Runtime Oriented HPF Compilation

Xiaoming Li

CRPC-TR97694
February 1997

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

submitted May 1997

Runtime Oriented HPF Compilation

— Lecture notes for CIS/CPS 600, Fall 1996

Xiaoming Li [1]

Northeast Parallel Architectures Center
at Syracuse University
111 College Place
Syracuse, NY 13244
lxm@npac.syr.edu

February 13, 1997

[1]Visiting scholar from Harbin Institute of Technology, China

Contents

List of Figures

List of Tables

Preface

As title of this manuscript suggested, materials presented here are from my lecture notes for an experimental graduate course, "Data Parallel Compilation Systems", offered in Syracuse University in Fall, 1996.

Teaching a parallel compiler course might be a challenge, especially when the instructor tries to deliver something that is being developed in an ongoing R&D project. Indeed, much of the materials covered in this course reflect how we have being doing for the PCRC (Parallel Compiler Runtime Consortium) project supported by an ARPA contract. Turning engineering practice into teachable materials is non trivial.

HPF has been around for a while. One of good things about HPF is it has an elegant data distribution model, which is ideal for teaching data parallel compiler courses.

Depending on where to draw the line between compiler and runtime, there are different approaches to building a data parallel compilation system. We emphasize *runtime oriented* approach, which makes this course different from typical parallel compiler courses.

Twelve 2.5 hour lectures had been delivered into this course. This document is then organized in twelve chapters accordingly.

Chapter 1 gives an overview of this course. Since this is considered as an experimental course, a thorough explanation on topics to be covered is given in the first lecture, together with what are expected from students.

Chapter 2 is an introduction to HPF language, since it will be our primary reference of language when talking about data parallel compilation systems. No effort is made to give an extensive discussion on this language.

Chapter 3 is an introduction to MPI, since it will be our underline communication system for building runtime system, an indispensable component of a data parallel compilation system.

Chapter 4 provides a quick tour on what we mean by SPMD node programs. Purpose of this lecture is to allow students to have an early understanding on what a compiler should produce, and how the compiler interplays with runtime.

Chapter 5 discusses an HPF-like data distribution model in detail. In particular, local storage allocation schemes and global/local index mappings are studied under a notion of alignment-distribution arithmetic.

Chapter 6 is a thorough discussion on design of a *distributed array descriptor*, **DAD** for short.

Chapter 7 introduces basic runtime functions for a data parallel compilation system supporting regular data distributions and data access patterns. Chapters 6 and 7 are sisters, together they describe an interface between compiler and runtime.

Chapter 8 develops a technique for determining communication requirement between two distributed regular array (section) references. In particular, we classify the communication requirement into three kinds: *no communication*, *shift communication*, and *remap communication*. Conditions for each of the kinds are derived. These conditions are tested in compile time in order to generate efficient node program.

Chapter 9 is a discussion on compilation process. Since students are expected to complete a small operational compilation system as a term project, this discussion is focused on a particular scheme associated with a particular subset of `HPF`, `minihpf`, the target of the project.

Chapter 10 includes an overview for some compiler construction tools, `HPFfe, Sage++`, and `SUIF`. This lecture is intended to help students to do the project, as well as giving them some idea on what kind of tools are available for compiler constructions. Since they have better working knowledge with some of the tools, Erol Akarsu and Guansong Zhang gave an introduction on SUIF and Sage++, respectively. I thank them for the help.

Chapter 11 addresses compilation issues related to multidimensional arrays, in particular, the problems we encounter when dealing with permuted dimensions, rank-reduced sectioning, etc.

Chapter 12 is a primitive discussion on some of the issues related to compiling irregular problems.

Beyond the lecture notes, I conclude with some remarks on this kind of course, and give some thoughts for further improvement.

About fonts used in this document, although it's hard to keep consistency, the following is more or less practiced. `Typewriter font` is used for things having meaning related to "constant", such as `MPI` constants, a piece of program code, etc.; *emphasized font* is for emphasis of some idea or the first occurrence of some terms. Nevertheless, another way to see the use of *emphasized font* and `typewriter font` is that they are used for any non English words first (or whenever) appearing in ordinary paragraphs. **Bold font** is for introduction of important notion or concept.

I thank Don Leskiw for proposing the idea of my teaching this course, thank professor Geoffrey Fox, director of NPAC, and professor Carlos Hartmann, director of CIS Department for supporting and approving this experimentation. I also thank the three students who had survived through this course, Grant Ingersoll, Xinying Li, and Erol Akarsu. At last but not least, discussions with my research fellows in Parallel Software Systems Group of NPAC, Bryan Carpenter, James Cowie, Guansong Zhang, and Yuhong Wen are always invaluable.

Xiaoming Li

Syracuse, New York
January 31, 1997.

Chapter 1

Course title explained: an overview of the course

The course is designated as "**Data Parallel Compilation Systems — Design and Implementation**", offered in Syracuse University in Fall, 1996, with code CIS/CPS 600.

Since this is the first time such a course is offered in Syracuse (and I'm doing it the first time), I consider it experimental. Thus, the first lecture is devoted to an explanation of the course. The explanation is around the three phrases in the title: *design and implementation*, *data parallel*, and *compilation systems*.

1.1 "Design and implementation"

The course is conceived to bias to construction of data parallel compilation systems, instead of some broad coverage of various analysis and program transformation theories/techniques in parallel compiler construction.

As a designed outcome, a student finished with this course is expected to have some working knowledge of a data parallel compilation system, from system organization and implementation point of view. He/she should be confident to be an active person in a team to develop a data parallel compilation system. He is not expected, out of this course, to have broad/deep knowledge on contemporary parallel compiler research results.

As a measure in this spirit, students are expected to do a project to build some primitive data parallel compilation system in this course. This project will count 70% towards final grade.

This sets basic tone of the materials to be delivered.

1.2 "Data parallel"

The term *data parallel* or *data parallelism* has been used in various contexts. Roughly, we often see them as in

- *data parallel* applications
- *data parallel* programming languages
- *data parallel* computation models

I don't know any more formal and well accepted definition for *data parallelism* than the following from HPF handbook [6], which should more or less convey the message.

> The same operation may be applied to many elements of a data object at the same time.

1.2.1 Data parallel applications

Classical references to data parallel applications include

- Partial differential equation (PDE) solvers
- Image processing
- Linear algebra operations

Typical "non" data parallel applications may be

- Text processing, such as many web applications.
- Many AI applications, such as games.

Data parallel applications usually exist as core components of higher level applications, such as

- Pattern recognition usually needs to apply image processing techniques as low level process.
- Manufacturing process simulation, such as chemical plants, usually consists of a set of coordinated data parallel components.

1.2.2 Data parallel programming languages

The following is a reasonable characterization for data parallel languages.

> Computer languages that provide facilities for programmer to express data parallelism in an application.

For instance, they should provide some kind of data parallel operation constructs, such as in Fortran 95,

```
REAL A(100), B(100)
A = B + 2
```

which usually means all elements of A are updated *in parallel* by corresponding elements of B plus 2. Another example in Fortran 95 is,

```
FORALL (i = 1:100) A(i) = B(i) + i
```

Once again, the 100 elements of A are supposed to get updated in parallel.

HPF (High Performance Fortran) provides more ways for expressing data parallelism, such as,

```
!HPF$ INDEPENDENT
      DO i = 1, 100
        A(i) = B(i) + 1
      END DO
```

which claims that the 100 iterations can be executed *in any order* — a stronger assertion than *in parallel*.

We note that ordinary DO loops in FORTRAN 77 is not a parallel operation construct, though people may *exploit* data parallelism out of it.

One more example in C* (TMC, connection machines),

```
shape [4][8]Aname;     /* specify a shape of rank 2 */
int:Aname a, b;        /* define two parallel variables of certain shape */
a = b + 2;             /* parallel assignment statement */
```

A data parallel programming language may also provide some data mechanism for programmer to tell compiler how data objects are best distributed from application point of view. For example in HPF,

```
PROCESSORS P(4)
DISTRIBUTE A(BLOCK) ONTO P
```

means to distribute elements of array A onto 4 processors in a blocky fashion.

1.2.3 Data parallel computation models

There are two basic models for data parallel computation. Architectural concept is depicted as in Figure 1.1.

- Host-node (master/slave) — host controls/coordinates program execution, invokes/suspends node processors' activities. Host and nodes usually run different programs.
- SPMD — Single Program Multiple Data — the same program runs in every processor. Processors *coordinate* themselves via the program.

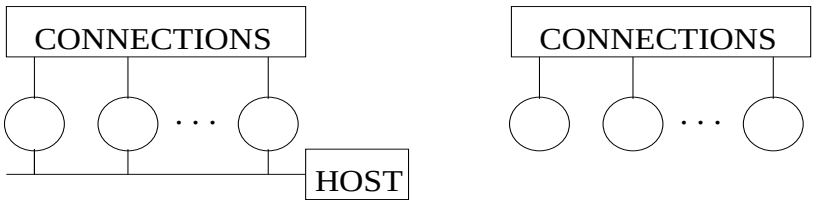


Figure 1.1: Data parallel computation models

The following skeletons represent a general framework of host-node model in some existing systems. As we can see, they are 'isomorphic', though from two totally different systems.

Host-node model in Express

```
    Host program                Node program

   ...                          ...
invoke node program          receive data
   (KXLOAD)                  from host (KXREAD)
   ...                          ...
send data to node            perform computation
program (KXWRIT)             together with other
   ...                       nodes (KXCOMB,etc)
collect result from             ...
node program (KXREAD)        send result to host
   ...                          (KXWRIT)
                                ...
```

Master/slave model in PVM

```
    Master program              Slave program

   ...                          ...
invoke slave program         receive data
   (pvm_spawn)               from host (pvm_recv)
   ...                          ...
send data to node            do some work on the data
program (pvm_send)           together with other
   ...                       nodes (pvm_reduce, etc)
collect result from             ...
node program (pvm_recv)      send result to host
   ...                          (pvm_send)
                                ...
```

We will restrict our discussions to `SPMD` for rest of the course.

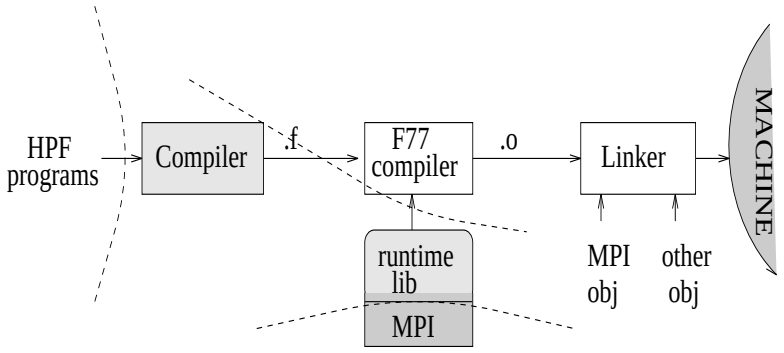


Figure 1.2: Architecture of a data parallel compilation system

1.3 "Compilation systems"

We want to distinguish something from *compiler*. For lack of a better term, *compilation system* is picked. The message is: a compilation system includes two equally important components , compiler and supporting library, in particular, the *runtime*.

In principle, code in runtime may be part of compiler's output, presumably implying higher performance. This view despises importance of runtime, which is not what we want to take. We want to emphasize the importance of runtime in a compilation system. The biggest advantage of placing runtime (in general, library support) in a distinct position perhaps is it offers a divide-conquer strategy for effective implementation of programming language systems. This strategy allows a system to be constructed in a less painful (even joyful) process, which is especially needed for academic research projects. Technically, strong library support turns many tedious expression manipulations during compile time into value calculation at runtime. We also see this is an appropriate approach to teaching an introductory course like this one.

With this approach on mind, we may view the construction of a data parallel compilation system as a process of understanding and implementing the following three interfaces, shown as in Figure 1.2.

1. The language.

   This is the interface between compiler writer and application programs. We assume some kind of HPF language as our target in this course. (Thus, we'll spend one lecture introducing HPF.)

2. The underline communication system that runtime is based on.

   This is an interface used by runtime writer to implement various data movement functions. We assume a subset of MPI (Message Passing Interface) for this course. (Thus, we'll spend another lecture for MPI.)

3. The architecture of node programs.

   Among things a compiler has to take care to produce operational node programs, there is an interface both compiler and runtime writers must observe.

   This interface will be the focus point of the course. It generally contains two different aspects, namely *data interface* and *functional interface*. Data interface specifies how compiler tells runtime functions about distributed data objects; functional interface is a specification of calling sequences to runtime that compiler must produce in the node program in order to accomplish certain functions. Once compiler conforms to the interface, runtime should produce expected result in return.

1.4 Specific topics to be covered

We assume compiler translates HPF program into SPMD (Single Program Multiple Data) type node program in FORTRAN 77. This kind of source-to-source translation scheme is not only popular in academic environment, but also found in some commercial systems, such as PGI HPF compiler, NAG Fortran 90 compiler.

Besides introductions to HPF and MPI, the course will be stretched over the following three dimensions.

1.4.1 Runtime system design

What's **runtime** ? Yes, we've been heard it a lot in different context, such as *runtime environment*, *runtime system*, *runtime functions*, or simply *runtime*, for both parallel sequential language systems. Once again, there seems no precise definition. For us, we view runtime as a set of procedures and some environment data, existing in a form of a library, callable from SPMD **node** programs.

So, what are in a runtime ? In general, this is design/implementation dependent. Nevertheless, we'll see the following in most implementations.

- Distributed data management functions.

  An array is distributed onto processor grid in an HPF program. Each processor gets part of that array. Since data elements in an array are usually related in the computation defined in HPF program, a processor running a node program should know how the array is distributed, in particular, what part of the array it owns. We'll define something called *distributed array descriptor*, or `DAD` for short, to carry the information. Runtime will have some functions to create and modify the DAD.

- Global/local index mapping functions.

  HPF arrays are declared in a global index space, and distributed to processors, in terms of local arrays in node program. The mappings from global index to local index and from local index to global index are basic operations a compiler must perform in

generating node programs. Some functions are provided in runtime to facilitate these operations.

- Data movement functions.

  We'll discuss collective communication routines only. By collective communication, we mean all processor participate in the operation, though may play different roles. A typical collective communication is broadcasting, where one sends and others receive.

- Miscellaneous, including initialization, etc.

1.4.2 Program transformations

Compiler's job is to perform program transformation. In our case, it turns an HPF program to an SPMD node program, such that a collective execution of multiple copies of the node program achieves the same effect as specified in the HPF program.

One major aspect of the transformation is to deal with parallelism. There are two types of parallelism. One is *explicit parallelism*, which is expressed by programmer in parallel programming language such as array assignment statements in Fortran 95; the other is *implicit parallelism*, which is not expressed in program but discovered by parallelizing compiler. We shall mainly address explicit parallelism in this course. In particular, we identify the following 9 language constructs in HPF that give rise to explicit parallelism.

- Array (section) assignment statement;
- `WHERE` statement (conditional array assignment);
- `WHERE` construct;
- `FORALL` statement;
- `FORALL` construct;
- `INDEPENDENT FORALL`;
- `INDEPENDENT DO` loop;
- Elemental intrinsics;
- Transformational intrinsics.

Among them, FORALL statement is of key importance, since others may be equivalently (or with some possible loss in performance) transformed to it. Figure 1.3 shows a suggested transformation map among the 9 language elements.

Also, array assignment is the most popular language features used in typical HPF programs. Thus, our emphasis will be placed on these two items.

Another aspect of the transformation is to deal with different types of data access patterns, classified as *regular* and *irregular* ones. By regular problem, we mean array elements are designated by direct index variables, such as `A(i)` and `A(2*i+1)`. By irregular problem, we mean array elements are indexed by another array (must be of rank 1, though), *indirection array*, such as `A(V)` and `A(V(i)+j)`, where V is some integer array. We'll discuss both of them, but with emphasis on regular one.

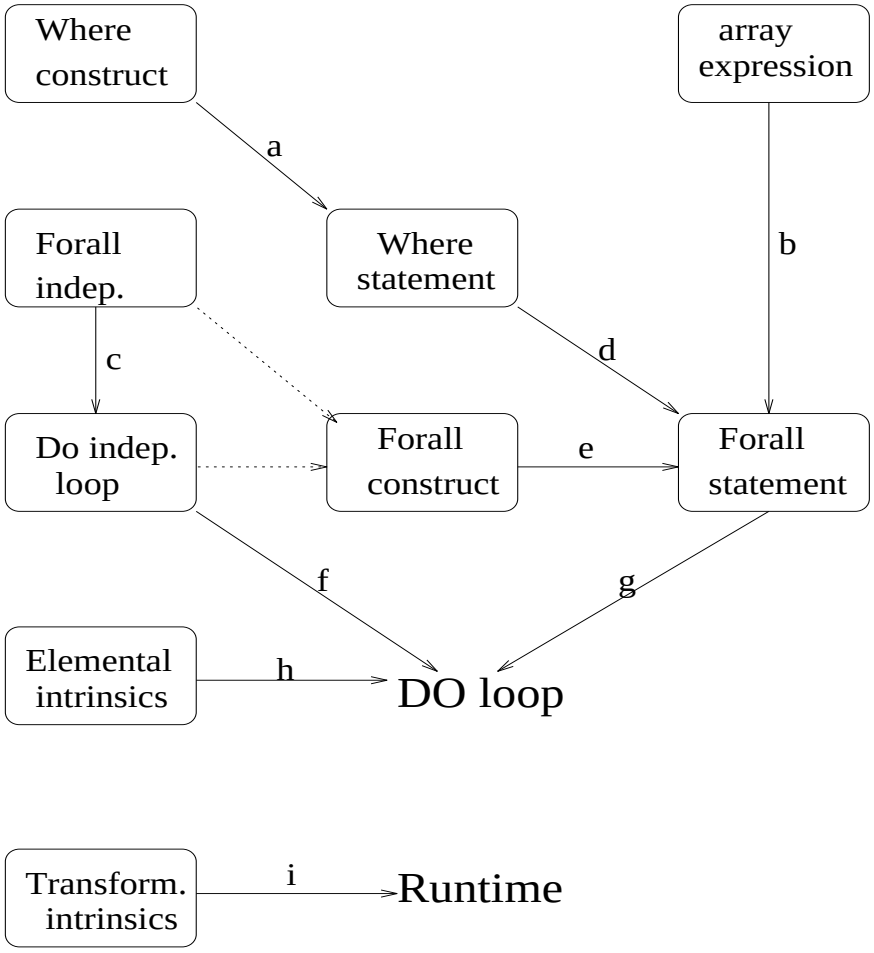


Figure 1.3: A possible transformation map for explicit parallelism

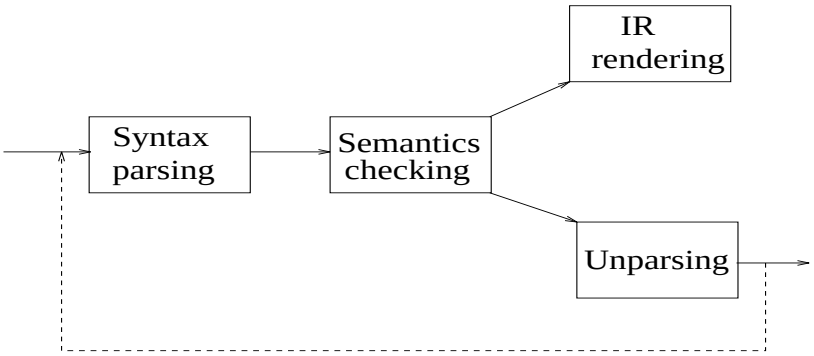


Figure 1.4: Modules and operations of HPFfe

As identified previously, the third interface, namely node program structure, will be our focus point. This implies our discussion will be 'node program oriented', i.e., we will often show how a node program segment looks like, corresponding to some HPF statement. This approach is made feasible largely due to our emphasis on runtime support; otherwise the node program would be too messy to be discussed in class. This approach should enable students to readily write down an operational node program manually for a given HPF program, once he understands the interface.

1.4.3 Introduction to some compiler construction tools

As mentioned in section 1.1, students are expected to do a project in this course. This project will contain implementation of a set of runtime functions as well as compiler transformations needed for a tiny subset of HPF language, called **minihpf**.

Students will be exposed to some contemporary compiler construction tools and infrastructures. This will be directly helpful for them to do the project, as well as to gain some knowledge on what kinds of tools available out there. More specifically, we'll spend some time to discuss

- `HPFfe`, a full featured HPF 1.0 front-end constructed by a joint effort among NPAC at Syracuse University, PACT Lab at Harbin Institute of Technology of China, and PACT Group at Peking University of China.

  There are four modules in HPFfe as shown in Figure 1.4, plus a class library for writing compiler transformations.

- `Sage++`, a C++ class library for facilitating program transformation, built by Indiana University, which now can work in concert with HPFfe.

- `SUIF`, a general compiler construction infrastructure built by Stanford University.

Chapter 2

An introduction to HPF language

As mentioned in the first lecture, High Performance Fortran (HPF) type of language will be the object of our data parallel compilation system. We discuss HPF 1.0 in this lecture, assuming moderate knowledge of FORTRAN 77. The purpose of this lecture is not to train students to become good HPF programmers, though they are supposed to do a homework writing an HPF program. Instead, we highlight some of important language features that are most often encountered in compilation of a data parallel language.

Some slides presented in this lecture are directly taken from A.C. Marshall's "HPF for Fortran 90 Programmers — 3 day Lectures Guide" and Charles Koelbel's tutorial "High Performance Fortran in Practice".

2.1 A brief history

HPF is an extension to Fortran language, designed to deliver *higher* performance in scientific and engineering computing than what ordinary Fortran can do with automatic parallelizing compilers.

The idea was initiated around 1990 with a project, Fortran D, carried out in Rice University and Syracuse University, USA.

A forum, HPFF, was established to discuss technical issues in defining this language, which gathers representatives and inputs from industry, national laboratories, and universities.

HPF 1.0 specification was released May, 1993, which is mainly based on ISO Fortran language standard (Fortran 90) issued in 1991. It should be noted that both Fortran standard and HPF are evolving — after HPF 1.0, ISO is releasing Fortran 95 standard, which incorporates some of the HPF features; HPF 2.0δ was released in October, 1996. For more information, visit `http://www.crpc.rice.edu/HPFF/home.html`. In what follows, our discussion will be targeted on Fortran 95 and HPF 1.0.

Currently, there are several vendor implementations of HPF 1.0, including IBM, DEC, and PGI, etc.. Experiments show there is a lot of room left for improving performance.

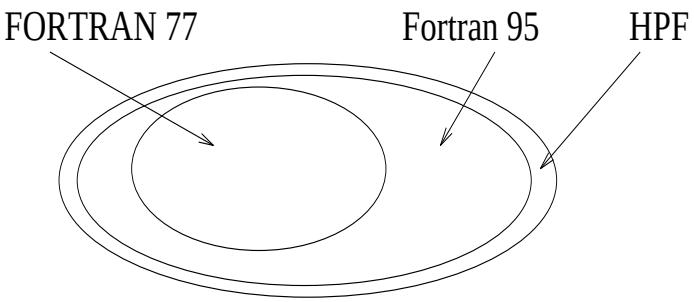


Figure 2.1: $HPF \supset Fortran\ \ 95 \supset FORTRAN77$

2.2 Language overview

The inclusion relation expressed in Figure 2.1 is generally true, including relative size of each set.

The following are main features in Fortran 95 minus FORTRAN 77. Once again, there are probably more than one way to capture this difference. I'll just take whatever I consider interesting for the purpose of this course. A list is given first, followed by subsections illustrating each item in the list, respectively.

- Free source form. Fortran 95 promotes so-called "free source form", while still supports the traditional "fixed source form".
- Array operations, masked array operations
- `FORALL` statement and construct
- More intrinsics, elemental and transformational
- Pointers, allocatables
- User-defined types, operators

see `ftp://ftp.ncsa.uiuc.edu/x3j3/96-007r1/ps` for complete information. For people who know FORTRAN 77, James F. Kerrigan's book [10] seems the best book for "migrating to Fortran 90(95)".

