming in HPJava.* Draft, April 2003. http://www.hpjava.org.

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

[11] Bryan Carpenter, Guansong Zhang, and Yuhong Wen. NPAC PCRC runtime kernel definition. Technical Report CRPC-TR97726, Center for Research on Parallel Computation, 1997. Up-to-date version maintained at http://www.npac.syr.edu/projects/pcrc/doc.

[12] W. Chen and D. Thissen. Local dependence indexes for item pairs using item response theory. volume 12, pages 265–289, 1997.

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

[14] Parasoft Corp. Express User's Guide Version 3.2.5. 1992.

[15] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficient Computing Static Single Assignment Form and the Control Dependence Grapg. *ACM Transactions on Programming Languages and Systems*, pages 451–490, 1991.

[16] R. Das, M. Uysal, J.H. Salz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[17] J. Dongarra, G.A. Geist, R. Manchek, , and V.S. Sunderam. Integrated PVM framework supports heterogeneous network computing. volume 7, pages 166–174, 1993.

[18] MPI Forum. MPI: A Message-Passing Interface Standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8.

[19] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[20] Erich Gamma, Richard Helm, Ralph Johson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1998.

[21] William Gropp and Ewing Lusk. MPICH - A Portable Implementation of MPI. http://www.mcs.anl.gov/mpi/mpich.

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

[23] HPJava Home Page. http://www.hpjava.org.

[24] Jacks (Java Automated Compiler Killing Suite). http://oss.software.ibm.com/developerworks/oss/jacks.

[25] JavaCC – Java Compiler Compiler (Parser Generator). http://www.webgain.com/products/java_cc/.

[26] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley Pub Co, 2000.

[27] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial Redundancy Elimination in SSA Form. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, 1999.

[28] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, Jr. G.L. Steel, , and M.E. Zosel. The High Performance Fortran Handbook. *MIT Press*, 1994.

[29] Han-Ku Lee, Bryan Carpenter, Geoffrey Fox, and Sang Boem Lim. Benchmarking HPJava: Prospects for Performance. In *Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers(LCR2002)*, Lecture Notes in Computer Science. Springer, March 2002. http://www.hpjava.org.

[30] Han-Ku Lee, Bryan Carpenter, Geoffrey Fox, and Sang Boem Lim. HPJava: Efficient Compilation and Performance for HPC. In *The 7th World Multiconference on Systemics Cybernetics, and Informatics (SCI 2003)*, Lecture Notes in Computer Science. Springer, July 2003.

[31] Han-Ku Lee, Bryan Carpenter, Geoffrey Fox, and Sang Boem Lim. HPJava: Programming Support for High-Performance Grid-Enabled Applications. In *The 2003 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2003)*, Lecture Notes in Computer Science. Springer, June 2003.

[32] Sang Boem Lim. *Platforms for HPJava: Runtime Support For Scalable Programming In Java*. PhD thesis, The Florida State University, June 2003.

[33] R. J. Mislevy and R. D. Bock. *BILOG3: Item Analysis and Test Scoring with Binary Logistic Models*. Chicago: Scientific Software, 1990.

[34] Etienne Morel and Claude Renvoise. Global Optimization by Suppression of Partial Redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

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

[36] R. Numrich and J. Steidel. F- -: A simple parallel extension to Fortran 90. *SIAM News*, page 30, 1997. http://www.co-array.org/welcome.htm.

[37] Manish Parashar and J.C. Browne. Systems engineering for high performance computing software: The HDDA/DAGH infrastructure for implementation of parallel structured adaptive mesh. In *Structured Adaptive Mesh Refinement Grid Methods*, IMA Volumes in Mathematics and its Applications. Springer-Verlag.

[38] Michael Philippsen and Matthias Zenger. JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225 – 1242, 1997.

[39] Pixar animation studios. http://www.pixar.com.

[40] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in FORTRAN: The Art of Scientific Computing, Second Edition*. Cambrige University Press, 1992.

[41] Calkin R., Hempel R., Hoppe H., and Wypior P. Portable programming with the PARMACS Message-Passing Library. volume 20, pages 615–632, 1994.

[42] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global Value Numbers and Redundant Computations. *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1988.

[43] Andreas G. Schilling. Towards real-time photorealistic rendering: Challenges and solutions. In *Proceedings of the 1997 SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware, Los Angeles, California, August 3–4, 1997*, pages 7–15, August 1997. Invited Paper.

[44] A. Skjellum, S.G. Smith, N.E. Doss, A.P. Leung, and M. Morari. The Design and Evolution of Zipcode. volume 20, pages 565–596, 1994.

[45] L. Snyder. A ZPL Programming Guide. Technical report, University of Washington, May 1997. http://www.cs.washington.edu/research/projects/zpl.

[46] A.J.C. van Gemund. The Importance of Synchronization Structure in Parallel Program Optimization. In *In 11th ACM International Conference on Supercomputing*, Lecture Notes in Computer Science, 1997.

[47] Kees van Reeuwijk, Arjan J. C. van Gemund, and Henk J. Sips. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, 1997.

[48] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: High-Performance Java Dialect. *Concurrency: Practice and Experience*, 10(11-13):825 – 836, 1998.

[49] W. M. Yen. Effect of Local Item Dependence on the Fit and Equating Performance of the Three-Parameter logistic Model. *Applied Psychological measurement*, 8:125 – 145, 1984.

[50] A. L. Zenisky, R. K. Hambleton, and S. G. Sireci. Effects of Local Item Dependence on the Validity of IRT Item, Test, and Ability Statistics, MCAT Monograph, 5. 2001.

[51] Guansong Zhang, Bryan Carpenter, Geoffrey Fox, Xiaoming Li, Xinying Li, and Yuhong Wen. PCRC-based HPF compilation. In *10th International Workshop on Languages and Compilers for Parallel Computing*, 1997. To appear in Lecture Notes in Computer Science.

BIOGRAPHICAL SKETCH

Han-Ku Lee

Han-Ku Lee received a B.S. in computer science from the Iowa State University, Ames, Iowa, in 1996, and a B.S. in mathematics from Sogang University, Seoul, Korea, in 1993. He received a M.S. in computer science from Syracuse University, Syracuse, New York, in 1999. Finally, He will receive a Ph.D. in computer science from the Florida State University, Tallahassee, Florida, in 2003.

During his Ph.D. years, he have worked in Community Grids Lab, one of Pervasive Technology Labs at Indiana University, Bloomington, Indiana, in Computational Science and Information Technoloy (CSIT) at Florida State University, and in Northeast Parallel Architecture Center (NPAC) at Syracuse University as a research assistant.

He focused on enhancing High Performance Java Environments in distributed systems and programming support for High-Performance Grid-Enabled Application under the supervision of the Distinguished Professor and Director, Geoffrey C. Fox and a Research Scientist, Bryan Carpenter. HPspmd programming model he have researched is a flexible hybrid of HPF-like data-parallel language feature and the popular, library-oriented, high-performance grid-enabled, SPMD style, omitting some basic assumptions of the HPF model.

His present interests are in the integration and development of high-performance grid-enabled applications, the development of Web Service Compilation, and the improvement of Java Numeric Environment on Grid Computing environments.