2.2.1 Free source form

The following points may be noted for free source form.

- no 7-72 column restriction any more, now any where from 1-132
- more than one statement allowed in one line, separated by ';'
- up tp 31 alphanumeric characters for a name

- new way for line continuation — '&' at end.
- comments by '!'

Here is an example,

```
0123456789012345678901234567890123456789012345678901234567890123456789...

! This is a complete F95 program, demonstrating various source
! form features
PROGRAM FOO
   INTEGER A,B,This_is_a_long_name
   A = 1; B = 1                          ! two statements in one line
   This_is_a_long_name = A + B           ! initialize long name
       This_is_a_long_name = A + B +  &
                             This_is_a_long_name   ! continuation
     ! look at the second ampersand in next continuation
       This_is_a_long_name = A + B + This_i&
     &s_a_long_name
 print *, This_is_a_long_name
end
```

2.2.2 Array operations

From data parallel compilation point of view, this is the major feature of Fortran 95. Examine the following program,

```
   PARAMETER (N=100)
   REAL, DIMENSION (100,100) :: A,B   ! the "attribute" way of declaring vars
!     In F95, possible attributes associated with a name include
!     data type, DIMENSION, POINTER, PARAMETER, INTENT, ALLOCATABLE, ...
   REAL C(100)
   ...
   A = B  ! equivalent to A(1:N,1:N) = B(1:N,1:N)
   A(1:N-1:2,1) = B(1:N-1:2,N) + B(2:N:2,N)
   C(2:N) = C(1:N-1)
! note this is different from
   DO i = 2,N
     C(i) = C(i-1)
   END DO
```

Key semantics of array operations:

> every element at right hand side is evaluated before assignment can happen; and shape conformance must be observed.

```
   C(1:50) = C(51:100)   ! shape conformed
   C(1:50) = scalar      ! also considered conformed
   C(1:50) = C(1:51)     ! this is not conformed, semantically invalid
```

Also note,

```
C(1:N-1) = C(2:N)
```

and

```
DO i = 1,N-1
  C(i) = C(i+1)
END DO
```

happen to achieve the same effect, but this *does not* mean array operation syntax is just a convenient notation for DO loop.

Since array and array section operations are main focus of data parallel compilation, we excerpt the following syntax rules from Fortran 95 specification.

```
array-section     is  name [(section-subscript-list)]
section-subscript is  subscript
                  or  subscript-triplet
                  or  vector-subscript
subscript         is  scalar-int-expr
subscript-triplet is  [subscript] : [subscript] [: stride]
stride            is  scalar-int-expr
vector-subscript  is  integer-array-of-rank-one
```

where for missing subscripts, declared values are assumed; 1 is assumed for missing stride.

Let's see some examples. For the declaration,

```
REAL, DIMENSION(100) :: A
REAL, DIMENSION(0:99,100) :: B
```

We conclude,

```
A(5:20) --- a section of shape (/16/), or length 16.
A(:) --- the same as A
A(2:) --- the same as A(2:100)
A(:98) --- the same as A(1:98)
A(1:99:2) --- a strided section of shape (/50/)
A(::2) --- the same as A(1:100:2), which in turn is the same as A(1:99:2)
A(5::5) --- the same as A(5:100:5), a strided section of shape 20.
A(99:1:-2) --- a section of 50 elements; different from A(1:99:2).
A(2:1) --- an empty section, or length 0 section.
A(5:5) --- a section of length 1.
```

All of above are considered as "rank one" array sections, while A(5) is considered as a scalar, or rank zero. Shape of an array or array section is always denoted by an integer array of rank one (also called an integer vector).

```
B(:10,100) --- the same as B(0:10,100), a rank one section of shape (/11/).
B(1:51:2,:) --- a rank two section of shape (/26,100/), size 2600.
B(3:2, 1:100) --- a rank two section of shape (/0,100/), size 0.
```

If `V = (/2,1,1,3/)`, then A(V) refers to a section of shape (4), namely A(2),A(1),A(1),A(3). Thus, it's valid to have B(1:4,1) = A(V), but not other way around.

Masked array operations.

```
where-statement   is  WHERE (mask-expr) assignment-stmt

where-construct   is  WHERE (mask-expr)
                         [ where-body-construct ]...
                      [ELSEWHERE
                        [where-body-construct] ...]
                      END WHERE

where-body-construct     is  assignment-stmt
                         or  where-stmt
                         or  where-construct

mask-expr                is  logical-expr
```

We note the "body-construct" is very much restricted.

2.2.3 FORALL statement and construct

In general, `FORALL` statement is a special case of `FORALL` construct.

```
forall-construct         is  FORALL forall-header
                                [forall-body-construct] ...
                              END FORALL

forall-body-construct    is  assignment-statement
                         or  where-stmt
                         or  where-construct
                         or  forall-stmt
                         or  forall-construct

forall-header            is  (forall-triplet-spec-list [,scalar-mask-expr])

forall-triplet-spec      is  index-name = subscript:subscript [:stride]

forall-stmt              is FORALL forall-header assignment-statement
```

Semantically, `FORALL` statement is a proper superset of array assignment statement, though we note its `body-construct` is quite restricted.

2.2.4 More intrinsics

From data parallel processing point of view, there are *elemental* intrinsic functions and *transformational* intrinsic functions.

Elemental intrinsics are those which may take whole array or array section as argument and perform the function on each element of the argument independently. For instance, `SIN(A(1:20:2))` will return an array of 10 elements that are *sin()* of the 10 elements of A(1:20:2), respectively. From compiler point of view, no communication will be required to perform elemental intrinsics, though argument array may be distributed among processors.

Transformational intrinsics also take whole array or array section as argument, but each returned value normally depends on a collection of input elements, such as matrix multiplication `MATMUL()`. Communication is usually required for transformational intrinsics on distributed array argument.

2.2.5 Pointers, allocatables

Pointers and `ALLOCATE` statement are something making Fortran closer to C, though with some subtle differences.

- `POINTER` is a possible attribute of some data objects
- A `POINTER` is associated with its `TARGET`, which is also an attribute of some data objects, in two ways:

  via *pointer assignment statement*, or via `ALLOCATE` statement.
- Referencing a pointer is to referencing its target.
- "The pointer itself" is referenced by pointer assignment statement.

Examples,

```
      REAL, POINTER :: P,Q
      REAL, TARGET :: A,B
      REAL C
      A = 1; B = 2
      P => A           ! P is associated with A via a point assignment-stmt
!     Q => C           ! This is _invalid_, since C has no TARGET attribute
!     Q = P            ! This is not making Q point to the same target as
                       ! P does. In fact, it is semantically invalid, since
                       ! Q is not associated with any thing yet. We have to do
      Q => P           ! to make two pointers associated with the same TARGET
! however,
      P = B            ! will have the same effect as A = B
      B = P - 1        ! will set B to 0, leave A being 1 intact.
      Q => B           ! Q is now associated with B
      Q = P            ! OK now, the same as B = A
      ALLOCATE(P)      ! will associate P with a _new_ target of REAL type.
                       ! this newly allocated target will be referenced via P
      P = 3            ! Note, this has nothing to do with A, or B.
      C = A + B + P    ! will make C = 6
```

2.2.6 User-defined types, operators

Like many modern languages, Fortran 95 allows user-defined types. The following defines a type RECORD with three fields.

```
TYPE RECORD
  CHARACTER (LEN=20) NAME
  INTEGER :: AGE,ID
END TYPE RECORD
```

Then one can declare some variable of type RECORD and operate on the variable as

```
TYPE (RECORD) :: HEAD
HEAD%NAME = 'JOHN SMITH'
HEAD%AGE = 42
HEAD%ID = 234
```

The next example shows how to build up a linked list via user-defined type.

```
TYPE NODE
  INTEGER :: VALUE
  TYPE (NODE),POINTER :: NEXT
END TYPE NODE

TYPE (NODE) FIRST_NODE, SECOND_NODE
FIRST_NODE%VALUE = 1
FIRST_NODE%NEXT => SECOND_NODE
SECOND_NODE%VALUE = 2
SECOND_NODE%NEXT => NULL ()
```

HPF has more stuff than Fortran 95 mainly in terms of *directives*. Five major directives are

- PROCESSORS — declares a processor grid that some data object will be distributed onto;
- TEMPLATE — declares a logical template for alignment between data objects;
- ALIGN — aligns an array to a template explicitly;
- DISTRIBUTE — distributes a template onto a processor grid;
- INDEPENDENT — indicates iterations of a DO loop have no 'dependence' relation. It can also be applied to some FORALL statements.

  While much of the course will be studies around the first four directives, we address some points about INDEPENDENT in the following.

  For instance, if the programmer knows foo() will not modify other_arg, he may write,

```
!HPF$ INDEPENDENT
      DO i = 1, N
         CALL foo(A(:,i),other_arg)
      END DO
```

Note, without the `INDEPENDENT` directive,, a parallelizing compiler has to assume the DO loop be executed sequentially (if it does not know whether `other_arg` will be modified by `foo()`.) Also note, this parallelism can not be expressed by `FORALL` ! (due to the restrictions for its body statements)

Of course, not every DO loop can be prefixed by an `INDEPENDENT`, such as,

```
DO i = 2, N
  A(i) = A(i-1)
END DO
```

Although `FORALL` is a data parallel construct, not every `FORALL` can be prefixed by an `INDEPENDENT`, either, such as,

```
FORALL (i = 2:N) A(i) = A(i-1)
```

`INDEPENDENT` says: "it can be done *in any order*", which is different from "it is to be done *in parallel*". To understand this, let's assume `N=4`, `A(1:N) = (/a,b,c,d/)`. Then the result of the `FORALL` is `A(1:N) = (/a,a,b,c/)`, while

```
!HPF$ INDEPENDENT
   FORALL (i = 2:N) A(i) = A(i-1)
```

may potentially result in any one of:

```
A(1:N) = (/a,a,a,a/);  A(1:N) = (/a,a,a,c/)
A(1:N) = (/a,a,b,b/);  A(1:N) = (/a,a,b,c/)
```

which is not well defined.

Finally in this lecture, we present a complete sample HPF program.

```
 PROGRAM LAPLACE

 INTEGER N,Nsteps
 PARAMETER(N=16)
 PARAMETER(Nsteps = 100)

 INTEGER i,j,k
 REAL A(N,N)

!HPF$ PROCESSORS PROC(2,2)
```

```
!HPF$ TEMPLATE TEMP(N,N)
!HPF$ DISTRIBUTE TEMP(BLOCK,BLOCK) ONTO PROC
!HPF$ ALIGN A(i,j) with TEMP(i,j)

      A(1:N:N-1,1:N) = 8.0
      A(2:N-1,1:N:N-1) = 8.0

      DO k = 1, Nsteps
        FORALL (i=2:N-1,j=2:N-1)                                        &
     &     A(i,j) = (A(i,j-1) + A(i,j+1) + A(i-1,j) + A(i+1,j))*0.25
      END DO

      print *, 'Final grid looks like'

      DO I= 1, N
        PRINT 101, (A(I,J),J=1,N)
      END DO

101   FORMAT(16F5.2)
      END
```

Homework: Write an HPF program solving system of linear equations using Gaussian elimination method. Run your program at least on 4 processors.

Chapter 3

An introduction to MPI

As planned in the first lecture, MPI will be our underline communication system for building runtime libraries. This lecture discusses basics of MPI. Some of the slides presented in this lecture are directly taken from William Gropp's "Tutorial on MPI: The Message-Passing Interface".

3.1 A brief history

MPI stands for **Message-Passing Interface**, which is an interface specification for message-passing libraries, produced by MPI Forum (MPIF).

Activities of MPI Forum was initiated in a workshop in April, 1992. As the case for HPF, representatives from academia, industry, and national labs have been participating in the forum.

MPI specification 1.0 was released in May, 1994.

MPI specification 1.1 was released in June, 1995.

An effort towards MPI-2 specification is ongoing.

There are many commercial and public-domain implementations of MPI available, and many parallel libraries have been reported being based on MPI. Please visit `http://www.erc.msstate.edu/mpi/implementations.html` for more information.

Contrasting the two important standardization effort, HPF and MPI, in high performance computing community in last years, it looks like MPI, though started late, is more successful than HPF at the moment.

3.2 A taste of MPI

MPI consists of 125 functions and many symbolic constants, extracted and inherited from various preceding message passing systems. I don't intend to give a systematic introduction to MPI (there are many good ones out there.) Rather, I'll present some features of MPI via examples, just to get students started writing and running MPI programs. `MPICH`, one of public domain implementations of MPI, is used for practice.

3.2.1 Basic MPI functions

Although there are 125 functions in total (more will be added in MPI-2), 6 of them are fundamental. It's like the instruction set of a modern computer — although there are perhaps 100 instructions, only a few are sufficient to do whatever the 100 can do, of course with inconvenience. We discuss this 6 functions in this section.

```
MPI_Init(int *argc, char ***argc)
```

starts MPI. Arguments of this function are designed for possible pass of command line arguments to MPI environment. In many cases, programmers do not have to worry about them.

```
MPI_Finalize(void)
```

exits MPI. Any message passing library must have these two functions, or something alike. Then we can write our first MPI program.

```
#include "mpi.h"

int main(int argc, char *argv[]) {
   MPI_Init(&argc,&argv);
   printf ("Hello, world ! This is from an MPI program\n");
   MPI_Finalize();
}
```

Note: inclusion of "mpi.h" is always required, which provides basic MPI definitions and types. One important distinction of this program from the well known "Hello, world !" program presented in almost every programming book is that you may see multiple copies of this messages printed out.

Two of the first questions asked in a message parallel program are: how many processors are participating ? and who am I ? MPI provides two functions for them.

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

returns, in *size*, the number of processors within communicator *comm*.

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

returns, in *rank*, the abstract name (an integer) of the calling processor within the communicator *comm*. In a communicator, ranks will be 0, 1, ..., size-1. (Note different meanings of *rank* in MPI and HPF.)

Communicator is a key concept in MPI. It may be considered as a *structured* set of processors that can communicate with each other.

I emphasize the word "structured" here, since the same set of processors may be structured in different ways, so belong to different communicators. There is always a default communicator, `MPI_COMM_WORLD`, that encompasses all processors upon starting an MPI program. One MPI program may involve several communicators.

Then we can have our second program.

```
#include "mpi.h"

int main(int argc, char *argv[]) {
   int size,rank;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&size);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   printf ("My name is %d in a size %d communicator.\n",rank,size);
   MPI_Finalize();
}
```

If you run it on four processors, you would get something like

```
My name is 0 in a size 4 communicator.
My name is 2 in a size 4 communicator.
My name is 3 in a size 4 communicator.
My name is 1 in a size 4 communicator.
```

Note the order of the messages is "random". So far, our two "message passing" programs have no messages passed among processors at all. In general, imagine a multicomputer system in which processors are participating in a computational task, we might have the perspective as in Figure 3.1.

MPI provides many communication functions for message passing. Among them, the following two are basic.

```
MPI_Send(void* out_buf, int count, MPI_Datatype datatype,
          int dest, int tag, MPI_Comm comm)
MPI_Recv(void* in_buf, int count, MPI_Datatype datatype,
          int source, int tag, MPI_Comm comm, MPI_Status *status)
```

where

- *out_buf, in_buf* — are base addresses of sending and receiving buffers, respectively.
- *count* — not number of bytes, but number of elements of *datatype*.

  For `MPI_Send()`, it's the actual number of elements sent.

  For `MPI_Recv()`, the actual count may not be known when called. A supplemental function `MPI_Get_count(MPI_Status*, MPI_Datatype, int*)` may be used to get actual received count, using the *status* and *datatype* returned from `MPI_Recv`.
- *datatype* — besides basic data types that are corresponding to C data types, one can define "derived data type" to facilitate communication (note MPI derived data type is more than C user-defined data type, as we'll see later.)
- *dest, source* — rank in a communicator. For convenience, `MPI_ANY_SOURCE` may be used at receiver side.

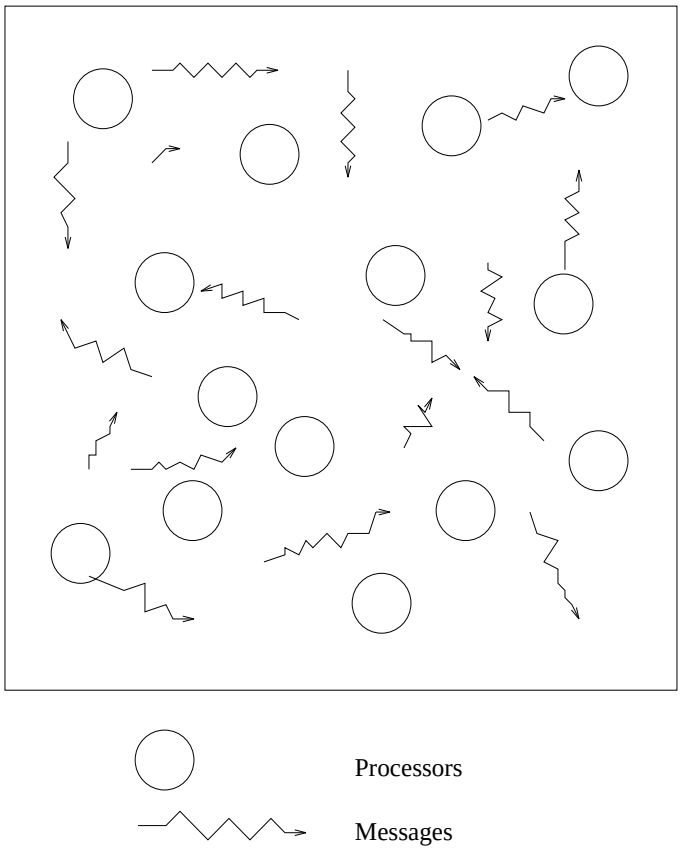


Figure 3.1: A view of message passing

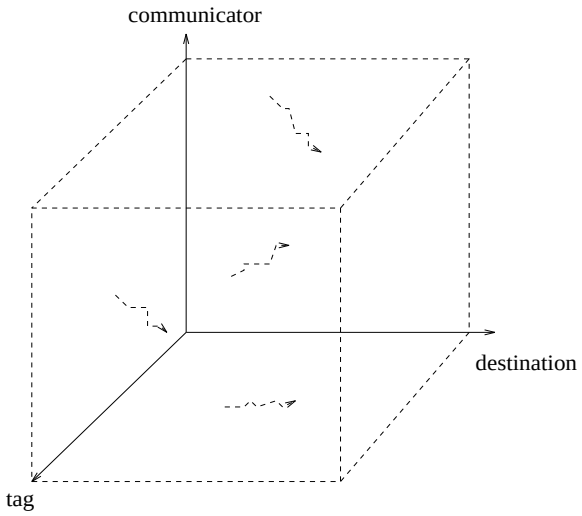


Figure 3.2: Partition of MPI message space

- *tag* — a method for partitioning message space. For convenience, `MPI_ANY_TAG` may be used at receiver side.

  status.`MPI_SOURCE` — gives sender's rank in case of `MPI_ANY_SOURCE` being used

  status.`MPI_TAG` — gives "sending tag" in case of `MPI_ANY_TAG` being used.

We see MPI provides three mechanisms to partition *message space*. Figure 3.2 provides some view.

- communicator: attributes a set of processors
- destination: partitions message space within a communicator
- tag: distinguishes messages arrived in one destination

They are very important and useful in developing parallel libraries. Note: these three things are not part of message that are received by a receiver. They are information to MPI system. In this sense, MPI system is like some agent between a set of senders and a set of receivers, gets messages from senders, using the three mechanisms to sort them, and deliver them to appropriate receivers.

MPI provides quite a few variants of the basic `send` and `receive` functions. In general, semantics of `send()` may be subtle. In particular, the order of arrivals of two messages sent from one processor to another can not be predetermined.

3.2.2 A "real" program

Now, we can write some "real" program: Jacobi iteration solving Laplace equations. We begin with the 6 functions discussed above, and then see how some of advanced MPI functionalities can help to make programming easier.

Problem specification: consider a domain of `SIZE x SIZE` elements. For comparison, we first recall how an HPF program looks like (the sample program in the last chapter.) We see a process pattern:

data input (we merely initialize them) $\longrightarrow$ compute $\longrightarrow$ output

Now we want to have an MPI program, such that a simultaneous execution of 4 copies of this program achieves the same effect as the HPF program.

The thought:

- we would like to keep the partition of problem domain the same as the HPF program, namely `(BLOCK,BLOCK)`. (we don't have to, though.) For convenience, we may designate the 4 portions of the domain as (0,0), (0,1), (1,0), and (1,1), respectively. Each portion is of `SIZE2 x SIZE2`, where `SIZE2 = SIZE/2`.

- Note the `MPI_COMM_WORLD` is presented as a linear list of processors: 0, 1, 2, 3. So we make some assignment:

  processor 0 handles portion (0,0)

  processor 1 handles portion (0,1)

  processor 2 handles portion (1,0)

  processor 3 handles portion (1,1)

- Since the "global" array has some boundary condition, each processor is actually responsible for updating `(SIZE2-1) x (SIZE2-1)` elements of its local array.

- There are communications between adjacent portions (processors), for instance, processor 0 needs to send a column to processors 1 for each sweep.

  Where should processor 1 keep the received column ? We extend the local array to include some "ghost area" to hold the incoming elements as depicted in Figure 3.3, thus we allocate:

```
float a[size2+1][size2+1];
```

  Then the computation part can be nicely written as

```
for (i=1;i<size2;i++)
   for (j=1;j<size2;j++)
      b[i][j]=(a[i-1][j]+a[i+1][j]+a[i][j-1]+a[i][j+1])*0.25
```

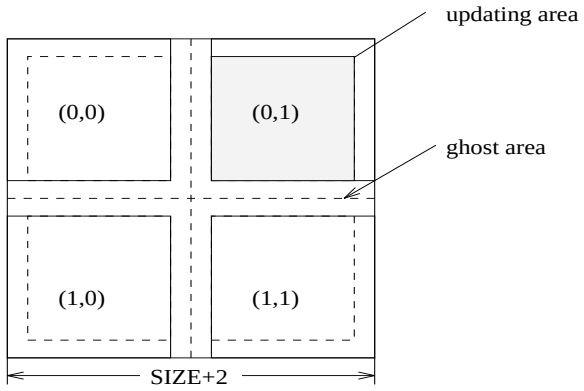


Figure 3.3: Use of ghost area

- Communications. Boundary rows and columns will be sent to adjacent processors, we would like to maintain a `send_buf[size2]` and a `recv_buf[size2]`, so that we can write something like

```
...
MPI_Send(send_buf,size2,MPI_FLOAT,col_neighbor,col_tag,MPI_COMM_WORLD);
MPI_Recv(recv_buf,size2,MPI_FLOAT,col_neighbor,col_tag,MPI_COMM_WORLD,&status);
...
MPI_Send(send_buf,size2,MPI_FLOAT,row_neighbor,row_tag,MPI_COMM_WORLD);
MPI_Recv(recv_buf,size2,MPI_FLOAT,row_neighbor,row_tag,MPI_COMM_WORLD,&status);
...
```

  in the program. To be valid to do this, we need

  - decide *col_neighbor* and *row_neighbor* based on the following observations.

    "my column neighbor has different right most bit from me"; and "my row neighbor has different the second right most bit from me"

    Thus,

```
col_neighbor = (myid & ~1) | (~myid & 1);
row_neighbor = (myid & ~2) | (~myid & 2);
```

    or simply,

```
col_neighbor =  myid ^ 1;  /* the exclusive OR. */
row_neighbor =  myid ^ 2;
```

  - get the proper elements into the send buffer before send,

```
for (i=0;i<size2;i++)
   send_buf[i] = a[i][send_col];
```

and store the contents of receive buffer into proper positions of the array after receive and before computation begins;

```
for (i=0;i<size2;i++)
   a[i][recv_col] = recv_buf[i];
```

(something similar for rows.) Thus, we need to compute *send_col*, *send_row*, *recv_col*, *recv_row*. The following observations give some hint.

```
  (00) ---> (01)                    (01) ---> (00)
col size2-1 to col 0                col 1 to col size2
  (00) ---> (10)                    (10) ---> (00)
row size2-1 to row 0                row 1 to row size2
  (11) ---> (01)                    (01) ---> (11)
row 1 to row size2                  row size2-1 to row 0
  (11) ---> (10)                    (10) ---> (11)
col 1 to col size2                  col size2-1 to col 0
```

Then,

```
send_col = (myid & 1)?1:size2-1;
send_row = (myid & 2)?1:size2-1;
recv_col = (myid & 1)?0:size2;
recv_row = (myid & 2)?0:size2;
```

A complete program based on the above analysis is

```
#include "mpi.h"
#define size 16
#define size2 size/2

int main(int argc, char *argv[]) {
   int n, myid, numprocs, i, j, nsteps=100;
   float a[size2+1][size2+1],b[size2+1][size2+1];
   float send_buf[size2],recv_buf[size2] ;
   int col_tag,row_tag,send_col,send_row,recv_col,recv_row;
   int col_neighbor,row_neig hbor;
   MPI_Status status;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);

   send_col = (myid & 1)?1:size2-1; send_row = (myid & 2)?1:size2-1;
   recv_col = (myid & 1)?0:size2; recv_row = (myid & 2)?0:size2;

   for (i=0;i<size2+1;i++) for (j=0;j<size2+1;j++) a[i][j] = 0;
   for (i=0;i<size2+1;i++) {a[size2-recv_row][i]=8; a[i][size2-recv_col]=8;}

   col_neighbor = myid ^ 1;  row_neighbor = myid ^ 2;
```

```
   col_tag = 5; row_tag = 6;

   for (n=0; n<nsteps; n++) {
      for (i=0;i<size2;i++) send_buf[i] = a[i][send_col];
      MPI_Send(send_buf,size2,MPI_FLOAT,
               col_neighbor,col_tag,MPI_COMM_WORLD);
      MPI_Recv(recv_buf,size2,MPI_FLOAT,
               col_neighbor,col_tag,MPI_COMM_WORLD,&status);

      for (i=0;i<size2;i++) a[i][recv_col] = recv_buf[i];
      for (i=0;i<size2;i++) send_buf[i] = a[send_row][i];
      MPI_Send(send_buf,size2,MPI_FLOAT,
               row_neighbor,row_tag,MPI_COMM_WORLD);
      MPI_Recv(recv_buf,size2,MPI_FLOAT,
               row_neighbor,row_tag,MPI_COMM_WORLD,&status);

      for (i=0;i<size2;i++) a[recv_row][i] = recv_buf[i];
      for (i=1;i<size2;i++) for (j=1;j<size2;j++)
         b[i][j] = (a[i+1][j]+a[i-1][j]+a[i][j+1]+a[i][j-1])*0.25;
      for (i=1;i<size2;i++) for (j=1;j<size2;j++) a[i][j] = b[i][j];
   };

   printf("Myid %d \n",myid);
   for (i=myid>>1;i<size2+(myid>>1);i++) { /* bit op has lower prec. */
      for (j=myid%2;j<size2+myid%2;j++) printf("%4.1f",a[i][j]);
      printf("\n");
   }
   MPI_Finalize();
}
```

Although the analysis we've conducted may not be the most elegant one, it's typical and reveals some of issues in message passing programming. We observe from the program,

- Similar in process pattern to HPF program, i.e., the process is roughly composed of

  $$\text{data initialization} \longrightarrow \text{computation} \longrightarrow \text{data output}$$

- There are some mapping between problem domain and processor domain to be done in "data initialization" part, which is sometimes tricky and in general non trivial. HPF program usually can avoid this.

- Explicit communication is interleaved with computation in "computation" part. We don't see this in HPF program.

- We obtain a global output from HPF program. But with MPI program, we have to "assemble" the outputs from each processors, unless some other special measure is taken.

A slight improvement: MPI provides a function `MPI_Sendrecv()`, such that the two consecutive `MPI_Send` and `MPI_Recv` can be literally replaced by one call to `MPI_Sendrecv()`.

Some major improvement: elimination of data packing and unpacking by MPI derived data type, and use of processor topology.

If we examine the analysis above closely, we would see it's not general for this problem, in addition to the trickness of computing *row_neighbor*, *col_neighbor*, etc. A question may be asked: what if more processors (say 3 x 3) are involved in the solving process ? We see

- A processor generally should communicate with 4 neighbors,
- But boundary processors only have 2 neighbors.

Thus, a more general treatment must be provided, if the program is of any use. For this, we present,

```
...
int main(argc,argv)
int argc; char *argv[];
{
   ... (define ordinary variables)
   MPI_Datatype c_column;
   MPI_Status status;
   MPI_Comm mesh2;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);

   dims[0]=2; dims[1]=2;
   periods[0]=0; periods[1]=0;
   reorder = 0;
   MPI_Cart_create(MPI_COMM_WORLD,2,dims,periods,reorder,&mesh2);
   MPI_Cart_coords(mesh2,myid,2,coords);
   ... (initialize A)
   MPI_Type_vector(size2,1,size2+1,MPI_FLOAT,&c_column);
   MPI_Type_commit(&c_column);

   MPI_Cart_shift(mesh2,0,1,&up,&down);
   MPI_Cart_shift(mesh2,1,1,&left,&right);

   for (n=0; n<nsteps; n++) {
      MPI_Sendrecv(&a[0][1],1,c_column,left,col_tag,
                   &a[0][size2],1,c_column,right,col_tag,
                   mesh2,&status);
      MPI_Sendrecv(&a[0][size2-1],1,c_column,right,col_tag,
                   &a[0][0],1,c_column,left,col_tag,
                   mesh2,&status);
      MPI_Sendrecv(&a[1][0],size2,MPI_FLOAT,up,row_tag,
                   &a[size2][0],size2,MPI_FLOAT,down,row_tag,
```

```
                mesh2,&status);
    MPI_Sendrecv(&a[size2-1][0],size2,MPI_FLOAT,down,row_tag,
                &a[0][0],size2,MPI_FLOAT,up,row_tag,
                mesh2,&status);
    for (i=1;i<size2;i++)
       for (j=1;j<size2;j++)
          b[i][j] = (a[i+1][j]+a[i-1][j]+a[i][j+1]+a[i][j-1])*0.25;
    for (i=1;i<size2;i++)
       for (j=1;j<size2;j++)
          a[i][j] = b[i][j];
 };
 ... (output)
}
```

The following points are observed from this program.

- `MPI_Cart_create(MPI_COMM_WORLD,2,dims,periods,reorder,&mesh2);`

  create another communicator out of `MPI_COMM_WORLD`. The new communicator has name *mesh2* with a `2 x 2` mesh structure. In general, this could be `m x n`.

  `MPI_Cart_coords(mesh2,myid,2,coords);`

  returns processor's coordinate in the mesh.

- `MPI_Type_vector(size2,1,size2+1,MPI_FLOAT,&c_column);`

  define a new datatype *c_column*, which is essentially a sequence of strided memory locations.

  `MPI_Type_commit(&c_column);`

  is needed after type definition.

- `MPI_Cart_shift(mesh2,0,1,&up,&down);`
  `MPI_Cart_shift(mesh2,1,1,&left,&right);`

  returns ranks of neighbors in a Cartesian structure. These ranks, *up*, *down*, *left*, and *right* will then be used in `MPI_Sendrecv` function. What happens to the boundary processors ? `MPI_PROC_NULL` is returned, if the shift is out of range.

- `MPI_Sendrecv` is called 4 times. It's general, and also safe for boundary processors since `MPI_PROC_NULL` was properly returned by `MPI_Cart_shift` in some of the ranks.

Chapter 4

A first glance of SPMD node programs

This lecture is intended to provide a quick idea on what we mean by an `SPMD` *node program*. Two notions are stressed.

- Semantical correspondence between an HPF program and multiple copies of a FORTRAN 77 program;
- Runtime support, in particular, global/local index mapping.

4.1 Some understanding of parallel languages

"Serial language", such as FORTRAN 77 or C, is not purely "serial" — some parallelism is explicitly expressed by programmer, namely, the evaluation of an expression (either at right hand side of an assignment statement or as some condition in an IF statement, etc.) is semantically "parallel". As seen in Figure 4.1, the parallelism in evaluation of the right hand side expression is only limited by the precedence of arithmetic operators. This kind of *explicit* parallelism is primary target of `VLIW` and `Superscalar` architecture.

The primary target of traditional parallelizing compilers is to exploit parallelism among statements, which may be implicitly expressed by the program. DO loops, not only consume most of program execution time, but also exhibit good structure for such exploitation.

"Serial" here really means *ONE* element at left hand side of an assignment statement.

"Parallel language" like Fortran 95 generalizes the above picture by two steps

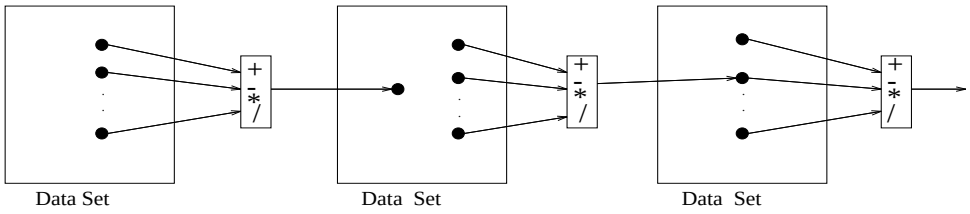
1. Array operations allow assignment to have multi destinations in memory, as depicted in Figure 4.2.

   As a result, the notion of *shape conformance* naturally gets into the scene:

   ```
   A(1:10) = B(5:10)     ! not allowed, while
   A(1:10) = B(5:50:5)   ! is fine
   ```

WHAT'S DESCRIBED BY A FORTRAN 77 PROGRAM ?

--- a data set, and a sequence of operations on elements of the data set

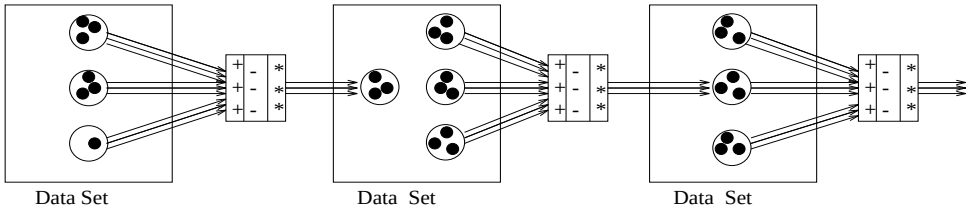



```
PROGRAM F77
REAL a, b, c, d, e
a = b + c - d * e
. . .
END
```

Figure 4.1: Explicit parallelism in FORTRAN 77 !

WHAT'S DESCRIBED BY A FORTRAN 95 PROGRAM ?

--- a data set, and a sequence of operations on subsets of the data set




```
PROGRAM F95
REAL a(100), b(100), c(100), d(100), e(100)
a = b + c - d * e
. . .
END
```

Figure 4.2: Parallel assignment in Fortran 95

and special treatment for scalar is devised:

```
A(1:10) = s  ! is OK, s goes to every element of A(1:10), while
s = A(1:10)  ! does not make sense.
s = A(1:1)   ! also not allowed, while
s = A(1)     ! OK.
```

2. FORALL statements can be used to express *parallel assignment* to "non regular" array sections, (we save the word "irregular" for other purpose) and provide possibility for more versatile right hand side.

   We define *regular array section* as a subset of array elements that can be expressed by *array-section* as specified by Fortran 95 specification minus the *vector-subscript part.* (we covered its syntax in the second lecture.)

   Thus, elements of A referenced in

   ```
   FORALL (i=1:N) A(i,i) = 0
   ```

   is not a regular array section. (we shall use term "array section" for "regular array section" from now on.)

4.2 The impact of parallel language on compilers

Now that the program can express parallel assignment, what are the issues to take some advantage of it ?

- Shared memory, physically global address space.
  - Who is responsible for this part of data ?
  - Data contention, producer/consumer paradigm for data sharing.
- Distributed memory, distributed address space, local data: fine; remote data: problem.
  - where are the data I need ?
  - how to get them ? (how can I store them once they arrive ?)
  - do I have some data needed by others ?

  It's the job of a compiler to figure out all of these and generate code (node program) to perform proper functions at proper time.

In this course, we address issues in distributed memory machines only. If we wanted to write a compiler for Fortran 95 programs to be executed in parallel, the compiler in general would have to perform 4 tasks :

- Data partitioning — distribute elements of data objects for high performance. For instance, decide who gets X(5) for

```
REAL X(100), Y(2:101), Z(0:99)
...
DO i = 1, 100
  X(i) = Y(i+1) + Z(i-1)
END DO
```

- Computation partitioning — assign operations to processors. For instance, decide who performs the second iteration of the above DO loop.
- Communication detection and insertion — get data from other processors when necessary.
- Node program generation — what program will be *executed* by a processor ?

As an example, for the following program,

```
PROGRAM F95
REAL A(100), B(100)
...
A(2:99) = B(1:98) + B(3:100)
...
END
```

a compiler should realize the data A and B is best block-distributed on a one dimensional processor grid. Another example,

```
DO i = 1, N-1
  FORALL (j = i+1:N) s(j) = -A(j,i)/A(i,i)
  FORALL (j = i+1:N) A(j,i+1:N+1) = A(j,i+1:n+1) + A(i,i+1:N+1)*s(j)
END DO
```

A smart compiler may be able to figure out that A should be distributed `(CYCLIC,*)`, (the first `FORALL` is along the row, the second `FORALL` together with the array assignment describes a triangle).

But things are not always so easy. In many cases, compilers just can not figure out an "optimal" way to partition data.

HPF says: let programmer tell you how data should be partitioned. If a compiler follows what the programmer instructs, it's job list becomes

- Computation partitioning
- Communication detection and insertion
- Node program generation

And if we follow *owner computes rule* (or some other simple heuristics), the compiler's job is further reduced to:

- Communication detection and insertion
- Node program generation

Nevertheless, "who is the owner ?" sometimes is not easy to answer by a compiler. See,

```
...
READ *, i
A(i) = B(i) + 5
...
```

The compiler just has no way to know who should do `B(i) + 5`, and runtime checking is required. Any way, "owner computes rule" really simplifies a lot of things for most situations. Thus, we'll follow it.

4.2.1 A glance of communication detection

Among the remaining two items, communication detection and insertion is something easy to get hands on, though it may not be easy to do it well. See the following example,

```
REAL A(100),B(100),C(50)
PROCESSORS P(4)
ALIGN C(i) WITH A(2*i-1)
DISTRIBUTE (BLOCK) ONTO P :: A,B
...
A = B + 1                  ! no communication
A(1:99) = B(2:100) + 1     ! shift communication
A(1:99:2) = B(1:50) + 1    ! some kind of "non regular" communication
...
A(1:50) = C + 1            ! some kind of "non regular" Communication
A(1:99:2) = C + 1          ! no communication
...
END
```

By analyzing data distribution status and array reference patterns, a compiler should be able to determine the assertions in the above comments.

4.2.2 Some high level issues in node program generation

In terms of compiling HPF programs, the following may be elaborated before getting started.

- Target language ? FORTRAN 77, Fortran 90(95), C, assembly language ? This is a strategical choice — a trade-off between technical issues and management (time constraint to get something done, though may not be elegant.)

- Fortran 90(95) is the "easiest" choice, since it's the closest to HPF, but not many Fortran 90(95) compiler available at the moment.
- C would be a good choice if we are aiming at a full featured compiler, and if we have more resources. The problem with C is it's a little far from HPF, many detailed language mapping work has to be done.

  By the way, NAG has implemented their Fortran 90 compiler with C as target.
- We choose FORTRAN 77 based on the following thought:
  * it's a subset of HPF, implying if we have developed an HPF front-end (with parse/unparse capability), compiler work only involves transformations on IR. (if C, we would need to turn the Fortran IR to C IR and do a C unparse) By the way, using FORTRAN 77 as target language is also an established practice by vendors, such as PGI.
  * The disadvantage is: it's hard, if not impossible, to have a full featured HPF (including full Fortran 95) compiler.

  All in all, as a research compiler or course project, this may be the most feasible approach.

- Some heuristic rules
  - Single Program Multiple Data (SPMD). Note, technically, SPMD is not an absolutely necessary choice of model, if we just talk about parallel execution of an HPF program. Other alternatives include host-node, even arbitrary MPMD (MIMD).

    Consequence of SPMD: compiler is responsible to generate proper conditional statements that are used to distinguish different behavior of different processors. These conditions are usually (naturally) based on processors' ranks or coordinates.
  - owner computes rule (as I mentioned, this is not necessary, but a rule to make life easier and deliver good performance in most common cases.)

    Consequence: compiler has (or generates proper calls to runtime) to figure out who is the owner.
  - replicate all variables that are not explicitly distributed. Once again, this is not a must.

    Consequence: compiler has the responsibility to maintain data consistency — once a non distributed datum is updated by one processor, this update must be broadcast to other processors in time (may not be immediately, though.)
  - use collective communication to eliminate explicit barrier synchronizations.
  - static vs dynamic storage management
  - linearization of all arrays in node program ?

4.3 How a node program may look like ?

For the remaining part of this chapter, we present a sequence of possible node programs for corresponding HPF programs. Purpose of this section is to have students obtain an early idea on how the "mysterious" node program may look like. Emphasis is on understanding the *equivalence* between the node program and HPF program, instead of efficiency or optimization issues, etc. We will use Fortran interface for MPI in the examples.

We start from the "empty" HPF program.

```
PROGRAM ONE
END
```

The node program should, in general, not be "empty". We have,

```
PROGRAM NODE_1
include 'mpif.h'
INTEGER ierror
CALL MPI_INIT(ierror)
CALL MPI_FINALIZE(ierror)
END
```

The second program involves simple I/O and a scalar (not distributed).

```
PROGRAM TWO
REAL x
READ *, x
x = x + 1
PRINT *, x
END
```

Then a reasonable node program,

```
PROGRAM NODE_2
include 'mpif.h'
INTEGER ierror,myid
REAL x

CALL MPI_INIT(ierror)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierror)

IF (myid.eq.0) READ *, x
CALL MPI_BCAST(x,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierror)

x = x + 1
if (myid.eq.0) print *,x

CALL MPI_FINALIZE(ierror)
END
```

We should get some feeling, from this example, that I/O introduces some complication.

Although the above is working, one may prefer the following more general treatment. Note the use of `COMMON` block.

```
PROGRAM NODE_2
include 'mpif.h'
COMMON myid
INTEGER ierror,myid
REAL x

CALL MPI_INIT(ierror)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierror)

CALL DATA_IN(x)
x = x + 1
CALL DATA_OUT(x)

CALL MPI_FINALIZE(ierror)
END

SUBROUTINE DATA_IN(x)
include 'mpif.h'
COMMON myid
REAL x
INTEGER ierror
IF (myid.eq.0) READ *, x
CALL MPI_BCAST(x,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierror)
RETURN
END

SUBROUTINE DATA_OUT(x)
include 'mpif.h'
COMMON myid
INTEGER ierror
if (myid.eq.0) print *,x
RETURN
END
```

The above two HPF programs are not really HPF programs — no HPF directives are used; the corresponding node programs are not really message passing node programs — no message passed among processors.

The first 'real' HPF program below (distributed one dimensional arrays) introduces the concept of index mapping.

```
      PROGRAM THREE
      REAL A(100),B(100)
!HPF$ PROCESSORS P(4)
```

```
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: A,B
      A = 1
      B(1:99) = A(2:100) + A(1:99)
      END
```

Two arrays are distributed, and obviously communication is needed for the second assignment. The node program may look something like,

```
      PROGRAM NODE_3
      include 'mpif.h'
      COMMON myid
      INTEGER ierror,myid,l,u,s

      REAL A(25),B(25),T(25)

      CALL MPI_INIT(ierror)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierror)

      CALL loop_bounds(1,100,4,block,...,l,u,s)  ! for this particular
      DO i = l, u, s             ! instance, a smart compiler may figure
        A(i) = 1                 ! out i=1,25. But we need loop_bounds()
      END DO                     ! in general, as we'll see below.

      CALL loop_bounds(1,100,1,99,4,block,...l,u,s)
      CALL some_comm_function(A,T,1,...)  ! move A to some temporary T
      DO i = l,u,s               ! We need to keep A intact. Once again, a
        B(i) = T(i) + A(i)       ! smart compiler may do better than this, such
      END                        ! as introducing a ghost cell in node program.

      CALL MPI_FINALIZE(ierror)
      END
```

A runtime function `loop_bounds()` converts global index range of A and B to local one for each processor. The parameters to this function are not completely specified in the program. Question: what should be included as parameters of `loop_bounds()` for it to be a general runtime function ?

The second 'real' HPF program introduces the issue of local array allocation.

```
      PROGRAM FOUR
      REAL A(100),B(100)
!HPF$ PROCESSORS P(4)
!HPF$ TEMPLATE T(101)
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: T
!HPF$ ALIGN A(i) WITH T(i)
!HPF$ ALIGN B(i) WITH T(i+1)
      A = 1
      B(1:99) = A(2:100) + B(1:99)
      END
```

It is the template T that gets distributed. While the size of a template is not a multiple of number of processors, we have to 'ceil' it to the closet multiple — the spirit of SPMD model — only a single node program is to be generated. Thus,

```
      PROGRAM NODE_4
      include 'mpif.h'
      COMMON myid
      INTEGER ierror,myid,l,u,s

      REAL A(26),B(26)

      CALL MPI_INIT(ierror)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierror)

      CALL loop_bounds(...,l,u,s)
      DO i = l, u, s
        A(i) = 1
      END DO

      CALL loop_bounds(...l,u,s)
      DO i = l,u,s                ! no communication this time
        B(i) = A(i) + B(i)
      END

      CALL MPI_FINALIZE(ierror)
      END
```

We see in general local array allocation is based on template size, instead of global array size. And we observe that unlike program THREE, this HPF program does not require communication once data is distributed as programmer suggested. Question: how can a compiler know if communication is needed ?

Another example on local array allocation. For the following program, elements of array B are aligned with template elements with *stride* 2.

```
      PROGRAM FIVE
      REAL A(20),B(20)
!HPF$ PROCESSORS P(4)
!HPF$ TEMPLATE T(40)
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: T
!HPF$ ALIGN A(i) WITH T(i)
!HPF$ ALIGN B(i) WITH T(2*i-1)
      ...
```

This situation may be depicted as

```
        p1          p2          p3          p4
T:  aaaaaaaaaa|aaaaaaaaaa|----------|----------
T:  b-b-b-b-b-|b-b-b-b-b-|b-b-b-b-b-|b-b-b-b-b-
```

We need to allocate 10 locations for both A and B locally, though p3 and p4 does not contain any A, and a half of local B has no corresponding global B elements.

```
      PROGRAM NODE_5
      include 'mpif.h'
      ...
      REAL A(10),B(10)
      ...
      END
```

Thus, we say some of elements of A, or B are "effective elements". Different processors may have different number of effective elements, though they have the same local array declaration. For the above example, processor 2 and 3 will have no effective elements in A; every other element of B is effective element in every processor.

Question: why not saving some storage space by defining B as B(25) ?

The next example shows what happens for cyclic distribution.

```
      PROGRAM SIX
      REAL A(16)
!HPF$ PROCESSORS P(4)
!HPF$ TEMPLATE T(16)
!HPF$ DISTRIBUTE (CYCLIC) ONTO P :: T
!HPF$ ALIGN A(i) WITH T(i)
      FORALL (i=1:16) A(i) = i
      ...
      END
```

```
     p1   p2   p3   p4
T:  aaaaaaaaaaaaaaaa
    1234123412341234
```

Each processor has 4 elements, and would have node program like,

```
      PROGRAM NODE_6
      include 'mpif.h'
      ...
      REAL A(4)
      ...
      END
```

More specifically, table 4.3 shows a reasonable assignment of global elements to processor and local elements.

The next example introduces the notion of local to global index conversion.

```
      PROGRAM SEVEN
      REAL A(200)
!HPF$ PROCESSORS P(4)
```

Table 4.1: Assignment of global elements for CYCLIC distribution

	P1	P2	P3	P4
a(1)	A(1)	A(2)	A(3)	A(4)
a(2)	A(5)	A(6)	A(7)	A(8)
a(3)	A(9)	A(10)	A(11)	A(12)
a(4)	A(13)	A(14)	A(15)	A(16)

```
!HPF$ TEMPLATE T(200)
!HPF$ DISTRIBUTE (CYCLIC) ONTO P :: T
!HPF$ ALIGN A(i) WITH T(i)
      FORALL (i=1:200) A(i) = i
      ...

      PROGRAM NODE_7
      include 'mpif.h'
      ...
      REAL A(50)
      CALL loop_bounds(...,l,u,s)
      DO i = l,u,s
        A(i) = local2global(i,1,200,cyclic,4,...)
      END
      ...
```

We see loop_bounds() is used again, which can be considered as a form of global to local index conversion function. The necessity of `local2global()` is clear in order to preserve the semantics of the HPF program.

Once again, what should be complete parameter structure of `local2global()` ?

We see some "well behaved" two dimensional arrays in next example.

```
      PROGRAM EIGHT
      REAL A(80,100)
!HPF$ PROCESSORS P(2,2)
!HPF$ DISTRIBUTE (BLOCK,CYCLIC) ONTO P :: A
      FORALL (i=1:80,j=1:100) A(i,j) = i+80*j
      ...
```

Two `loop_bounds()` should be called naturally, as in

```
      PROGRAM NODE_8
      include 'mpif.h'
      ...
      REAL A(20,25)
      CALL loop_bounds(...1,80,block,li,ui,si)
      CALL loop_bounds(...1,100,cyclic,lj,uj,sj)
```

```
      DO i = li,ui,si
        DO j = lj,uj,sj
          A(i,j) = local2global(i,...) + 80*local2global(j,...)
        END DO
      END DO
```

However, an optimizing compiler may generate,

```
      ...
      DO i = li,ui,si
        n = local2global(i,...)
        DO j = lj,uj,sj
          A(i,j) = n + 80*local2global(j,...)
        END DO
      END DO
```

which moves a call to runtime out of innner loop. Another "well behaved" array example,

```
      PROGRAM NINE
      REAL A(80,100)
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE (BLOCK,*) ONTO P :: A
      FORALL (i=1:80,j=1:100) A(i,j) = i+80*j
      ...
```

An *ad hoc* node program might be,

```
      PROGRAM NODE_9
      include 'mpif.h'
      ...
      REAL A(20,100)
      CALL loop_bounds(...1,80,block,li,ui,si)
      CALL loop_bounds(...1,100,*,lj,uj,sj)
      DO i = li,ui,si
        DO j = lj,uj,sj
          A(i,j) = local2global(i,...) + 80*local2global(j,...)
        END DO
      END DO
```

Once again, an optimizing compiler should generate,

```
      ...
      DO i = li,ui,si
        n = local2global(i,...)      !!!
        DO j = 1,100                 !!!
          A(i,j) = n + 80*j
        END DO
      END DO
```

Array "slicing", or rank-reduced sectioning, introduces some complication.

```
      PROGRAM TEN
      REAL A(100,100),B(100)
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE (BLOCK,*) ONTO P :: A
!HPF$ DISTRIBUTE (CYCLIC) ONTO P :: B
      B = 1
      A(30,:) = B
      ...
```

The 30th row of A is distributed on processor 2, thus other processors should not perform the array assignment. To cope with this situation, a logical runtime function `onMe()` may be designed.

```
      PROGRAM NODE_10
      include 'mpif.h'
      ...
      REAL A(25,100),B(25),T(25)
      CALL loop_bounds(...1,100,cyclic,l,u,s)
      DO i = l,u,s
        B(i) = 1
      END DO
      CALL loop_bounds(...1,100,2,*,l,u,s)
      CALL some_communication_function(T,B,...)     ! T <-- B
      IF (onMe(A,30,...)) THEN
        k = global2local(30,block,...)
        DO i=l,u,s
          A(k,i) = T(i)
        END DO
      END IF
      ...
```

Question: parameters of onMe() ? We also see another runtime function `global2local()` used in the node program.

Note: it is possible to design `loop_bounds()` in such a way that it returns proper (l,u,s) so that the DO loop will not be executed on processors with `onMe()` false. In that case, node program does not have to call `onMe()`.

Our last example involves subroutine calls.

```
      PROGRAM ELEVEN
      REAL A(100)
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: A
      ...
      CALL FOO(A(1:99:2))
      ...
      SUBROUTINE FOO(X)
```

```
      REAL X(:)
      X = X + 1
      RETURN
      END
```

A general array section as actual argument passed to subroutine is something difficult to deal with in compilation system. Nevertheless, the following should be a reasonable skeleton for this example.

```
      PROGRAM NODE_11
      include 'mpif.h'
      REAL A(25)
      ...
      CALL loop_bounds(...,l,u,s)
      CALL FOO(A(l:u:s)...)
      ...

      SUBROUTINE FOO(X...)
      REAL X(*)
      ...
      CALL loop_bounds(...l,u,s) ! we can not simply copy the l,u,s from
      DO i = l,u,s               ! main program, since a subroutine may
        X(...) = X(...) + 1      ! be separately compiled.
      END DO
      ...
      RETURN
      END
```

Question: how we are going to figure out l,u,s in a subroutine without knowing dimension information of dummy X ?

Homework: Given

```
      DIMENSION X(lx:ux)
!HPF$ PROCESSOR P(p)
!HPF$ TEMPLATE T(lt:ut)
!HPF$ ALIGN X(i) WITH T(ax*i+bx)
!HPF$ DISTRIBUTE T(d) ONTO P
```

where d takes one of `BLOCK` and `CYCLIC`. Derive a formula (or procedure) that determines l,u,s of local X(l:u:s) corresponding to global X(gl:gu:gs) for each processor, assuming local array index starts from 0. (As you may realize, this procedure is the core of loop_bounds().) For instance, given `lx=2, ux=40, p=2, lt=-5, ut=80, ax=2, bx=-1, d=BLOCK` and `gl:gu:gs = 6:30:3`, your procedure (or formula) should figure out: `l:u:s = 16:40:6` for processor 1, and `l:u:s = 3:21:6` for processor 2.

Chapter 5

Data distribution model

Explicit array data distribution is the core concept of HPF, which potentially frees compiler from the task of *data partitioning*. We discuss HPF data distribution model in detail in this lecture. While HPF programmer and implementor may have different views of this model, we approach it from the latter. In particular, a reasonable local memory allocation scheme corresponding to this model is elaborated.

5.1 The impact of data distribution directives on compilers

Without HPF data distribution directives, the compiler has to perform data partitioning, either according to some rule-of-thumb, or based on an analysis of data access pattern presented in the program. Now with HPF directives, what compilers have to do with respect to data partitioning then becomes

- Analyze the data distribution directives given by programmer;
- Allocate *proper* local storage in node program, corresponding to the global arrays specified in the HPF program;

There are many issues connected with the word *proper*. One essential issue is that the storage allocation scheme should make global to local (and vice versa) index conversion (an unavoidable common operation in both node program and runtime) as efficient as possible. The following example should give you some idea on what we mean by *index conversion* or *index mapping*. For the HPF program below,

```
      INTEGER X(100)
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE X(BLOCK)
      FORALL (i=1:100) X(i) = i
      ...
```

A reasonable node program may look like,

```
      INTEGER X(25)
      DO i=1,25
         X(i) = local2global(...,i)
      END
      ...
```

That is, the 100 elements are evenly distributed on 4 processors (each gets 25), and the runtime function local2global() must return different values for the same value i on different processors. The way how local storage is allocated determines the algorithm used in this function.

In general, given a description of data distribution (say array X), there is a list of related questions should be answered efficiently (from algorithmic point of view).

- Shape of corresponding local array ?
- Layout of *effective elements* in the local array ?
- Which processor(s) hold a particular element of X ?
- What's the corresponding local index for this element ?
- What portion of X does a processor have ?
- What's the corresponding global index of a local element ?

  ...

The following program illustrates the concept behind the second item in the above list.

```
      INTEGER X(50)
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE X(BLOCK)
      ...
```

Since SPMD node program is to be produced by the compiler, we need to allocate at least 13 elements for local array on each of the 4 processors. As a result, some of the 52 local memory locations do not hold any of the 50 array elements. For those do, we call them *effective local array elements.*

5.2 An alignment-distribution arithmetic

This section describes a local memory allocation scheme and algorithms, with respect to the scheme, to answer the questions posted in the previous section.

For convenience, we define an *alignment-distribution scheme* (or AD scheme in short.) is composed of two maps

$$X(i_1, i_2, ..., i_m) \longmapsto T(e_1, e_2, ..., e_n)$$
$$T(d_1, d_2, ..., d_n) \longmapsto P(p_1, p_2, ..., p_k)$$

the first map is called *alignment*, the second *distribution*, where

- i_j, j=1,...,m, is *align-source-list*, each align-source is one of
  - *align-dummy*,
  - `*`,

  where *align-dummy* is a variable name.
- e_j, j=1,...,n, is *align-subscript-list*, each align-subscript is one of
  - a linear expression of some align-dummy, such as `a*i+b`, where i is an align-dummy, `a` and `b` are constants.
  - `*`
- d_j, j=1,...,n, is *dist-format-list*, each dist-format is one of
  - `BLOCK`
  - `CYCLIC`
  - `*`
- p_j, j=1,...,k, is a list of positive integers.

Constraint:

- each align-dummy must appear in align-subscript-list exactly once;
- an align-subscript may contain at most one align-dummy;
- the number of non `*` in dist-format-list must be equal to k.

Semantics of AD-scheme

- dimension matching
  - align-dummies in align-source-list match linear expressions in align-subscript-list by dummy name. Thus, permutation on dimensions is possible;
  - non `*` items in dist-format-list match dimensions of processor grid by position, from left to right.
- position matching
  - the expression `a*i+b` specifies a position in template for an array element indexed by `i`;

Table 5.1: Illustration of alignment

`X(i,j) --> T1(i,2*j)`	stridden in the second dimension
`X(i,*) --> T2(i+5)`	the second dim of X is collapsed, each element of T2(6:15) holds a row of X.
`Y(i) --> T1(i,*)`	each element of Y is replicated along the second dimension of T1 for the first 10 rows of T1, i.e., T1(i,1)=T1(i,2)=...=T1(i,30) := Y(i)
`X(*,i) --> T1(i,*)`	each column of X is replicated along the second dimension of T1 for the first 10 rows of T1, i.e., T1(i,1)=T1(i,2)=...=T1(i,30) := X(1:10,i)

- collapsing and replication
  - a `*` in align-source-list means the corresponding array dimension will be 'collapsed' onto the *template element* specified by other non `*` items in align-source-list.
  - a `*` in dist-format-list means the template dimension is 'collapsed' on processor grid;
  - a `*` in align-subscript-list means the array element(s) mapped to the template element specified by other non `*` items of align-subscript-list is(are) replicated along this template dimension.

    Note: `*` in T(...) in alignment has nothing to do with `*` in T(...) in distribution. They have different meanings and may appear in the same or different positions.

Table 5.1 illustrates some examples of alignment for specification `X(10,10), Y(10), T1(20,30), T2(20)`,

Table 5.2 illustrates some examples of distribution for specification `T1(20), T2(20,30), T3(20,30,40)`.

Template elements are distributed/partitioned among processors; the ultimate effect is the array elements aligned to template elements are distributed.

5.3 How do we determine local rank and shape of an array ?

Local rank. From the distribute directive, we see local rank should be in general at least

Table 5.2: Illustration of distribution

`T1(CYCLIC) --> P(4)`	template is cyclically distributed
`T2(BLOCK,BLOCK) --> P(2,2)`	two dimension block distribution
`T2(*,CYCLIC) --> P(4)`	columns of T2 are collapsed, i.e. each processor owns several columns of T1, i.e. T2(1,i),T2(2,i),...,T2(20,i) are all on the same processor for fixed i.
`T3(*,*,BLOCK) --> P(4)`	each processor owns several two dimensional slices of T3, i.e., T3(1:20,1:30,i) is on a processor for any fixed i.

as template rank. Notice each template element may correspond to some 'slice' of original array, due to collapsed alignment, we have

$$rank(local) = rank(template) + number\ of\ collapsed\ dims\ of\ global\ X$$

on the other hand, we see the non replicated template dims capture the whole global array, and the array is replicated as many dimensions as number of replicated template dims. Thus, we also have

$$rank(local) = rank(global) + number\ of\ replicated\ dims\ of\ template$$

(we can also derive the second formula from the first one by noting the constraint: number of non collapsed X dims = number of non replicated T dims. Note the rank(local) is independent of rank(processor).)

Local shape. We need to decide extent of each local dimension as well as order of the dimensions.

- **Extents**
  - if a template dimension of extent e is distributed (either block or cyclic) over p processors, we have a local extent = $\lceil \frac{e}{p} \rceil$;
  - if a *template* dimension of extent e is collapsed, we have a local extent = e;
  - if an *array* dimension of extent e is collapsed, we have a local extent = e;
- **Dimension orders.** We let a local array preserve the same dimension order as its global array.
  - if array dim i is collapsed, local dim i will have the same extent as global one;
  - if array dim i is aligned to template dim j, local dim i will have extent resulting from template dim j;

Table 5.3: Determine local shape from AD scheme

`X(i,*) --> T1(i+5)` `T1(CYCLIC) --> P(4)`	X has local shape (5,10)
`X(i,j) --> T2(i,2*j)` `T2(BLOCK,BLOCK) --> P(2,2)`	X has local shape (10,15)
`Y(i) --> T2(i,*)` `T2(*,CYCLIC) --> P(4)`	X has local shape (20,8)
`X(*,i) --> T2(i,*)` `T2(CYCLIC,BLOCK) --> P(2,2)`	X has local shape (10,10,5)
`X(i,j) --> T3(2*j-1,2*i+1,*)` `T3(*,*,BLOCK) --> P(4)`	X has local shape (30,20,10) Figure 5.1 depicts this situation

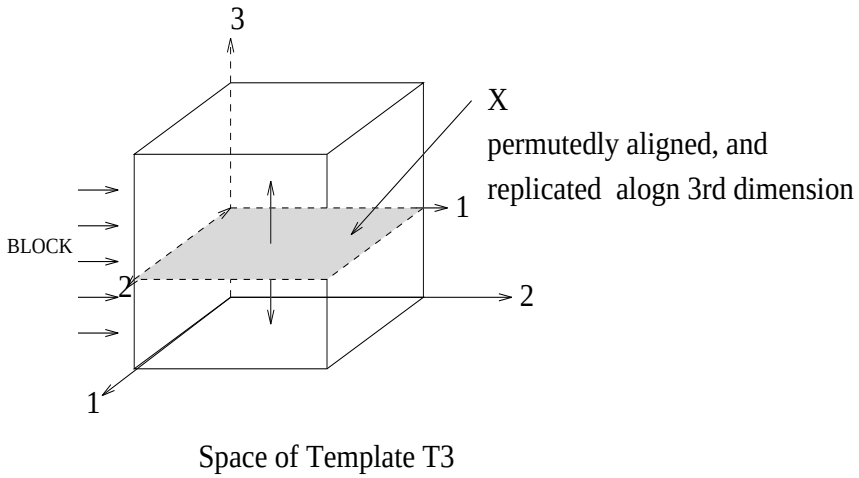


Figure 5.1: A view of an AD scheme

– the extents resulting from replicated template dimensions will be placed after the first rank(global) positions in order they appear in template.

Table 5.3 shows the local array shapes corresponding to various AD schemes for specification,

```
X(10,10), Y(10), T1(20), T2(20,30), T3(20,30,40)
```

When given an AD scheme and declaration of array X and template T in the form $X(xrng_1, xrng_2, ..., xrng_m)$ and $T(trng_1, trng_2, ..., trng_n)$ where $xrng_i$, i=1,...,m, and $trng_j$, j=1,...,n are in the form $l_x : u_x$ and $l_t : u_t$ (with $l < u$ being integers). For convenience, we use $xe_1, ..., xe_m$ to denote the extents of $xrng_1, ..., xrng_m$, namely u-l+1, respectively; and use $te_1, ..., te_n$ for extents of $trng_1, ..., trng_n$.

We want to be able to answer:

- Which processor(s) hold a particular element of X, say $X(a_1, a_2, ..., a_m)$? What's the corresponding local indices for this element ?
- What portion of X does a processor (given its coordinates in its grid) have ? Note, we refer to HPF processor grid here, not to confused with processor mesh in MPI. Here, coordinates are 1-based, while MPI mesh uses 0-based coordinate system.
- Local shape (layout) of X on a particular processor ?
- What's the global indices of a local X ?

We distinguish six types of alignment:

1. offset alignment: $i \longrightarrow i + b$
2. stridden alignment: $i \longrightarrow a * i$
3. permuted alignment: $(i, j) \longrightarrow (j, i)$
4. collapsed alignment: $(*, i) \longrightarrow (i)$
5. replicated alignment: $(i) \longrightarrow (i, *)$

   of course, a particular alignment may demonstrate more or all of the above.
6. perfect alignment: permutation function is identity function; shape(X) = shape(T), (the above two imply a=1, b=0); no collapsing (the above three imply no replication.)

We distinguish three types of distribution:

1. `BLOCK` distribution
2. `CYCLIC` distribution
3. collapsed distribution

Assumptions

- Memory allocation of a local array is based on the shape(local) description specified early.
- The index of local array is 0-based.

We note, an AD scheme induces a function from {1,2,...,m} to {1,2,...,n,*}. Let's call this function *perm()*, and we may think of a vector of length m to represent it. For instance,

```
X(i,*,j,k,*,*) --> T(*,j-1,k+1,*,2*i+1)
```

the *perm()* vector would look like: (5,0,2,3,0,0), we use 0 for '*'.

An AD scheme also induces an order preserving onto function from {1,2,...,n} to {1,2,...,k}. Let's call it *on()*, and we may also think of a vector of length n to represent it. For instance,

```
T(*,block,block,*,cyclic) --> P(2,3,4)
```

the *on()* vector would look like: (0,1,2,0,3)

Now, where is $X(a_1, a_2, ..., a_m)$? ($a_1, ..., a_m$ are given integer constants.)

- for each i=1,...,m, such that $perm(i) \neq 0$,

  compute $e_{perm(i)}$, based on the AD scheme and a_i;

  We end up with an 'instantiated' $T(e_1, e_2, ..., e_n)$, where e_i is either some integer constant or *.

- for each i=1,...,n, such that $on(i) \neq 0$, {
  if ($e_i \neq' *'$) then
  if d_i = 'block' {
  $w_i = \lceil te_i / p_{on(i)} \rceil$;
  $c_{on(i)} = \lceil (e_i - lt_i + 1)/w_i \rceil$;
  }
  if d_i = 'cyclic' {
  $c_{on(i)} = ((e_i - lt_i)\ mod\ p_{on(i)}) + 1$;
  }
  else
  $c_{on(i)} = 1, 2, ..., p_{on(i)}$
  }

Then $(c_1, c_2, ..., c_k)$ obtained from the procedure represent the coordinate(s) of the processor(s) that hold the $X(a_1, a_2, ..., a_m)$.

Note, it's possible for more than one processor to have the same element.

Examples. For the declaration X(10,10), Y(10), T1(20), T2(20,30), T3(20,30,40), we locate the questioned global array elements for each of the cases below.

```
X(1,1) ?
X(i,*) --> T1(i+5)
T1(CYCLIC) --> P(4)
```

P(2) has it as X(1,0).

```
X(5,8) ?
X(i,j) --> T2(i,2*j);
T2(BLOCK,BLOCK) --> P(2,2);
```

P(1,2) has it as X(4,0).

```
Y(6) ?
Y(i) --> T2(i,*);
T2(*,CYCLIC) --> P(4);
```

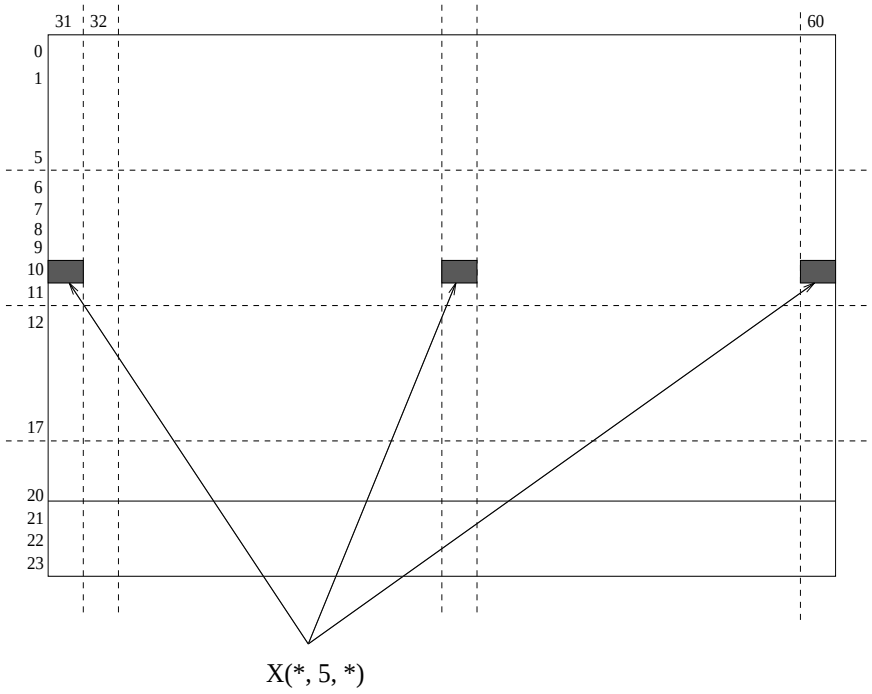


Figure 5.2: Locate global element in local array

P(1:4), i.e., every one, have it as Y(5,0:7), i.e., even multi copies in one processor (we'll place some restriction on replicate alignment to avoid this waste.)

```
X(5,8) ?
X(*,i) --> T2(i,*);
T2(CYCLIC,BLOCK) --> P(2,2);
```

P(2,1:2) have is as X(4,3,0:14).

```
X(1,1) ?
X(i,j) --> T3(2*j-1,2*i+1,*)
T3(*,*,BLOCK) --> P(4);
```

P(1:4) have it as X(2,0,0:9).

In the above examples, we have not only answered which processor owns a particular global array element, but also given its local index. In general, local index of the element $x(h_1, ..., h_m, ..., h_q)$, (note q-m+1 = number of replicated template dimensions), may be determined by the following procedure.

```
for i=1,...,m
   if perm(i)=0 then h(i) = a(i)-lx(i);    /* local index 0-based */

for i=1,...,n, such that e(i) <> "*"
   if (on(i) <> 0) then
      if d(i) = 'block'
         w(i) = ceiling(te(i)/p_on(i));
         h_perm'(i) = e(i)-lt(i) mod w(i);
      if d(i) = 'cyclic'
         h_perm'(i) = floor((e(i)-lt(i))/p_on(i));
   else
      h_perm'(i) = e(i)-lt(i);
/* the above fill up the first m dimensions. For the rest
   dimensions, let r(i) be the i-th replicated template dimension */
for i=1,...,q-m+1
   if d(r(i)) <> '*' then
      h(m+i) = {0,1,...,ceiling((ut(r(i))-lt(r(i)))/p(r(i))};
   else
      h(m+i) = {0,1,...,ut(r(i))-lt(r(i))};
```

As mentioned above, arbitrary replication may result in multi copies of the same element in a processor, which is a waste. For any practical use, we might want to impose the following constraint on AD scheme:

> in ALIGN directive, if an array is replicated along certain dimensions of a template, the corresponding dimensions of the template must be distributed, and extents of the dimensions must be equal to the extents of corresponding dimensions of the processor grid.

under this constraint, we shall always have $rank(local) = rank(global)$.

We see one more example. For the following,

```
X(1:10,-3:10,-10:10), T(0:20,31:60)
X(2,5,-1) ?
X(*,i,*) --> T(i+5,*)
T(b,b) --> P(4,30)
```

Processors P(2,1:30) have the questioned element as x(1,4,2). Figure 5.2 depicts the situation.

Homework:

1. For the following setting, which processor(s) have the questioned element ? And what's its local index ?

```
X(10,100,200,300), T(100,200,300,400,500)
X(1,50,100,150) ?
X(i,*,j,*) --> T(*,2*i,*,j+10,*)
T(b,*,c,b,*) --> P(4,4,4)
```

(Do you get: P(1:4,1:4,2) have the element as local x(1,49,9,149,1:25,1:75,1:500) !!)

2. Design an algorithm that for given processor $P(c_1, c_2, ..., c_k)$ and an AD scheme, determines what portion of global X this processor has.

Chapter 6

Distributed array descriptor (DAD)

The third interface of a data parallel compilation system can be conveniently (and conceptually) viewed as composed of two components: *data interface* and *functional interface*. This chapter presents a detailed design of the data interface, namely a data object that describes a distributed (global) array to runtime functions (node).

6.1 How do we pass an array to a subroutine in FORTRAN 77 ?

Put it in another way, what caller has to prepare to call a Fortran subroutine, or how should a subroutine writer design its parameter structure ?

To its completeness, we see the following information is necessary for each array,

- base address, usually represented by the name of an array.
- element type (often implied by nature of the subroutine), used to determine how many bytes of memory an array element takes.
- dimension information, i.e., shape of the array.
- majority, if the call is within the same language, it's often implied; but needed for mixed language programming, unless some other equivalent measure is taken.

in addition to the basic assumption that an array is stored in a contiguous address space in the memory.

Some common mistake is often observed, due to not understanding the above. For instance, the following program won't output 3 as one might think it would. What's wrong with it ?

```
SUBROUTINE SUB(X,n,m)
REAL X(n,m)
DO i=1,n
  DO j=1,m
    X(i,j) = i+j
  END DO
END DO
END

PROGRAM MAIN
REAL X(10,10)
DATA X/100*0/      ! initialize to all 0
READ *,n,m         ! say n=2, m=3
CALL SUB(X,n,m)
PRINT *, X(1,3)     ! do you get 4 ??
END
```

You'll actually get 0 ! This is because the `MAIN` and `SUB` see two arrays of the *same* base address, the *same* type, the *same* majority, but *different* shapes. One way to do it right is the following,

```
SUBROUTINE SUB(X,d1,d2,n,m)
INTEGER d1,d2,n,m
REAL X(d1,d2)
DO i=1,n
  DO j=1,m
    X(i,j) = i+j
  END DO
END DO
END

PROGRAM MAIN
REAL X(10,10)
DATA X/100*0/      ! initialize to all 0
READ *,n,m
CALL SUB(X,10,10,n,m)
PRINT *, X(1,3)   ! I get 4
END
```

This time, two program units see exactly the same array in memory.

What do you expect to get if the `SUB` is called by the following C `main` ?

```
main() {
   float x[10][10] = {0,0,...,0};  /* there are 100 zeros */
   int n,m,d1=10,d2=10;
   scanf("%d%d",&n,&m);

   sub_(x,&d1,&d2,&n,&m);    /* FORTRAN routine needs address */
```

```
   printf("%f\n",x[0][2]);
}
```

Do you get 4 ? No, you'll get 0, instead ! This is because x[0][2] is the 3rd element in the memory from C point of view, but FORTRAN's 3rd element is x(3,1), which is not touched by the `SUB`. One way to get expected 4 is by switching dimension information when calling FORTRAN routine as,

```
main() {
   float x[10][10] = {0,0,...,0};
   int n,m,d1=10,d2=10;
   scanf("%d%d",&n,&m);

   sub_(x,&d2,&d1,&m,&n);    /* switch the dimensions */

   printf("%f\n",x[0][2]);
}
```

Thus, if we let data type be understood by the design of subprograms, we may see **(base address, dim info)** as array descriptor in sequential program. Here, we assume caller's responsibility to take care of majority problem, though it's possible for callee to handle it (then majority info must be included in the parameters, callee uses it to do IF statement). Since the information in such a descriptor is not much, we normally do not use a distinct data object to represent it.

To describe a distributed (global) array to a node program (such as a runtime function), much more information is needed. It's convenient to use a distinct data object, called DAD, to encompass the information. Incidentally, to my knowledge (after we pick it for *distributed data descriptor*), DAD was first used as acronym in related context for *data access descriptor* in [16].

6.2 Design of a DAD

In design of a DAD, two basic aspects should be considered.

- what information should be in the DAD ?
- how do you organize those pieces of information ?

if redundant information is introduced for efficiency reason, a third issue must be considered

- the relation among different pieces of information.

if we would want the DAD to support HPF subprograms (not node program), a fourth issue to be considered is

- a mechanism to represent dummy array to runtime function. For example,

```
      SUBROUTINE SUB(X,Y)
!HPF$ INHERIT X,Y
      REAL X(0:),Y(1:)
      X = Y + 1
      END
```

  there is no AD information for X and Y available, how does a compiler generate node program to call runtime functions, in case the assignment needs communication, etc. ?

  Naturally, DAD of actual argument should be used, but there are some complications.

  - array section can be used as actual argument from caller;
  - dummy may be specified as assumed-shape array.

6.2.1 What should be in DAD ?

We'll develop the contents of a DAD in this section. We first observe some preliminary need, and then refine it as we understand the requirement.

- *local* base address. Yes, it's local. HPF arrays are only *logical* entities used in program by programmers. Physical address must be local.
- element type
- global shape. We need it for index mapping. Although a programmer may specify lower bound and upper bound for an array dimension, we only need extent in DAD, assuming compiler takes care of normalization work when providing parameters to runtime.
- rank, to know how many elements in the shape.
- template extent that an array dimension is align with. We also need it for index mapping. Note, we do not need template shape, since what's really important is the extent of a template dimension, which potentially determines the local array's extent, as we discussed in last chapter.
- alignment information, including things in "align-source-list" and "align-subscript-list"
- distribution information, including things in "dist-format-list"
- processor dimension this array dimension is *eventually* distributed on.
- processor shape (or extents).

The above information is sort of orthogonal and complete. Let's see how they may be represented.

```
struct Dad {
   void base_address;
   int type_code;
   int rank;
   int p_rank;     /* to tell # elements in p_shape

   int *g_extent;  /* "rank" integers */
   int *t_extent;
   int *t_stride;
   int *t_offset;
   int *dist_code;
   int *on_pdim;

   int *p_shape;  /* "p_rank" integers */
}
```

We define:

- `*(t_extent+i) = 0` iff the $i+1^{th}$ dimension of the array is 'collapsed', otherwise `*(t_extent+i)` is the extent of the template that array dimension i+1 is aligned with.

  (note the relation between `*t_extent` and perm() function introduced in last chapter.)

- when `*(t_extent+i)==0`, `*(t_stride+i)` and `*(t_offset+i)` are insignificant, otherwise, they represent the stride of array elements and the offset of the first array element on the template dimension, respectively.

- `*(on_pdim+i)` indicates the processor grid dimension that this array dimension is distributed on.

- `*(p_shape+i)` is the number of processors in (i+1)th processor grid dimension.

Is the above sufficient ? Consider

```
      PROGRAM HPF_FOO
      REAL X(16,16),Y(17,17)
!HPF$ PROCESSORS P1(4)
!HPF$ PROCESSORS P2(2,2)
!HPF$ TEMPLATE T(100,100)
!HPF$ ALIGN X(i,j) WITH T(2*i+5,j)
!HPF$ ALIGN Y(i,j) WITH T(3*i+7,j)
!HPF$ DISTRIBUTE T(BLOCK,BLOCK) ONTO P2
      ...
      X(1:16,1:16) = Y(2:17,2:17)

      END
```

with a reasonable node program as,

```
PROGRAM NODE_FOO
REAL X(25,25),Y(25,25)
...
(call DAD constructor for array X)
(call DAD constructor for array Y)
CALL data_move(DAD_X,1,16,1,16,DAD_Y,2,17,2,17)

END
```

How does a process (executing one copy of the node program) know what data it has, and where to get the data it needs ?

- it must know its position in the processor grid
- then it must first know which processor grid the array is distributed on, noting HPF allows more then one `PROCESSORS` directive.

Thus, processor grid the array is ultimately distributed onto should have an identity presented in DAD, and some *way* for a processor to get its position in the processor group.

The representation of a processor group identity is implementation dependent. If `MPI` is used as an underlying communication system, an `MPI` communicator is a natual choice, and the way for a processor to get its position in the processor group could be by calling

```
MPI_Comm_rank(MPI_Comm comm, int *myid);
MPI_Cart_coords(MPI_Comm comm, int myid, int maxdims, int *coords);
```

As a reasonable alternative, we may let the DAD contain the coords[] information during its creation (at runtime), so that no need to call the above two `MPI` functions each time when some data movement is needed. Then we may add the following entries to struct Dad

```
struct Dad {
   ...
   MPI_comm comm;
   int *my_coord;  /* "p_rank" integers */
}
```

Note, this is an example of trading space for time. Also, importantly, the `*my_coord` is the first piece of information in DAD (so far) that is *processor dependent.*

Now, we should say our DAD is quite complete, if we do not want to support HPF subprograms. From information contained in it, runtime function is able to figure out the basic question we posted last lecture, namely *what data do I have ?*

Nevertheless, to answer such a question, based on the above DAD info, is non trivial. And since this question (or some variant of it) needs to be answered very often, it should be a good idea to store the answer in DAD during its creation, to avoid repeated non trivial computation — another example of trading space for time. In particular, we would like to have

```
struct Dad {
   ...
   int *l_extent;  /* "rank" integers for local shape of the array */
   int *l_lb;      /* "rank" integers for indice of 1st effective element */
   int *l_ub;      /* "rank" integers for indice of last effective element */
   int *l_stride;  /* "rank" integers for local stride on each dimension */
}
```

One of uses of these quantities is they are simply handy for traversing local effective elements, such as

```
   for (i=0;i<rank;i++)
      index[i] = *(l_lb+i);

   do {
     for (i=0;i<rank;i++)
        for (j=i+1;j<rank;j++)
           coef[i] = coef[i]*(*(l_extent+j));

     address = base_address;
     for (i=0;i<rank;i++)
        address = address + index[i]*coef[i];

   ... *address ...    /* is the element, we assume row major here */

     index[rank-1] = index[rank-1] + *(l_stride+rank-1);
     for (i=rank-1;i>0;i--)       /* it's like carry propagation */
        if (index[i] > *(l_ub+i)) {
           index[i] = *(l_lb+i);
           index[i-1] = index[i-1]+*(l_stride+i-1);
        }
   }
   while(not all index[i] == *(l_lb[i]));  /* finish when return to init */
```

We note, *l_extent* and *l_stride* are processor independent. (the first one is easy to observe, the second, may need some thought.), while *l_lb* and *l_ub* are clearly processor dependent.

To support subprogram, another entry `slice_coord` is needed in DAD, to represent rank reduced array section passed as actual argument (more on this later.). Another often needed support is ghost area. We include them in DAD.

Before we go on to algorithms for the last 4 quantities, we elaborate a little bit on the DAD organization. It could be just like the struct, if C is used as language for runtime.

```
struct Dad {
   void base_address;
   int type_code;
   int rank;
   int p_rank;
```

```
   MPI_comm comm;
   int majority;

   int *g_extent;  /* "rank" integers */
   int *t_extent;  /* "t_rank" integers */
   int *t_stride;  /* align stride, "rank" integers. */
   int *t_offset;  /* align offset, "rank" integers */
   int *dist_code; /* code for BLOCK, CYCLIC, and * */
   int *on_pdim;
   int *l_extent;  /* "rank" integers for local shape of the array */
   int *l_lb;      /* "rank" integers for indice of 1st effective element */
   int *l_ub;      /* "rank" integers for indice of last effective element */
   int *l_stride;  /* "rank" integers for local stride on each dimension */
   int *ghost_size;

   int *p_shape;  /* "p_rank" integers */
   int *my_coord;  /* "p_rank" integers */
   int *slice_coord;
}
```

However, if we are designing a DAD for potentially mixed language runtime functions, say intrinsic functions are coded in Fortran, other data movement functions in C, the above struct would be inconvenient. Thus, we might want to employ an integer array, called `DAD_dim`, of 15 x 7 that is used to hold

```
struct Dad {
   int *g_extent;  /* "rank" integers */
   int *t_extent;  /* "t_rank" integers */
   int *t_stride;  /* align stride, "rank" integers. */
   int *t_offset;  /* align offset, "rank" integers */
   int *dist_code; /* code for BLOCK, CYCLIC, and * */
   int *on_pdim;
   int *l_extent;  /* "rank" integers for local shape of the array */
   int *l_lb;      /* "rank" integers for indice of 1st effective element */
   int *l_ub;      /* "rank" integers for indice of last effective element */
   int *l_stride;  /* "rank" integers for local stride on each dimension */
   int *ghost_size;
   int *p_shape;  /* "p_rank" integers */
   int *my_coord;  /* "p_rank" integers */
   int *slice_coord;
}
```

in the first 14 rows (we know ranks are no more than 7 in Fortran.), and use the last row to hold

```
   int typeCode;
   int rank;
   int p_rank;
```

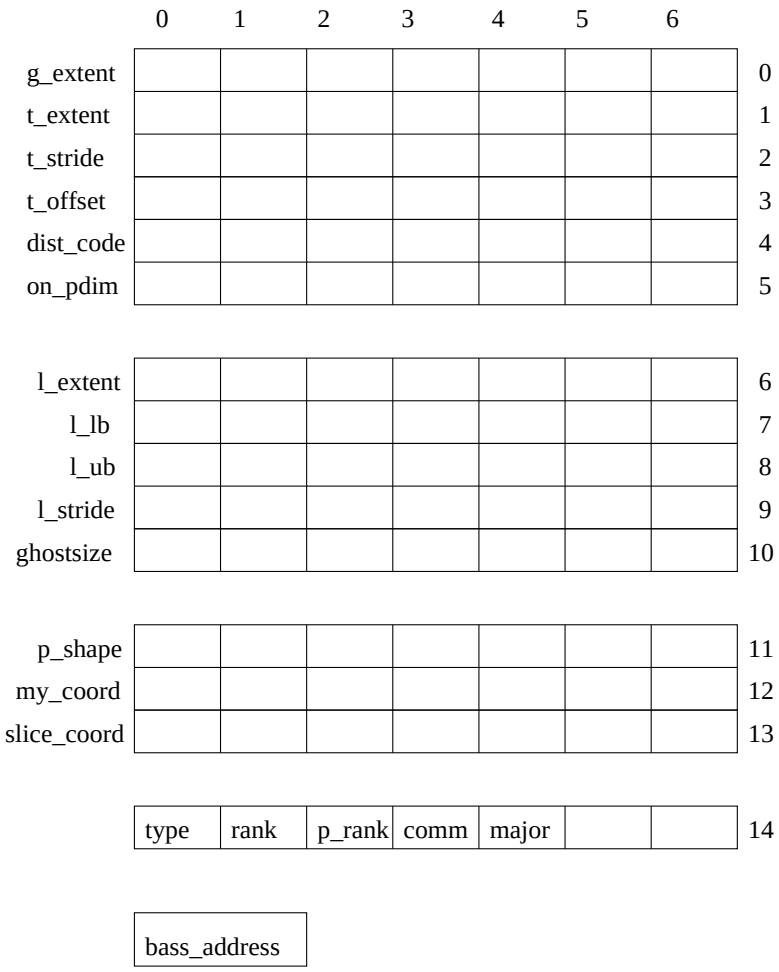


Figure 6.1: A practical distributed array descriptor

```
MPI_comm comm;
int majority;
```

Together with base address, we see a DAD represented as in Figure 6.1.

6.2.2 DAD constructor algorithms

Now, we study the algorithm for *l_extent*, *l_lb*, *l_ub*, and *l_stride*. First, we recall that we actually had a procedure for em l_extent in the last lecture notes. Second, it is clear, in our current context, *l_stride* = *t_stride* for `BLOCK` distribution.

What might not be so obvious is for `CYCLIC` distribution,

```
l_stride = lcm(t_stride, p_shape)/p_shape
```

where *lcm()* is the least common multiple. This is equivalent to say, if

```
a0, a0+d1, ..., a0+m*d1, ...
```

and

```
b0, b0+d2, ..., b0+n*d2, ...
```

meet, they meet at every other *lcm(d1,d2)* position in the axis. For instance, d1=4, d2=6, *lcm(d1,d2)* = 12

```
1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61,...
   -       --       --       --
3,9,15,21,27,33,39,45,51,57,63,...
  -    --    --    --
```

This is clear if we think of two sequences of stridden points on axis.

```
*---*---*---*---*---*---*---*---*---*
*-----*-----*-----*-----*-----*-----*
```

and consider the effect of `CYCLIC` distribution on local memory allocation: a processor gets its next element from every other p_shape template element. We see the claim for *l_stride* holds.

The algorithm for *l_lb* and *l_ub*. You'll see it's quite similar to one of previous homeworks. This algorithm is implemented as part of DAD constructor function invoked at *runtime*. We'll present its core here followed by an illustration.

Consider a one dimensional array distributed onto a one dimensional processor group, suppose those global parameters hava been collected in DAD, for processor i (the index to the processor group), the following procedure computes *l_lb* and *l_ub*:

```
switch (distCode) {
   case 1: /* block */
      w = ceil(t_extent/p);
      l_t = w*(i-1);     /* note i is processor number */
      u_t = w*i-1;
      if u_t < t_offset or t_stride*(g_extent-1)+t_offset < l_t
         l_lb = l_ub = l_stride = 0;
      else {
         j = ceil((l_t-t_offset)/t_stride);
         /* determine min j in 0:g_extent-1,
            such that t_stride*j+t_offset >= l_t. */
         if (j<0) j=0;
         l_lb = t_stride*j+t_offset - l_t;   /* 0-based local index */
         j = floor((u_t-t_offset)/t_stride);
         /* the max j in 0:g_extent-1, t_stride*j+t_offset <= u_t. */
         if (j>g_extent-1) j=g_extent-1;
         l_ub = t_stride*j+t_offset - l_t;   /* 0-based local index */
```

```
    }
    break;
case 2: /* cyclic */
    o = t_offset+1;
    eea(t_stride,-p,&x0,&y0,&d);  /* extended Euclidean algorithm */
    if (d does not divide (i-o))
       l_lb = l_ub = l_stride = 0;
    else {
       a0 = ceil( (d*(g_extent-1) - (i-o)*x0)/(-p));
       a1 = floor( -(i-o)*x0/(-p));
       b1 = floor( -(i-o)*y0/t_stride);   /* t_stride > 0 */
       b0 = ceil( (d*(t_extent-i)/p - (i-o)*y0)/(-t_stride));
       if (intersection(a0,a1,b0,b1,&l,&u)==0)
          l_lb = l_ub = l_stride = 0;
       else {
          l_lb = y0*(i-o)/d -u*t_stride/d;
          l_ub = y0*(i-o)/d -l*t_stride/d;
       }
    }
    break;
case 3: /* collapsed */
    l_lb = t_offset;
    l_ub = t_stride*(g_extent-1)+t_offset;
}
```

where `eea(a,b,x0,y0,d)` is a procedure implementing the extended Euclidean algorithm as described in [17]. More specifically, for given integers em a and b, it finds x_0, y_0, and d, such that $a \cdot x_0 + b \cdot y_0 = d$, where $d = gcd(a, b)$. Since this algorithm is essential for effective treatment of `CYCLIC` distribution, we will devote a subsection to it later.

`intersection(a0,a1,b0,b1,l,u)` finds intersection [l,u] of two intervals [a0,a1] and [b0,b1], returning 0 if the intersection is empty.

6.2.3 Extended Euclidean Algorithm

For Diophantine equation

$$a \cdot x + b \cdot y = c \tag{6.1}$$

let $d = gcd(a, b)$. We have: *equation 6.1 has a solution iff* $d \mid c$.

The Extended Euclidean Algorithm is to compute $d, x_0, and\ y_0$, such that $a \cdot x_0 + b \cdot y_0 = d$. If $d \mid c$, then the general solution to equation 6.1 is

$$\begin{cases} x = c' \cdot x_0 + b' \cdot t \\ y = c' \cdot y_0 - a' \cdot t \end{cases}$$

where t is any integer, and $(a', b', c') = (a/d, b/d, c/d)$.

Further, if x and y are bounded as

$$0 \le x \le X, \quad 0 \le y \le Y \tag{6.2}$$

the intersection of the following two sets of t's

$$0 \leq c' \cdot x_0 + b' \cdot t \leq X$$
$$0 \leq c' \cdot y_0 - a' \cdot t \leq Y$$

will give the solution to equation 6.1 under constraint 6.2.

The Algorithm:

1. Initialize $(x_1, y_1, c_1) \longleftarrow (1, 0, \mid a \mid)$; $(x_2, y_2, c_2) \longleftarrow (1, 0, \mid b \mid)$;
2. If $c_2 = 0$, set $(x_0, y_0, d) \longleftarrow (sig(a) \cdot x_1, sig(b) \cdot y_1, c1)$, done.
3. Set $q = \lfloor \frac{c_1}{c_2} \rfloor$;
   $(t_1, t_2, t_3) \longleftarrow (x_1, y_1, c_1) - q \cdot (x_2, y_2, c_2)$;
   $(x_1, y_1, c_1) \longleftarrow (x_2, y_2, c_2)$;
   $(x_2, y_2, c_2) \longleftarrow (t_1, t_2, t_2)$;
   goto step 2.

The basic idea behind this algorithm can be viewed as maintaining

$$\begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \end{pmatrix} \begin{pmatrix} \mid a \mid \\ \mid b \mid \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$$

by Gaussian elimination with basic Euclidean algorithm applied on c_1 and c_2.

6.3 DAD implementation

Related to our course project, I'd like to discuss a possible implementation of the DAD in a compilation system.

We know in general, `DAD` = (base_address, info(15,7)). Since bass address is normally passed separately into runtime functions, we would just consider `DAD` as the `15 x 7` data object for the discussion.

From each processor's point of view, a global storage, called 'DAD_pool', may be conveniently employed to maintain the collection of `DAD`s during the execution of a node program. This DAD_pool is accessible by all runtime functions, and possibly by node program. In C, it may be implemented as

```
extern int DAD_pool[1000*15*7];
```

while in Fortran it could be

```
INTEGER DAD_pool(1000*15*7)
COMMON /DADs/DAD_pool
```

In node program, a DAD is simply represented by an integer (*handle*), thue, a typical approach to create a DAD might be

```
DAD_X = DADalloc()
CALL set_DAD_values(DAD_X,all necessary data quantities
                                collected from HPF program)
```

The `DADalloc()` keeps track usage of DAD_pool, returns an index to the DAD_pool. Clearly, `DADalloc()` should internally record the usage status of DAD_pool.

`set_DAD_values()` fills the 15*7 elements starting from DAD_pool(DAD_X). `DAD_X` then can be used to call other runtime functions. For instance,

```
...
FORALL (i=3:n:m) X(i) = i
...
```

could be turned into node program segment as

```
...
CALL loop_bounds(DAD_X,1,3,n,m,l,u,s)  ! the 1 is for dimension
DO i=l,u,s
   X(i) = local2global(DAD_X,1,i)     ! 1 designates dimension.
END DO
...
```

Each runtime function sees DAD_pool and can access proper information, given an index to it. Note, this is the first node program segment in this course that show *complete* parameters to a non trivial runtime function. We recall that we always had '...' in those runtime function calls previously.

Homework: Design and implement the following runtime functions:

```
local2global(DAD,dim,i)
global2local(DAD,dim,i)
loop_bounds(DAD,dim,gl,gu,gs,ll,lu,ls)
```

Chapter 7

Runtime functions

We address the second component of compiler/runtime interface, namely functions and calling sequence specifications. After this lecture, I think we should have done enough to begin working on our term project. Thus, a setting of the project is described after the main material.

As mentioned in the first lecture, our runtime functions may be classified into four categories. It should be understood that while the functions described here indeed constitute a working runtime, not all runtime must have precisely the same set of functions, though any runtime supporting HPF system should more or less provides a similar functionality.

7.1 Interface functions to DAD

```
SUBROUTINE set_array_info(dad,type,rank,p_rank,comm,major)
INTEGER dad,type,rank,p_rank,comm,major
```

Description: use *dad* handle to access DAD_pool(), set the last row of DAD table. We use 1 for `type=INTEGER`, 2 for `type=REAL`. Before calling this function, two other functions, `new_processor_grid()` and `dad_alloc()`, must be called to obtain *comm* and *dad*, respectively.

```
SUBROUTINE set_dim_info(dad,dim,g_lb,g_ub,collapsed,a_stride,
          a_offset,t_lb,t_ub,dist_code,p_dim,ghostsize,num_procs)
INTEGER dad,dim,g_lb,g_ub,collapsed,a_stride
INTEGER a_offset,t_lb,t_ub,dist_code,p_dim,num_procs
```

Description: set array dimension and alignment/distribution information for one dimension. *g_lb* and *g_ub* are bounds of the array declared in HPF program, which will be turned into `g_extent` in DAD; *t_lb* and *t_ub* are corresponding template bounds, unspecified when *collapsed*=1 indicating the array dimension is collapsed; *a_stride* and *a_offset* are alignment stride and offset, to be converted to *t_stride* and *t_offset* in the DAD. For `dist_code`, we use -1 for `block`, $+1$ for `cyclic`, and 0 for `*`.

This function should be called multi times for a multidimensional array.

```
SUBROUTINE set_dad_done(dad)
```

Description: compute *l_extent*, *l_lb*, *l_ub*, *l_stride*, and *my_coord* for all dimensions, using the algorithm presented in last chapter. A dad is created after this routine.

Thus, to create a DAD, the above three functions need to be called. It's possible, though, to design on function that does all.

```
INTEGER FUNCTION dad_copy(dad_source)
```

Description: returns a new DAD handle with DAD content identical to *dad_source*. An error message is issued if no more DAD space.

Note, this function will call `dad_alloc()`. The reason we want `dad_alloc()` to be explicitly called before `set_array_info()`, and implicit here is a matter of taste. For one thing, we want as few runtime functions appearing in node programs as possible; for the other, we want to emphasize the semantics of setting some information into a data object.

```
INTEGER FUNCTION section_dad(dad,dim,low,up,step)
```

Description: returns a new DAD handle, which inherits content of *dad* but with some modification at dimension *dim* according to the sectioning information, low:up:step. Note, the compiler should provide normalized *low* and *up* against declared lower bound of the array dimension.

Procedure to do this modification could be something like

```
g_extent = (up - low)/step + 1
t_offset = t_stride * low + t_offset
t_stride = t_stride * step
l_lb = ...
l_ub = ...
l_stride = ...
```

7.2 Index conversion functions

```
(8)   INTEGER FUNCTION global2local(dad,dim,i)
```

Description: returns the local index for global index *i* on dimension *dim*. For non replicated array, exactly one processor should have a corresponding local index. Return -1 for 'I don't have this element'. (note, local array is 0-based. Also note, -1 here is not an error condition, error only if every process returns -1, which means no body has that element !)

```
INTEGER FUNCTION local2global(dad,dim,i)
```

Description: returns the global index for local index em i on dimension *dim*. It's possible that some local index (for non effective local element) has no corresponding global index. A runtime error message is issued for that case. (note, different processors usually return different values, all valid, for the same local index)

```
SUBROUTINE loop_bounds(dad,dim,gl,gu,gs,l,u,s)
```

Description: returns the local triplet *l:u:s* corresponding to global *gl:gu:gs* on dimension *dim*. 0:-1:1 is returned if no local correspondence.

This function is kind of 'global2local()' conversion function. It maps a global triplet to a set of local ones. The name of the function may not properly reflect its semantics. We use it, since it's convenient for determining local `DO` loop bounds out of global `FORALL` indices.

```
SUBROUTINE loop_upper_bound(dad,dim,gl,gu,gs,ub)
```

Description: this is a variation of `loop_bounds()`. It basically returns ub = (u-l)/s, or -1. Sometime this function is convenient for node program generation.

7.3 Data movement functions

```
SUBROUTINE remap(x,dad_x,y,dad_y)
```

Description: move data from x described by *dad_x* to y described by dad_y. Note, globally the two data objects described by *dad_x* and *dad_y* should have shape conformance (this does not mean X and Y must conform), otherwise a runtime error message is issued.

```
SUBROUTINE bcast(sink,x,dad_x,index)
(choice) (OUT) sink
INTEGER (IN) dad
INTEGER (IN) index()
```

Description: broadcast the global array element *X(index)* into replicated scalar *sink*. Size of index should match rank of the array. Care should be taken for replicated *X(index)*.

```
SUBROUTINE edge2ghost(x,x_dad,dim,amount)
INTEGER (INOUT) x_dad
INTEGER (IN) dim, amount
```

Description: send data from edge of x to the ghost area of the neighbor processor along dimension *dim*. The width of the edge is specified by *amount*. *x_dad* will be modified to reflect the resulting situation.

7.4 Other functions

```
INTEGER FUNCTION new_processor_grid(p_rank,p_shape)
INTEGER (IN) p_rank
INTEGER (IN) p_shape(p_rank)
```

bf Description: returns a handle to a newly created processor group, given the shape of it. With MPI as underline communication system, the handle could be an MPI communicator (INTEGER in Fortran). This might cause problem if runtime is mixed language programmed, since communicator is represented differently in Fortran and C. Thus, a general runtime may maintain its own group representations.

```
SUBROUTINE minihpf_init()
```

Description: initialize runtime system. This is the first function to call.

```
SUBROUTINE minihpf_done()
```

Description: clean up minihpf runtime system.

```
INTEGER FUNCTION dad_alloc()
```

Description: returns a handle to a DAD. In our implementation, The handle is actually an index to `DAD_pool`. If no more DAD space exists, a runtime error message is issued, program exits. (I initially wanted it to return -1 for 'no space', but I realize there is no point doing that way, since 'no space' implies 'nothing can be done further'. Thus, a simple exit is more natural, which also simplifies node program — no checking is need after calling `dad_alloc()`.)

```
INTEGER FUNCTION malloc(size)
```

Description: allocate memory for given size, returns a base address of allocated memory, or a runtime error message is issued if no space available.

During runtime, some temporary arrays may be need. This function is intended to accommodate that need. For `FORTRAN 77` as target, the temporary storage may be implemented as a big `COMMON` block, and `malloc()` returns an index to that block.

```
INTEGER FUNCTION myid(comm)
```

Description: returns MPI rank in communicator. This function is convenient for controlling I/O operations.

```
LOGICAL FUNCTION on_me(dad,dim,i)
```

Description: determines if global array index i at dimension *dim* is distributed on this processor. Clearly, this can be viewed as a variation of global2local().

7.5 Examples of node programs with the runtime functions

One of the purposes of taking a *runtime oriented* approach in this course is to enable students readily write down a node program manually for a given HPF program. Understanding and producing the correspondence between an HPF program and a node program are considered essential for compilation system construction. With the runtime functions discussed so far, we'll see how they are actually used, and work in concert, in terms of *complete* node programs. (We only talked *segments* of node programs before.)

Consider the following minihpf program

```
      PROGRAM minihpf_1
      REAL A(100)
!HPF$ PROCESSORS P(4)
!HPF$ TEMPLATE T(300)
!HPF$ ALIGN A(i) WITH T(2*i+5)
!HPF$ DISTRIBUTE T(BLOCK) ONTO P

      A = 1

      PRINT *,A

      END
```

A complete node program could be

```
      PROGRAM node_1
      include 'minihpf.h'
      include 'node_only.h'
      REAL A(75)
      INTEGER comm,p_shape(1),dad_a,l,u,s
      p_shape(1) = 4
      CALL minihpf_init()
      comm = new_processor_grid(1,p_shape)

      dad_a = dad_alloc()
      CALL set_array_info(dad_a,2,1,1,1,comm,1)
      CALL set_dim_info(dad_a,1,1,100,1,1,2,5,1,300,1,1,4)
      CALL set_dad_done(dad_a)

      CALL loop_bounds(dad_a,1,1,100,1,l,u,s)
      DO i=l, u, s
         a(i) = 1
      END DO

      PRINT *, 'From processor', myid(comm),': ',(a(i), i=l,u,s)

      CALL minihpf_done()
      END
```

As we can see, 9 runtime functions are involved. For the following,

```
      PROGRAM minihpf_2
      REAL A(100),B(100)
!HPF$ PROCESSORS P(4)
!HPF$ TEMPLATE T(300)
!HPF$ ALIGN A(i) WITH T(2*i+5)
!HPF$ ALIGN B(i) WITH T(3*i-1)
!HPF$ DISTRIBUTE T(BLOCK) ONTO P

      A = 1
      B = A

      PRINT *,B

      END
```

A complete node program could be

```
      PROGRAM node_1
      include 'minihpf.h'
      include 'node_only.h'
      REAL A(75),B(75)
      INTEGER comm,p_shape(1),dad_a,dad_b,l,u,s
      p_shape(1) = 4
      CALL minihpf_init()
      comm = new_processor_grid(1,p_shape)

      dad_a = dad_alloc()
      CALL set_array_info(dad_a,2,1,1,1,comm,1)
      CALL set_dim_info(dad_a,1,1,100,1,1,2,5,1,300,1,1,4)
      CALL set_dad_done(dad_a)

      CALL loop_bounds(dad_a,1,1,100,1,l,u,s)
      DO i=l, u, s
         a(i) = 1
      END DO

      dad_b = dad_alloc()
      CALL set_array_info(dad_b,2,1,1,1,comm,1)
      CALL set_dim_info(dad_b,1,1,100,1,1,3,-1,1,300,1,1,4)
      CALL set_dad_done(dad_b)

      CALL remap(a,dad_a,b,dad_b)
      CALL loop_bounds(dad_b,1,1,100,1,l,u,s)
      PRINT *, 'From processor', myid(comm),': ',(b(i), i=l,u,s)

      CALL minihpf_done()
      END
```

Besides the previous 9, `remap()` is also needed to run this program.

7.6 Project setting

Task: building a compilation system for a subset of HPF, called minihpf.
Purpose: acquiring an integrated working experience with a data parallel compiler/runtime system construction.

This work consists of two parts. We'll start from implementing the runtime functions specified today, and work on compiler part two weeks later.
Environment setting

- The platform: DEC Alpha cluster, MPI.

  Working directory: lxm/minihpf.

- minihpf.h — the file to be included in both node program and all runtime functions. This file currently declares `COMMON` blocks shared by node program and runtime.

```
INTEGER CAPACITY
PARAMETER (CAPACITY=1000)
INTEGER DAD_pool(CAPACITY*15*7)
INTEGER int_stack(CAPACITY*1000)
REAL real_stack(CAPACITY*1000)
EQUIVALENCE (int_stack,real_stack)
COMMON /DADs/DAD_pool
COMMON /STACK/real_stack
```

  It's possible for node program not seeing DAD_pool, and runtime not seeing stack (only knowing its size). The approach we take is for some general convenience.

- runtime_only.h — the file to be included in all runtime functions. mpif.h is included in this file. Currently, it is as

```
include 'mpif.h'
INTEGER dad_pool_ptr,stack_ptr
COMMON /POINTERS/dad_pool_ptr,stack_ptr
```

  Note, we use one pointer to keep track two 'virtual' stacks (real_stack,int_stack).

- node_only.h — the file to be included in node program. This file explicitly declares (not defines) runtime functions with proper types. If some runtime function needs to call other runtime function, it should declare the callee explicitly, instead of including 'node_only.h', since 'node_only.h' may contain the caller, which will cause a 'redeclaration' warning.

- minihpf.a — the runtime library.

- Makert — a make file to compile a newly developed file for runtime function/subroutine, and archive it to minihpf.a. If your program file is in Fortran, do

```
% make addf OBJ=file -f Makert
```

  if in C, do

```
% make addc OBJ=file -f Makert
```

- Makefile — a make file to compile and link a node program for execution.

```
% make EXEC=node
```

- wrapper.f — the wrapper file for Fortran interfaces to C functions. (we'll say more about the wrapper later.)
- minihpf.lst — the list of runtime functions/routines available so far. Each entry to this list is of format:

```
function/subroutine header, file name, author
```

  For instance, right now we already have "minihpf Runtime function/subroutine list":

```
SUBROUTINE minihpf_init(), global.f, lxm
SUBROUTINE minihpf_done(), global.f, lxm
INTEGER FUNCTION dad_alloc(), memory.f, lxm
INTEGER FUNCTION malloc(size), memory.f, lxm
INTEGER FUNCTION myid(comm), global.f, lxm
SUBROUTINE set_array_info(), dad.f, lxm
SUBROUTINE set_dim_info(), dad.f, lxm
```

Collaboration process

- I introduce ideas behind each runtime functions (simple ones first) in lectures and give assignment;
- You design and implement the functions in the shared directory (create a symbolic link from your directory to `~lxm/minihpf`); archive them to `minihpf.a`, register an entry to `minihpf.lst`, and if it's a function, declare it properly in `node_only.h`.
- For the initial couple of weeks, I'll provide testing node programs to test your runtime functions. For instance, the following program tests some of the functions I've written:

```
PROGRAM node
include 'minihpf.h'
include 'node_only.h'
INTEGER i, new_dad, real_base, int_base

CALL minihpf_init()

DO i=1,1000
   new_dad = dad_alloc()
   print *, new_dad
END DO

CALL minihpf_done()

END
```

It can be compiled, linked, and run under MPI environment. Of course the developer of the runtime should be able to tell if the output is as expected.

- After we have built up some basic runtimes and got some knowledge on program transformation, I'll give some minihpf programs and ask you to hand-translate them to node programs, and run.
- Finally, we investigate the strategy to replace the "hand translation" process by a program — the 'compiler'.

In case you want to write some runtime functions in C. A Fortran interface is needed, since our node program is in Fortran. DEC Fortran compiler provide some convention for calling C subprograms. (a handout, and an example.) Main points in the convention:

- declare C functions as EXTERNAL
- C functions defined with a trailing underscore, referenced without it.
- use %VAL() to pass value to C function, otherwise address is passed.
- C array is 0-based, Fortran 1-based.
- C array is stored in row major, Fortran column major.
- Use global variable of proper struct type to match Fortran `COMMON` block. The global variable must end with an underscore.

Chapter 8

Communication detection and insertion

After covering key issues in runtime library design, we now turn to compiler (translator) related tasks. As pointed out earlier, HPF directives and *owner computes rule* free the compiler from data partitioning and computation partitioning tasks. Communication detection and insertion becomes the central issue.

8.1 Motivating examples

Consider HPF statement

```
REAL X(16), Y(16)
...
X(1:15) = Y(2:16)
```

How this assignment statement is turned to a segment of node program depends on many factors. Data distribution is obviously a primary one. The following instances provide some motivation.

1. A need for *shift* communication. Consider,

```
      REAL X(16), Y(16)
!HPF$ TEMPLATE T(48)
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE T(BLOCK) ONTO P
!HPF$ ALIGN (i) WITH T(i) :: X,Y
       ...
      X(1:15) = Y(2:16)
```

   we see the data layout in memory as

```
                P1          P2           P3           P4
X(1:15):   xxxxxxxxxxx|xxx--------|------------|------------
Y(2:16):   -yyyyyyyyyy|yyyy--------|------------|------------
```

A shift of Y by 1 location to the left (+1) is needed. Note, the shift is best visualized on template. It is, in general, not a shift on array. This notion will be clearer by later examples.

2. No communication needed, due to proper alignment.

```
REAL X(16), Y(16)
TEMPLATE T(48)
PROCESSORS P(4)
DISTRIBUTE T(BLOCK) ONTO P
ALIGN X(i) WITH T(i+1)
ALIGN Y(i) WITH T(i)
...
X(1:15) = Y(2:16)
```

Although the assignment statement appears like a shift operation, no actual communication is needed as shown in the following data layout.

```
X(1:15):   -xxxxxxxxxx|xxxx--------|------------|------------

Y(2:16):   -yyyyyyyyyy|yyyy--------|------------|------------
```

3. Some kind of 'difficult' communication is required.

```
REAL X(16), Y(16)
TEMPLATE T(48)
PROCESSORS P(4)
DISTRIBUTE T(BLOCK) ONTO P
ALIGN X(i) WITH T(i+1)
ALIGN Y(i) WITH T(i)
...
X(1:9:2) = Y(2:14:3)
```

has the following data layout in memory.

```
X(1:9:2)   -x-x-x-x-x--|------------|------------|------------

Y(2:14:3)  -y--y--y--y-|-y----------|------------|------------
```

Obviously, communication is needed, but a shift on template can not eliminate it. We call those communication requirement, that can not be met by a shift, *remap* communication.

Thus, for an assignment between two arrays/sections, three kinds of communication requirement may result.

- No communication;
- Shift communication, '+' to the left, '−' to the right;
- Remap communication.

4. Complicated situation may not imply remap communication. See

```
REAL X(16), Y(16)
TEMPLATE T(48)
PROCESSORS P(4)
DISTRIBUTE T(BLOCK) ONTO P
ALIGN X(i) WITH T(3*i-1)
ALIGN Y(i) WITH T(2*i+1)
...
X(1:9:2) = Y(2:14:3)
```

   with the following memory layout,

```
              _     _     _     _     _
X(1:9:2)   -x--x--x--x-|-x--x--x--x-|-x----------|------------

              _     _     _     _     _
Y(2:14:3)  --y-y-y-y-y-|y-y-y-y-y-y-|y-y-y-------|------------
```

   No communication is need, each processor has what it needs.

5. CYCLIC distribution introduces some new issue.

```
REAL X(16), Y(16)
TEMPLATE T(48)
PROCESSORS P(4)
DISTRIBUTE T(CYCLIC) ONTO P
ALIGN X(i) WITH T(3*i-1)
ALIGN Y(i) WITH T(2*i+1)
...
X(1:9:2) = Y(2:14:3)
```

   has memory layout as,

```
              _     _     _     _     _
X(1:9:2)   -x--x--x--x--x--x--x--x--x--------------------
           123412341234123412341234123412341234

              _     _     _     _     _
Y(2:14:3)  --y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-------------------
           123412341234123412341234123412341234
```

   It needs a shift, +1 or -3, on template. Note BLOCK distribution always results in shift in one direction, if needed.

6. Templates make difference. The following program is almost the same as the second one above.

```
REAL X(16), Y(16)
TEMPLATE TX(36)
TEMPLATE TY(48)
PROCESSORS P(4)
DISTRIBUTE (BLOCK) ONTO P :: TX,TY
ALIGN X(i) WITH TX(i+1)
ALIGN Y(i) WITH TY(i)
...
X(1:15) = Y(2:16)
```

   but it needs a remap communication, as shown below.

```
B
   X(1:15):   -xxxxxxxx|xxxxxxx--|---------|---------

   Y(2:16):   -yyyyyyyyyyy|yyyy--------|-----------|-----------
```

   You may argue that a shift -3 is good enough. Yes, true for these very specific array segments. What if the segments span across 3 processors ?

7. The last one.

```
REAL X(8), Y(8)
TEMPLATE TX(36)
TEMPLATE TY(48)
PROCESSORS P(4)
DISTRIBUTE (BLOCK) ONTO P :: TX,TY
ALIGN X(i) WITH TX(3*i+1)
ALIGN Y(i) WITH TY(4*i-1)
...
X(1:7) = Y(2:8)

X(1:7):   ---x--x--|x--x--x--|x--x-----|---------

Y(2:8):   ------y---y-|--y---y---y-|--y---y-----|-----------
```

   No communication.

One of our compiler's job is to detect different communication requirements based on an analysis of the program, and generate proper node program segments correspondingly.

8.2 A theory of communication detection

We want to develop some conditions that distinguish the three communication patterns. We observe the following factors affecting the condition

- description of array sections (and their offsets from lower bound of the array index), `xl:xu:xs, xo; yl:yu:ys, yo`
- size of templates (we assume the sizes are multiples of p), `tx; ty`
- alignment parameters `ax,bx; ay,by`
- distribution mode `BLOCK, CYCLIC`

Among the 3 communication patterns, 'shift communication' is the key, since

- 'no communication' can be viewed as 'shift communication' with shift_amount=0;
- 'not shift communication' $\longrightarrow$ 'remap communication'

Thus, our task becomes: determine the condition for shift communication,

```
if the condition holds, then
   calculate shift_amount;
   if shift_amount=0 then
      we have 'no communication' condition;
   end if
else
   we have a 'remap communication' condition;
end if
```

Although there are many parameters involved in this model, we observe that status of section elements on templates is what we are ultimately interested in. This status is described by

- size of the templates, and number of processors the templates are distributed on; `tx, ty, p`
- position of the first section element on template; `xo, yo`
- stride on template `sx = ax*xs, sy = ay*ys`
- distribution mode, `BLOCK` or `CYCLIC`

shift-homomorphic condition: assume template `TX(tx)` and `TY(ty)` are distributed over p processors,

Two array sections `X(xl:xu:xs)` and `Y(yl:yu:ys)` are *shift-homomorphic* with respect to `TX` and `TY`, and alignment parameters `(ax,bx)` and `(ay,by)`, respectively, if

They have the same extent (number of elements), and

$$\frac{ax \cdot xs}{ay \cdot ys} = \frac{tx}{ty} \tag{8.1}$$

for both `TX` and `TY` being block-distributed;

$$ax \cdot xs = ay \cdot ys \pmod{p} \tag{8.2}$$

for both `TX` and `TY` being cyclic-distributed.

We want to show shift-homomorphic condition is the condition for shift communication, namely, if two array sections satisfy the condition, it is always possible to shift X or Y some amount along its template, so that corresponding section elements lie in the same processor.

For BLOCK distribution

Idea of the proof: let x and y be the positions of first element of `X(xl:xu:xs)` on template `TX`, `Y(yl:yu:ys)` on template `TY`, respectively. If we can show that

> there exits a position y' on template `TY`, such that if we shift array `Y` along `TY` so that the first element of `Y(yl:yu:ys)` lies at position y' then each processor will contain the same number of elements from both `X(xl:xu:xs)` and `Y(yl:yu:ys)`,

we done !

```
.....x..|........|........|........
.......y'....|.....y.......|............|............
       <------------
```

To this end, we claim: $y' = \lceil \frac{x \cdot ty}{tx} \rceil$.
Proof: define w1=tx/p and w2=ty/p, and $\Phi(i) = \lceil \frac{i \cdot ty}{tx} \rceil$. Let

```
i, i+sx, i+2*sx, ..., i+k*sx
```

be any consecutive subsection of X(xl:xu:xs). We show if

$$(n-1) \cdot w1 + 1 \leq i, \quad and \quad i + k \cdot sx \leq n \cdot w1$$

then

$$(n-1) \cdot w2 + 1 \leq \Phi(i), \ and \ \Phi(i) + k \cdot sy \leq n \cdot w2$$

Clearly, if this is true, each processor will have the same number of elements from sequences

```
x, x+sx, x+2*sx, ..., x+m*sx
```

and

```
y', y'+sy, y'+2*sy, ..., y'+m*sy
```

Indeed, from $(n-1)\cdot w1+1 \le i$, we have

$$((n-1)\cdot w1+1)\cdot\frac{ty}{tx} \le i\cdot\frac{ty}{tx}$$

$$\lceil((n-1)\cdot w1+1)\cdot\frac{ty}{tx}\rceil \le \lceil i\cdot\frac{ty}{tx}\rceil = \Phi(i)$$

$$\lceil(n-1)\cdot w2+\frac{ty}{tx}\rceil \le \Phi(i)$$

note w1*ty/tx=w2.

$$(n-1)\cdot w2+1 \le (n-1)\cdot w2+\lceil\frac{ty}{tx}\rceil \le \Phi(i)$$

on the other hand, from $i+k\cdot sx \le n\cdot w1$, we have

$$(i+k\cdot sx)\cdot\frac{ty}{tx} \le n\cdot w1\cdot\frac{ty}{tx}$$

$$\lceil\frac{i\cdot ty}{tx}\rceil + k\cdot sy = \lceil\frac{(i+k\cdot sx)\cdot ty}{tx}\rceil \le \lceil\frac{n*w1*ty}{tx}\rceil$$

note s1*ty/tx = s2

$$\Phi(i)+k\cdot sy \le \lceil\frac{n\cdot w1\cdot ty}{tx}\rceil = n\cdot w2$$

Q.E.D.

Clearly, $shift_{amount} = y' - y = \lceil\frac{x\cdot ty}{tx}\rceil - y$.

For CYCLIC distribution

Recall condition sx = sy (mod p).

Once again, let x and y be initial positions (on templates) of the first elements from `X(xl:xu:xs)` and `Y(yl:yu:ys)`. Let y' be any feasible position on template `TY`, such that

$$x = y' \ (mod \ \ p)$$

we claim if we shift `Y` such that first element of `Y(yl:yu:ys)` lies on y', then `X(i)`, i=xl,xu,xs, will be in the same processor as `Y(j)`, j=yl,yu,ys.

$$x = y' \ (mod \ \ p) \longrightarrow X(xl) \ is \ \ with \ \ Y(yl)$$

the next corresponding elements `X(xl+xs)` and `Y(yl+ys)` are at positions $x+sx$ and $y'+sy$, respectively. Since

$$x = y' \ (mod \ \ p), \ and \ \ sx = sy \ (mod \ \ p)$$

imply

$$x + sx = y' + sy \ (mod \ \ p)$$

we see `X(xl+xs)` and `Y(yl+ys)` are in the same processor, too. Obviously, this analysis can be applied recursively until all elements are exhausted.

`Q.E.D.`

Note, we have used *feasible position* to describe y'.

```
1234123412341234
-x-x-x-x--------

12341234123412341234
----y-----y-----y-----y-
```

The second and 6th positions on TY are feasible, 10th is not. Thus, we need a way to determine the closest feasible position and induced 'shift_amount' in this case.

```
if ((x-1)%p + 1) > ((y-1)%p + 1) and y > x then
   shift_amount = (y-1)%p+1 - (p - ((x-1)%p+1))
else
   shift_amount = (y-1)%p - (x-1)%p
```

for the above example, shift_amount= 1 - (4-2) = -1. Two more examples,

```
1234123412341234
x-x-x-x-x--------

12341234123412341234
----y-----y-----y-----y-
```

shift_amount = 0 - 0 = 0, no communication

```
1234123412341234
------x-x-x-x-x-

12341234123412341234
----y-----y-----y-----y-
```

shift_amount = 0 - 2 = -2.

To apply the above result to compilation practice — compiler algorithm, upon seeing

```
...
X(xl:xu:xs) = Y(yl:yu:xs)
...
```

do

step 1. Insert a call to runtime function as

```
CALL loop_bounds(dad_x,1,l,u,s,ll,lu,ls)
```

where dad_x is a descriptor for distributed array X.

step 2. Check if communication is detectable. If not, go to 6.

step 3. Detect communication pattern between arrays X and Y under this assignment statement. If shift communication is detected, the shift_amount is also determined.

step 4. If no communication, insert

```
CALL loop_bounds(dad_y,1,l,u,s,lly,luy,lsy)
iy = lly
DO i = ll,lu,ls
   X(i) = Y(iy)
   iy = iy + lsy
END DO
```

where dad_y is a descriptor for distributed array Y.

step 5. Else if shift communication, insert

```
...
CALL dad_copy(dad_tmpy,dad_y)
CALL edge2ghost(y, dad_tmpy, 1, amount)
CALL loop_bounds(dad_tmpy,1,l,u,s,lltmpy,lutmpy,lstmpy)
itmpy = lltmpy
DO i = ll,lu,ls
   X(i) = Y(itmpy)
   itmpy = itmpy + lstmpy
END DO
```

where dad_tmpy describes the result of Y after edge2ghost().

step 6. Else remap communication is needed, insert

```
dad_xs = section_dad(dad_x,...)
dad_ys = section_dad(dad_y,...)
CALL remap(tmpx,dad_xs,y,dad_ys)
DO i = ll,lu,ls
   X(i) = tmpx(i)
END DO
```

where dad_xs and dad_ys are descriptors for the sections involved.

Finally, let's see the node programs for preceding examples (7), (5), and (3).

```
! for convenience, I note the interfaces for
!    set_array_info(dad,type,rank,p_rank,comm,major)
!    set_dim_info(dad,a_dim,g_lb,g_ub,collapsed,a_stride,a_offset,
!   +             t_lb,t_ub,dist_code,p_dim,ghostsize,num_procs)
   PROGRAM NODE_7
   include 'minihpf.h'
   include 'node_only.h'
   REAL X(9),Y(12)
   INTEGER comm,p_shape(1),dad_x,dad_y
   INTEGER ll,lu,ls,lly,luy,lsy,iy
   p_shape(1) = 4
   CALL minihpf_init()
   comm = new_processor_grid(1,p_shape)

   dad_x = dad_alloc()
   CALL set_array_info(dad_x,2,1,1,comm,1)
   CALL set_dim_info(dad_x,1,1,8,0,3,1,1,36,1,1,0,4)
   CALL set_dad_done(dad_x)

   dad_y = dad_alloc()
   CALL set_array_info(dad_y,2,1,1,comm,1)
   CALL set_dim_info(dad_y,1,1,8,0,4,-1,1,48,1,1,0,4)
   CALL set_dad_done(dad_y)

   CALL loop_bounds(dad_x,1,1,7,1,ll,lu,ls)
   CALL loop_bounds(dad_y,1,2,8,1,lly,luy,lsy)
   iy = lly
   DO i = ll,lu,ls
      X(i) = Y(iy)
      iy = iy + lsy
   END DO
   CALL minihpf_done()
   END

   PROGRAM NODE_5
   include 'minihpf.h'
   include 'node_only.h'
   REAL X(12),Y(12),tmpy(12)
   INTEGER comm,p_shape(1),dad_x,dad_y,dad_tmpy
   INTEGER ll,lu,ls,lly,luy,lsy,iy
   p_shape(1) = 4
   CALL minihpf_init()
   comm = new_processor_grid(1,p_shape)
```

```
   dad_x = dad_alloc()
   CALL set_array_info(dad_x,2,1,1,comm,1)
   CALL set_dim_info(dad_x,1,1,16,0,3,-1,1,48,2,1,0,4)
   CALL set_dad_done(dad_x)

   dad_y = dad_alloc()
   CALL set_array_info(dad_y,2,1,1,comm,1)
   CALL set_dim_info(dad_y,1,1,16,0,2,1,1,48,2,1,1,4)
   CALL set_dad_done(dad_y)

   CALL loop_bounds(dad_x,1,1,9,2,ll,lu,ls)
   dad_tmpy = dad_copy(dad_y)
   CALL edge2ghost(Y,dad_tmpy, 1, +1)
! note edge2ghost() will adjust dad_tmpy according to result of shift
   CALL loop_bounds(dad_tmpy,1,2,14,3,lly,luy,lsy)
   iy = lly
   DO i = ll,lu,ls
      X(i) = Y(itmpy)
      iy = iy + lsy
   END DO

   CALL minihpf_done()
   END

   PROGRAM NODE_3
   include 'minihpf.h'
   include 'node_only.h'
   REAL X(12),Y(12),tmpx(12)
   INTEGER comm,p_shape(1),dad_x,dad_y,dad_xs,dad_ys
   INTEGER ll,lu,ls,lly,luy,lsy,iy
   p_shape(1) = 4
   CALL minihpf_init()
   comm = new_processor_grid(1,p_shape)

   dad_x = dad_alloc()
   CALL set_array_info(dad_x,2,1,1,comm,1)
   CALL set_dim_info(dad_x,1,1,16,0,1,1,1,48,1,1,0,4)
   CALL set_dad_done(dad_x)

   dad_y = dad_alloc()
   CALL set_array_info(dad_y,2,1,1,comm,1)
   CALL set_dim_info(dad_y,1,1,16,0,1,0,1,48,1,1,0,4)
   CALL set_dad_done(dad_y)

   CALL loop_bounds(dad_x,1,1,9,2,ll,lu,ls)
   dad_xs = section_dad(dad_x,1,1,9,2)
   dad_ys = section_dad(dad_y,1,2,14,3)
   CALL remap(tmpx,dad_xs,y,dad_ys)
```

```
DO i = ll,lu,ls
   X(i) = tmpx(i)
END DO

CALL minihpf_done()
END
```

Applying the result to FORALL statement. Consider,

```
REAL X(..), Y(..)
PROCESSORS ..
TEMPLATE ...
ALIGN ...
DISTRIBUTE ...
...
FORALL (i=l:u:s) X(a0*i+b0) = Y(a1*i+b1)
...
END
```

Observe this FORALL statement is equivalent to

```
X(xl:xu:xs) = Y(yl:yu:ys)
```

where

```
xl = a0*l+b0,  xu = a0*u+b0, xs = a0*s
yl = a1*l+b1,  yu = a1*u+b1, ys = a1*s
```

We done !

Homework: As we have observed, once a shift-homomorphism condition is detected for `BLOCK` distribution, the *shift-amount* may not be unique. Derive a formula that determines the range of the *shift-amount*.

Chapter 9

Writing a compiler for `minihpf`

In lecture 7 (Chapter 7 of this notes), we described a general setting of our term project. As we are finishing the runtime part of the project, we'll discuss compiling (translation) techniques in this lecture.

We specify a subset of HPF, called `minihpf`, for a term project. Students are expected to write a compiler (translator) that turns a `minihpf` program into a node program in FORTRAN 77, calling runtime functions developed earlier in this course. Once compiled by a native FORTRAN 77 compiler and linked with the runtime, a collective execution of multiple copies of the node program should achieve the same semantical effect as the `minihpf` program.

A compilation strategy for this project is also discussed in the lecture.

9.1 The `minihpf` language

A complete syntax specification of `minihpf` is provided at end of this chapter. Here are some basic considerations behind the decisions made on this `minihpf`.

- It should be feasible as a term project to be completed in two months.
- Rationals on technical aspects:
  - cover most of key issues in data parallel compilation system construction, in particular:
    - The 'forest' — a complete process from source to target translation, instead of just some 'key technologies', and to be able to observe the generated node program to be linked with runtime and run ! This is in compliance with designed outcome of this course.
    - The 'trees' — the following technical issues will be delt with in the work, though may not be thoroughly discussed.
      - DAD

· global $\longleftrightarrow$ local index conversion
· data movement
· communication detection and insertion
· subprogram interface analysis
· memory management

We consider they are fundamental issues one must encounter when constructing a data parallel compilation system. (I regret I eventually gave up 'subprogram interface analysis', due to loss of two lectures for some business trips.)

– avoid low $\frac{importance}{effort}$ features.

Based on these thoughts, we have

- The language:
  - one dimensional arrays and templates
  - single processor grid
  - `REAL` and `INTEGER` data types
  - assignment statement with one term at right hand side
  - one index `FORALL` statement without masking
  - `DO` loops with control variable not used in body.
  - subroutines
  - no input statement
  - simple `PRINT` statement for output

9.2 Outline of `minihpf` compiler code

We may see two passes of processing by the compiler.

1. Analysis: gather information about the program, and record them properly.
2. Node program generation: based on the information gathered from the first pass and *translation strategy*, spell out corresponding statements sequence in node program for each `minihpf` statement.

Assuming we have a front-end (such as HPFfe) that can parse HPF programs into some intermediate representation (IR), and unparse IR for node program back to FORTRAN 77 program, then, the compiler program could look like:

```
main{int argc, char * argv[]) {

   input IR of a minihpf program;
```

```
    analyze(it);
    transform(it);    // into node program's IR
    output the transformed IR;

}
```

We discuss the algorithm involved in analyze() and transform() today, in particular, the interface between them.

9.3 Developing a translation scheme

Considering *whole array* as a special case of *array-section*, and see constant as a scalar value, we distinguish the following 13 cases to transform.

1. assignment statement — involving $3^2 - 2$ cases

   (1.1) scalar to scalar

   (1.2) scalar to array-element

   (1.3) array-element to scalar

   (1.4) array-element to array-element

   (1.5) scalar to array-section

   (1.6) array-element to array-section

   (1.7) array-section to array-section

   (we can not have 'array-section to scalar' or 'array-section to array-element', due to shape conformance requirement.)

2. `FORALL` statement, in its assignment statement,

   (2.1) `FORALL` index (scalar) to array-element

   (2.2) non `FORALL` index scalar to array-element

   (2.3) array-element to array-element

3. `PRINT` statement

   (3.1) scalar

   (3.2) array-element

   (3.3) array-section

Let's elaborate a translation strategy for them one by one

(1.1) scalar to scalar

```
s1 = s2     -->    s1 = s2
```

(1.2) scalar to array-element

```
x(a*i+b) = s     -->    it = global2local(dad_x,1,a*i+b)
                        IF (it .NE. -1) x(it) = s
```

(1.3) array-element to scalar

```
s = x(a*i+b)     -->    CALL bcast(s,x,dad_x,a*i+b)
```

(for simplicity, we assume bcast() takes a scaler as the 4th argument, since we only deal with one dimensional arrays.)

(1.4) array-element to array-element

```
x(a*i+b) = y(c*i+d)  -->  CALL bcast(s,dad_y,c*i+d)
                          it = global2local(dad_x,1,a*i+b)
                          IF (it .NE. -1) x(it) = s
```

Note, we should pick properly typed scalar for s. Let's use *is*, and *rs*, respectively.

(1.5) scalar to array-section

```
x(a1*i1+b1:a2*i2+b2:a3*i3+b3) = s
--> CALL loop_bounds(dad_x,1,a1*i1+b1,a2*i2+b2,a3*i3+b3,ll,ul,sl)
    DO i = ll,ul,sl
       x(i) = s
    END DO
```

Note, *ll*, *ul*, and *sl* should be typed `INTEGER` in the node program.

(1.6) array-element to array-section

```
x(a1*i1+b1:a2*i2+b2:a3*i3+b3) = y(a4*i4+b4)
--> CALL bcast(s,y,dad_y,a4*i4+b4)
    CALL loop_bounds(dad_x,1,a1*i1+b1,a2*i2+b2,a3*i3+b3,ll,ul,sl)
    DO i = ll,ul,sl
       x(i) = s
    END DO
```

Once again, properly typed scalar must be chosen for s.

(1.7) array-section to array-section

```
x(a1*i1+b1:a2*i2+b2:a3*i3+b3) = y(a4*i4+b4:a5*i5+b5:a6*i6+b6)
```

compiler function `detect_comm()` should be called to decide on one of three possible node program segments.

(a)
```
CALL loop_bounds(dad_x,1,a1*i1+b1,a2*i2+b2,a3*i3+b3,ll,ul,sl)
CALL loop_bounds(dad_y,1,a4*i4+b4,a5*i5+b5,a6*i6+b6,lr,ur,sr)
ir = lr
DO i = ll,ul,sl
   x(i) = y(ir)
   ir = ir + sr
END DO
```

(b)
```
CALL loop_bounds(dad_x,1,a1*i1+b1,a2*i2+b2,a3*i3+b3,ll,ul,sl)
dad_tmp = dad_copy(dad_y)
CALL edge2ghost(y,dad_tmp,1,amount)
CALL loop_bounds(dad_tmp,1,a4*i4+b4,a5*i5+b5,a5*i5+b5,lr,ur,sr)
ir = lr
DO i = ll,ul,sl
   x(i) = y(ir)
   ir = ir + sr
END DO
```

(c)
```
dad_s1 = section_dad(dad_x,1,a1*i1+b1,a2*i2+b2,a3*i3+b3)
dad_s2 = section_dad(dad_y,1,a4*i4+b4,a5*i5+b5,a6*i6+b6)
CALL remap(x,dad_s1,y,dad_s2)
```

(2.1) `FORALL` index (scalar) to array-element, assume `FORALL` triplet is gl:gu:gs.

```
x(a*i+b) = i
--> CALL loop_bounds(dad_x,1,a*gl+b,a*gu+b,a*gs,ll,ul,sl)
    DO i = ll,ul,sl
       x(i) = (local2global(i) - b)/a
    END DO
```

(2.2) non `FORALL` index scalar to array-element

```
x(a*i+b) = c
--> CALL loop_bounds(dad_x,1,a*gl+b,a*gu+b,a*gs,ll,ul,sl)
    DO i = ll,ul,sl
       x(i) = c
    END DO
```

(2.3) array-element to array-element

```
x(a*i+b) = y(c*i+d)
```

compiler function `detect_comm()` should be called to decide on one of three possible node program segments.

(a)
```
CALL loop_bounds(dad_x,1,a*gl+b,a*gu+b,a*gs,ll,ul,sl)
CALL loop_bounds(dad_y,1,c*gl+d,c*gu+d,c*gs,lr,ur,sr)
ir = lr
DO i = ll,ul,sl
   x(i) = y(ir)
   ir = ir + sr
END DO
```

(b)
```
CALL loop_bounds(dad_x,1,a*gl+b,a*gu+b,a*gs,ll,ul,sl)
dad_tmp = dad_copy(dad_y)
CALL edge2ghost(y,dad_tmp,1,amount)
CALL loop_bounds(dad_tmp,1,c*gl+d,c*gu+d,c*gs,lr,ur,sr)
ir = lr
DO i = ll,ul,sl
   x(i) = y(ir)
   ir = ir + sr
END DO
```

(c)
```
dad_xs = section_dad(dad_x,1,a*gl+b,a*gu+b,a*gs)
dad_ys = section_dad(dad_y,1,c*gl+d,c*gu+d,c*gs)
CALL remap(x,dad_xs,y,dad_ys)
```

(3.1) scalar

```
PRINT *,s   -->  PRINT *, 'From proc',myid(comm),':',s
```

(3.2) array-element

```
PRINT *, x(a*i+b)
-->  it = global2local(dad_x,1,a*i+b)
     IF (it .NE. -1) THEN
        PRINT *, 'From proc',myid(comm),':',x(it)
     END IF
```

(3.3) array-section

```
PRINT *, x(a1*i1+b1:a2*i2+b2:a3*i3+b3)
--> CALL loop_bounds(dad_x,1,a1*i1+b1,a2*i2+b2,a3*i3+b3,ll,ul,sl)
    PRINT *, 'From proc',myid(comm),':',(x(i),i=ll,ul,sl)
```

9.4 What else to be worried about ?

- interface between the two passes
  - analyze() figures out the 13 cases (in fact, 17 cases, considering different results of detect_communication()), and stores the result, together with related parameters, in a record;
  - transform() looks at the record, emits code as described above.
- storage management
  - some additional variables are needed in the node program. We need to have a naming convention to create them. A good news is that we only have to deal with scalars for these variables, due to the strong restriction that expression can only contain one term.
  - determination of local array sizes, especially in the case of non zero ghost area.

9.5 The interface — information in the record

The idea is that we want to have a unified record structure capable of representing all of the 17 cases. After examining the 17 cases, we may end up with the following

```
struct executable_stmt {
   int  case;
   statement *stmt-ptr;  // pointer to the statement in IR
   symbol *left;
   expression  *expr-1;
   expression  *expr-2;
   expression  *expr-3;
   symbol *right;
   expression  *expr-4;
   expression  *expr-5;
   expression  *expr-6;
}
```

Let's map the structure with cases (refer to section 9.3 for translation scheme).

(1.1) scalar to scalar

`case` =1, others `NULL`. `Transform()` sees this record, leaves the statement intact.

(1.2) scalar to array-element

`case=2`, `left` points to x entry in the symbol table, `expr-1` points to the expression `a*i+b`, `right` points to s entry in the symbol table. Others `NULL`.

(in case a*i+b is degenerated to simple variable or constant, conversion to expression needs to be done in `analyze()`.)

`transform()` sees this record, make up two statements, and replace original statement by the two. It will need additional variable *it* and *dad_x*. We note *it* can be reused, and some convenience may result if dad of an array with *name* is alway named as *dad_name*.

(1.3) array-element to scalar

`case`=3, `left`, `right`, and `expr-4` are effective.

(1.4) array-element to array-element

`case` =4, `left`, `expr-1`, `right`, and `expr-4` are effective. Note, `transform()` determines *is* or *rs* to be used.

(1.5) scalar to array-section

`case` =5, `left, expr-1, expr-2, expr-3`, and `right` are effective.

(1.6) array-element to array-section

`case` =6, `left, expr-1, expr-2, expr-3, right`, and `expr-4` are effective. Again, `transform()` determines *is* or *rs* to be used.

(1.7) array-section to array-section

As discussed previously, one of the three sub cases may result.

(a) `case` =7, `left, expr-1, expr-2, expr-3, right, expr-4, expr-5`, and `expr-6` are all effective.

(b) `case` =8, `left, expr-1, expr-2, expr-3, right, expr-4, expr-5`, and `expr-6` are all effective, though node program segment is different from case 7.

(c) `case` =9, `left, expr-1, expr-2, expr-3, right, expr-4, expr-5`, and `expr-6` are all effective.

(2.1) `FORALL` index (scalar) to array-element

`case` =10, `left, expr-1, right, expr-4, expr-5`, and `expr-6` are effective, with a understanding that `expr-4, expr-5`, and `expr-6` represent components of the `FORALL` triplet.

(2.2) non FORALL index scalar to array-element

`case` =11, `left, expr-1, right, expr-4, expr-5, and expr-6` are effective, with a understanding that `expr-4, expr-5`, and `expr-6` represent components of the `FORALL` triplet.

(2.3) array-element to array-element

Again, three sub cases may result.

(a) `case` =12, `left, expr-1, expr-2, right, expr-4, expr-5`, and `expr-6` are effective, with a understanding that `expr-4, expr-5`, and `expr-6` represent components of the `FORALL` triplet, and `expr-2` represents expression in right hand side element.

(b) `case` =13, `left, expr-1, expr-2, right, expr-4, expr-5`, and `expr-6` are effective, with a understanding that `expr-4, expr-5`, and `expr-6` represent components of the `FORALL` triplet, and `expr-2` represents expression in right hand side element.

(c) `case` =14, `left, expr-1, expr-2, right, expr-4, expr-5`, and `expr-6` are effective, with a understanding that `expr-4, expr-5`, and `expr-6` represent components of the `FORALL` triplet, and `expr-2` represents expression in right hand side element.

(3.1) `PRINT` a scalar

`case` =15, `left` is effective.

(3.2) `PRINT` an array-element

`case` =16, `left` and `expr-1` are effective.

(3.3) `PRINT` an array-section

`case` =17, `left, expr-1, expr-2`, and `expr-3` are in effect.

9.6 Specification part

We have more or less done with execution-part. Now we deal with the variable-spec-part with respect to HPF-directive-part.

How should analyze() prepare for transform() to specify local variables ?

We observe that in our case, most additional variables are transient. They are only needed for a segment of code, then can be reused. If you examine carefully, only 17 plus number of DADs additional variables are needed, no matter how big a minihpf program is !

We adopt the following assumptions/conventions:

- All variables used in a minihpf program keep the same names in node program, i.e., X(100) in minihpf program will also be called X(..) in node program, though extent may be changed.
- All arrays declared are aligned and distributed explicitly.
- DAD of an array with *name* will be designated by dad_*name*. For instance, REAL X1(200) will have dad_x1. (These are long lasting additional variable names.)
- In case of shift communication, the temporary DAD for an array will be dad_tmp, independent of array names.

- Returned local loop bounds for left hand side of assignment will be ll,ul,sl, while lr, ur, sr for right hand side, independent of array names.
- Use 'amount' for shift amount.
- Use 'ir' for right hand side array element index in local loop, while using 'i' for left hand side and DO loop index.
- Use 'dad_s1' and 'dad_s2' for array sections in case of remap.
- Use 'it' for returned value from global2local; use 'is' for integer scalar in bcast(), 'rs' for real scalar.

  The above has identified those transient names, 15 of them. There are 2 more long lasting variables:

- p_shape(1), comm

We should realize in general that a scheme for naming and keeping track of additional variables may be non trivial, though we are able to do some exhaustive analysis here. Any way, we may see the specification part of our node programs *always* look like something like,

```
PROGRAM name
include 'minihpf.h'
include 'node_only.h'
REAL  ... modified from minihpf program ...
INTEGER  ... modified from minihpf program ...
INTEGER p_shape(1), comm, amount, i, ir, dad_s1, dad_s2
INTEGER dad_tmp, ll, ul, sl, lr, ur, sr, it, is
REAL    rs
INTEGER  ... the DADs ...
```

`Analyze()` should provide enough information for `transform()` to complete those unfinished declarations, in particular, size of local arrays. Here is a proposed structure to record this information.

```
struct re_decl {
   statement * stmt-ptr;  // pointer to a type-spec-stmt
   local_size * list[];   // a list of pairs
}

struct local_size {
   symbol * symb-ptr;     // pointer to an array name
   int extent;            // local extent figured out by analyze()
}
```

Thus, `analyze()` prepares a bunch of re_decl's (number of them depends on the number of type-spec-stmts in miniphf program), and `transform()` looks at them one by one and modify the declaration based on 'extent'. For non array items in the type-spec-stmt, transform() just skips it.

Moreover, the 're_decl' also gives hint for the DAD variable declarations, namely every symbol appearing in the list[] will have a DAD.

9.7 Initialization and finishing

Not done yet ! We see the skeleton of our node program:

```
PROGRAM name
... specification part ...
p_shape(1) = ?
CALL minihpf_init()
comm = new_processor_grid(1,p_shape)

... a sequence of dad creations ...
    dad_x = dad_alloc()
    CALL set_array_info(dad_x,-,1,1,comm,?)
    CALL set_dim_info(dad_x,1,1,-,0,-,-,1,-,-,1,-,?)
    CALL set_dad_done(dad_x)

... the execution-part ...

CALL minihpf_done()
END
```

Thus, `analyze()` also provides: (1) a pointer to an expression for size of processor array, p_size. (2) The following structure for dad creation

```
struct dad_primitive {
   symbol * symb-ptr;  // pointer to symbol of an array
   expression * data_type;  // data type code
   expression * g_ub;
   expression * a_stride;
   expression * a_offset;
   expression * t_ub;
   expression * dist_code;
   expression * ghostsize;
}
```

for each array.

Putting all together, the interface between analyze() and transform() is:

- a list of re_decl;
- p_size;
- a list of dad_primitive;
- a list of executable_stmt;

9.8 Calling `detect_comm()` in `analyze()`

We have implemented a `detect_comm()` based on the algorithm developed in last lecture. The `analyze()` can simply call it by observing the following.

For a general `FORALL` statement as,

```
FORALL (i=l:u:s) X(a0*i+b0) = Y(a1*i+b1)
```

prepare parameters to call

```
detect_comm(detect_DIM * dad_x, detect_DIM * dad_y,
            int *l, int *u, int *s,
            int *a0, int *b0, int *a1, int *b1, int *amount);
```

where

```
typedef struct detect_DIM {
   int dist_code;
   int num_procs;
   int t_lb;
   int t_ub;
   int t_stride;
   int t_offset;
} detect_DIM;
```

and get return value: 0 no communication; 1 shift communication, amount is effective; 2 remap communication.

9.9 Minihpf specification

It's basically a direct stripped-down from Fortran 95 and HPF spec.

```
R1101 program               is  PROGRAM name
                                    [variable-spec-part]
                                    [HPF-directive-part]
                                    [execution-part]
                                END

R304  name                  is  letter [alphanumeric-character] ...

R302  alphanumeric-character    is  letter
                                or  digit

R501  variable-spec-part    is  type-spec-stmt
                                    [type-spec-stmt] ...

R502  type-spec             is  INTEGER entity-decl [, entity-decl] ...
                            or  REAL entity-decl [, entity-decl] ...
```

```
R504  entity-decl           is  name [( int-constant )]

      (note: we use default lower bound for arrays.)

H100  HPF-directive-part    is  !HPF\$ PROCESSORS name(int-constant)
                                  other-stuff
                                  [other-stuff] ...

H101  other-stuff       is  !HPF\$ TEMPLATE name(int-constant)
                            !HPF\$ DISTRIBUTE name(dist) ONTO name
                            !HPF\$ ALIGN name(name) WITH name(linear-expr)

H102  dist                  is  BLOCK
                            or  CYCLIC

H103  linear-expr           is  name
                            or  int-constant
                            or  int-constant * name
                            or  int-constant * name + int-constant
                            or  int-constant * name - int-constant

      Constraint: 'name' must be a scalar.

R208  execution-part        is  executable-stmt
                                    [executable-stmt] ...

R216  executable-stmt       is  assignment-stmt
                            or  forall-stmt
                            or  print-stmt

R735  assignment-stmt       is  variable = expr

R601  variable              is  name
                            or  subobject

R602  subobject             is  array-element
                            or  array-section

R615  array-element         is  name(subscript)

R616  array-section         is  name(subscript-triplet)

R619  subscript-triplet     is  [subscript] : [subscript] [:stride]

R617  subscript             is  linear-expr

R620  stride                is  linear-expr
```

```
R723  expr                 is  constant
                           or  variable

R306  constant             is  int-constant
                           or  real-constant

R404  int-constant         is  digit-string

R413  real-constant        is  digit-string . digit-string

R402  digit-string         is  digit [digit] ...

R754  forall-stmt          is  FORALL ( forall-triplet-spec ) assignment-stmt

R750  forall-triplet-spec  is  name = subscript : subscript [: stride]

R911  print-stmt           is  PRINT * , variable
```

Chapter 10

Compiler construction tools

Three packages were discussed. `HPFfe`, `Sage++`, and `SUIF`. Purpose of this lecture was two-folded: to help students work on project; to acquaint students with some of contemporary compiler construction tools out there in public domain. Nevertheless, I'll just include an introduction to `HPFfe` in this chapter, and refer readers to `http://www.extreme.indiana.edu/sage/` and `http://suif.stanford.edu/suif/suif.html` for `Sage++` and `SUIF`, respectively.

10.1 An overview of `HPFfe`

`HPFfe` is a compiler front-end for High Performance Fortran Version 1.0, developed by a joint effort of NPAC at Syracuse University, USA, PACT Lab at Harbin Institute of Technology, China, and PACT group at Peking University, China.

The main thrust of `HPFfe` is its complete coverage of HPF 1.0 syntax and most of compile-time checkable semantics. As a result, Fortran 90 is fully covered.

For a more detailed general description of HPFfe, the reader is referred to [12]. For technical implementation details, the reader is referred to [13]. This document describes how HPFfe may be used. Since HPFfe has adopted the internal data structure of Sage++ system, which in turn was evolved from Sigma system, the reader is referred to [14] for a discussion of its origine.

10.1.1 The intended users

- The primary users of HPFfe are HPF (or its subsets) compiler writers.
- HPF flavored compiler tool builders may also use HPFfe as a basis to start their work.
- Any language translation work that is to take HPF or its subsets as input will find HPFfe useful. For instance, Qiang Zheng and Wu Zhigang at Harbin Institute of Technology have crafted a Fortran to Java (f2j) translator based on this front-end.

10.1.2 Macro operations of `HPFfe`

There are four separate executable programs provided with HPFfe package. They are

- hpf2dep — takes as input any HPF program, say *filename.f*, produces *filename.dep* as output. `filename.dep` is a file that contains an intermediate representation of the input HPF program, in terms of an abstract syntax tree (AST) and some tables (among them the symbol table (ST) is the most important one).

```
Usage: hpfc2dep [-fixed] [-d] HPF_file.f
       -d : debugging switch
       -fixed : fixed format
```

- hpfsc — takes as input a file *filename.dep*, performs various semantics check while augments the AST to some extend, and outputs a modified version of `filename.dep`.

```
Usage: hpfsc [-vhcdoiD] HPF_file.dep
       -v : version
       -h : usage
       -m : write .mod files for module
       -c : compile the source files, but do not link
       -d : debugging parser
       -o : output filename
       -i : standard library compilation
       -D : display the module dependence
```

- dumpdep — takes as input a file *filename.dep*, either from output of hpf2dep or output of hpfsc, produces a readable form of it to `stdout`.

```
Usage: dumpdep [-d nnn][-m] dep_file
       -d : debugging switch
       -m : generate the readable AST in HTML format
```

- unparse — takes as input a file *filename.dep*,either from output of hpf2dep or output of hpfsc, generate an equivalent HPF program onto `stdout`.

```
Usage: unparse [-debug] [-verbose] [-version] dep_file
       -debug : Print useful debugging information
       -verbose : Turn on Verbose Mode
       -version : Print version number of compiled program
```

The relation of these operations is shown as in the Figure 1.4.

Besides these 4 executables, there is a class library `xsageop` also included in the package. This library is valuable for writing program transformation modules.

10.2 Installation

HPFfe has been successfully installed on the following platforms with various combinations of cc, gcc, lex, flex, yacc, and bison.

- IBM RS/6000, AIX 3.2.5
- Sun SparcStation 1+, OS 4.1.1
- Sun Sparc 10, OS 4.1.4
- Sun Sparc workstation, Solaries 2.4
- Sun Sparc workstation, Solaries 2.5
- DEC Alpha, OSF/1, V3.0
- HP, HP-UX
- SGI INDY, IRIX 5.3
- PC, Linux 2.0

HPFfe package comes in a compressed tar file (about 700K) named as hpfFrontEnd-x.xx.tar.gz, where x.xx designates version number. The highest version as time of this writing is 1.71. Here is a step by step installation and testing procedure.

1. Visit `http://www.npac.syr.edu/projects/pcrc/hpffe.html` and download the hpfFrontEnd-x.xx.tar.gz to your favorite directory, say `xyz`.
2. Do

   xyz% `gzip -d -c hpfFrontEnd-x.xx.tar.gz | tar -xvf -`

   As a result, you should see a subdirectory `hpfFrontEnd-x.xx` created in `xyz`. It's about 5MB in size.
3. If you do not like the directory name "hpfFrontEnd-x.xx", rename it now.
4. Go into the subdirectory, do

   hpfFrontEnd-x.xx% `configure`

   A Makefile will be generated automatically according to your system configuration.

5. Do

   hpfFrontEnd-x.xx% `make install`

   This will build the system and move executable programs to proper places. The resulting directory takes about 21MB disk space. After this step, you should be able to see the following directory structure

```
hpfFrontEnd-x.xx/
   INSTALL
   README
   Makefile
   configure
   bin/
      dumpdep
      hpf2dep
      hpfsc
      unparse
   src/
      basicop/
      hpf2dep/
      hpfsc/
      xsageop/
      tools/
         dump/
         unparse/
      include/
   include/
   testsuite/
   lib/
      libbasic.a
      libhpf.sl
      libxsage.a
```

   The four files in hpfFrontEnd-x.xx/bin correspond to the four macro operations introduced in the previous section. You can set your environment variable path to include the directory, if you want to conveniently apply them in other directories.

6. Some testing HPF files are also supplied with the package. To see if you have installed HPFfe successfully, do

hpfFrontEnd-x.xx% `make check`

which conducts a comprehensive test.

7. If you want to save some disk space, you may do

   hpfFrontEnd-x.xx% `make clean`

   now, which removes all object/executable files except those four in hpfFrontEnd-x.xx/bin. The resulting directory hpfFrontEnd-x.xx takes about 8MB spaces. (`make distclean` will remove all object/executables, bring the directory back to the status right after the package is opened (tarred).)

10.3 A skeleton of applications utilizing `HPFfe`

As indicated in section 10.1.1, primary users of HPFfe would be HPF compiler writers. Thus, by "application", we mean some program that perform some kind of program transformation on the intermediate representation of an HPF program. As special examples, three of the four macro operations of HPFfe (`hpfsc`, `dumpdep`, and `unparse`) may be viewed as such applications.

Input and output of an application: In general, we anticipate an application takes the .dep file generated by hpf2dep or hpfsc as input, and produces a possibly modified .dep file as output.

There are two possible ways to use the front-end.

One is to use the class library `xsageop`, which is an extension of Sage++ to our front-end, to write compiler transformation modules. The following is a sample skeleton.

```
...
#include "sage++user.h"

main(int argc, char * argv[])
{
  SgProject * project = new SgProject("test", argc, argv);

  for (int i=0; i< project->numberOfFiles(); i++) {
    ...
    SgStatement * s;

    s = global_file->firstStatement();
    ...
    global_file->saveDepFile("nodeProgram.dep");
  };
  return 0;
}
```

For use of `xsageop` classes, please visit Indiana's Sage++ site as I indicated in the beginning of the chapter. A good online documentation is provided.

The other way is more hacker oriented, which uses some primitive operations provided with the package. Although we do not encourage this approach, a general description is given below any way.

Since `.dep` file is to be input, which encompasses a rich set of data structures defined in header files in hpfFrontEnd-x.xx/include, those header files must be included before doing anything on the .dep file. We call these data structures *internal interface* between compilation modules and the front-end. HPFfe also provides some primitive functions that operate on the data structures. This functions exist as `libbasic.a` in hpfFrontEnd-x.xx/lib. We call the functions *external interface*. Thus, a generic framework of an application would look like (suppose .dep file is supplied on the command line as the first argument to your application)

```
...
#include "hpffe.h" /* with proper use of -I switch when compiling */
...
FILE *depFile;
PTR_FILE f = (PTR_FILE) calloc(1, sizeof(struct file_obj)));
...
depFile = fopen(argv[1], "r");
f->fid = depFile;
...
x_readNode(f);
...
/* possible calls to other functions in libbasic.a */
...
x_writeNodes(f,(char*)"someName.dep");
...
```

The application code would then be compiled and linked to libbasic.a.

The terms *internal interface* and *external interface* make sense here, since an application may not call the functions at all, while it must include those header files.

The available functions are defined in `src/basicop`.

10.4 External interface

Besides the four macro operations (utilities) mentioned above, HPFfe provides a minimum functional interface for compiler writers to build their own compilation modules.

10.4.1 How to read the output of `dumpdep`

Before discussing the functions in `libbasic.a`, let's first see how to read the output of dumpdep utility. Taking the following program as an example,

```
program test
integer a,b
```

```
   a = 0; b = 0
   if (a.eq.0) then
      a = a + 1
   else
      b = b - 1
   end if
   call sub(a)
   end

   subroutine sub(n)
   n = n + 1
   return
   end
```

After processing by hpf2dep and hpfsc, we end up with a test.dep file.

```
xyz% dumpdep test.dep
```

will generate the following to stdout.

```
Source is "test.f"

 #blob  #bfnd  #llnd  #symb      #type  #labl  GblBf #deps  #cmnt  #file
-------------------------------------------------------------------------
   16     16     27      9         11     0      1      0      0      1

***BIF NODES***

  id  var  cp   bp1  cmnt  sym   lp1    lp2    lp3   dp1 dp2  lab  lbl glin llin file  thread
 1-B 100  ----  --  ----  ----   ----   ----   ----  --   --   ----  --   0  0   1-F   2-B       GLOBAL
        L:    2-B    13-B    --
        R:   --
        decl_specs(bits set)=
 2-B 101   1-B  13  ----   3-S   ----   ----   ----  --  --   ----  --   1  1   1-F   3-B    PROG_HEDR
        L:    3-B     4-B     5-B     6-B    11-B    12-B    --
        R:   --
        decl_specs(bits set)=
 3-B 154   2-B  --  ----  ----    3-E    1-E   ----  --  --   ----  --   2  2   1-F   4-B     VAR_DECL
        L:   --
        R:   --
        decl_specs(bits set)=
 4-B 111   2-B  --  ----  ----    6-E    7-E   ----  --  --   ----  --   3  3   1-F   5-B  ASSIGN_STAT
        L:   --
        R:   --
        decl_specs(bits set)=
 5-B 111   2-B  --  ----  ----    8-E    9-E   ----  --  --   ----  --   3  3   1-F   6-B  ASSIGN_STAT
        L:   --
        R:   --
        decl_specs(bits set)=
 6-B 145   2-B  --  ----  ----   12-E   ----   ----  --  --   ----  --   4  4   1-F   7-B   LOGIF_NODE
        L:    7-B     8-B    --
        R:    9-B    10-B    --
        decl_specs(bits set)=
 7-B 111   6-B  --  ----  ----   13-E   16-E   ----  --  --   ----  --   5  5   1-F   8-B  ASSIGN_STAT
        L:   --
        R:   --
        decl_specs(bits set)=
```

```
 8-B 104   6-B  --  ----  ----   ----   ----   ----  --  --   ----  --   5  5   1-F   9-B  CONTROL_END
        L:   --
        R:   --
        decl_specs(bits set)=
 9-B 111   6-B  --  ----  ----   17-E   20-E   ----  --  --   ----  --   7  7   1-F  10-B  ASSIGN_STAT
        L:   --
        R:   --
        decl_specs(bits set)=
10-B 104   6-B  --  ----  ----   ----   ----   ----  --  --   ----  --   8  8   1-F  11-B  CONTROL_END
        L:   --
        R:   --
        decl_specs(bits set)=
11-B 113   2-B  --  ----  ----   23-E   ----   ----  --  --   ----  --   9  9   1-F  12-B    PROC_STAT
        L:   --
        R:   --
        decl_specs(bits set)=
12-B 104   2-B  --  ----  ----   ----   ----   ----  --  --   ----  --  10 10   1-F  13-B  CONTROL_END
        L:   --
        R:   --
        decl_specs(bits set)=
13-B 102   1-B  --  ----   8-S   ----   ----   ----  --  --   ----  --  12  1   1-F  14-B    PROC_HEDR
        L:   14-B    15-B    16-B    --
        R:   --
        decl_specs(bits set)=
14-B 111  13-B  --  ----  ----   24-E   27-E   ----  --  --   ----  --  13  2   1-F  15-B  ASSIGN_STAT
        L:   --
        R:   --
        decl_specs(bits set)=
15-B 136  13-B  --  ----  ----   ----   ----   ----  --  --   ----  --  14  3   1-F  16-B  RETURN_STAT
        L:   --
        R:   --
        decl_specs(bits set)=
16-B 104  13-B  --  ----  ----   ----   ----   ----  --  --   ----  --  15  4   1-F  ----  CONTROL_END
        L:   --
        R:   --
        decl_specs(bits set)=

***LOW-LEVEL NODES***

    id    var    type    symb    ll1     ll2                    tag...
    1-E   479     1-T    ----    ----    ----                      TYPE_OP
    2-E   307    ----     4-S    ----    ----                      VAR_REF
    3-E   312     9-T    ----     2-E     5-E                     EXPR_LIST
    4-E   307    ----     5-S    ----    ----                      VAR_REF
    5-E   312     9-T    ----     4-E    ----                     EXPR_LIST
    6-E   307     1-T     4-S    ----    ----                      VAR_REF
    7-E   300     1-T                       0                     INT_VAL
    8-E   307     1-T     5-S    ----    ----                      VAR_REF
    9-E   300     1-T                       0                     INT_VAL
   10-E   307     1-T     4-S    ----    ----                      VAR_REF
   11-E   300     1-T                       0                     INT_VAL
   12-E   328     6-T    ----    10-E    11-E                        EQ_OP
   13-E   307     1-T     4-S    ----    ----                      VAR_REF
   14-E   307     1-T     4-S    ----    ----                      VAR_REF
   15-E   300     1-T                       1                     INT_VAL
   16-E   334     1-T    ----    14-E    15-E                       ADD_OP
   17-E   307     1-T     5-S    ----    ----                      VAR_REF
   18-E   307     1-T     5-S    ----    ----                      VAR_REF
   19-E   300     1-T                       1                     INT_VAL
   20-E   350     1-T    ----    18-E    19-E                     MINUS_OP
```

```
   21-E   307     1-T     4-S    ----    ----                      VAR_REF
   22-E   312     9-T    ----    21-E    ----                     EXPR_LIST
   23-E   370    ----     8-S    22-E    ----                     PROC_CALL
   24-E   307     1-T     9-S    ----    ----                      VAR_REF
   25-E   307     1-T     9-S    ----    ----                      VAR_REF
   26-E   300     1-T                       1                      INT_VAL
   27-E   334     1-T    ----    25-E    26-E                       ADD_OP

***SYMBOL NODES***

 id  var   type  attr     next   scope  variable-name
1-S  550   ----    0      2-S    1-B         *                                  DEFAULT
2-S  505   ----    0      3-S    1-B    PROG_NAME symb_lst= ---- prog_hedr = 2-B PROGRAM_NAME
3-S  505   ----    0      7-S    1-B       test   symb_lst= ---- prog_hedr = 2-B PROGRAM_NAME
4-S  503    1-T 536870912 5-S    2-B         a    local= 600 nxt_in= ---- nxt_out= ---- VARIABLE_NAME
5-S  503    1-T 536870912 6-S    2-B         b    local= 600 nxt_in= ---- nxt_out= ---- VARIABLE_NAME
6-S  600    2-T    0      ----   ----       sub                                 LOCAL
7-S  505   ----    0      8-S    1-B    PROG_NAME symb_lst= ---- prog_hedr= 13-B PROGRAM_NAME
8-S  506   ----    0      ----   1-B        sub   n_in=    1 n_out=    0 n_io=    0
                                        in_l= 9-S out_l= ---- sy_l= 9-S p_hdr= 13-B    PROCEDURE_NAME
9-S  503    1-T 536870912   ----   13-B      n    local= 603 nxt_in= ---- nxt_out= ---- VARIABLE_NAME

***TYPE NODES***

   id   var   name length
   1-T  551   ----   --                                    T_INT
   2-T  552   ----   --                                    T_FLOAT
   3-T  553   ----   --                                    T_DOUBLE
   4-T  554   ----   --                                    T_CHAR
   5-T  556   ----   --                                    T_STRING
   6-T  555   ----   --                                    T_BOOL
   7-T  564   ----   --                                    T_COMPLEX
   8-T  583   ----   --                                    T_DCOMPLEX
   9-T  550   ----   --                                    DEFAULT
  10-T  584   ----   --                                    T_PROCESSORS
  11-T  585   ----   --                                    T_TEMPLATE

  bits set:   0 syn/protected,  1 shared/public,  2 private,  3 future,  4 virtual,
              5 inline,  6 unsigned,  7 signed,  8 short,  9 long,  10 volatile,
              11 const,  12 typedef,  13 extern,  14 friend,  15 static,  16 register,
              17 auto,  18 global,  19 Sync,  20 atomic,  21 __private, 22 restrict

***LABEL NODES***

***COMMENT NODES***

***FILENAME NODES***

    1-F test.f
```

The first line indicates the name of corresponding source file name, followed by a statistics of various data objects used by the program. The "L" and "R" lines after each bif node indicate its left and right control children (in a chain of blob nodes), respectively.

10.4.2 The interface functions

This interface exists as `libbasic.a` in hpfFrontEnd-x.xx/lib. It contains the following functions:

Table 10.1: Columns in the BIF NODES section

id	the sequence number of the node
var	the class (tag,variant) of the node
cp	its control parent
bp1	
cmnt	possible comment/annotation associated with the statement
sym	user provided symbol associated with some statement
lp1	the first low level node associated with it
lp2	the second low level node associated with it
lp3	the third low level node associated with it
dp1	(data dependency test related)
dp2	(data dependency test related)
lab	statement label
lb1	label involved in the statement, such as goto
glin	global line number in the file
llin	local line number in the program unit
file	file it belongs to
thread	its next bif node in allocation sequence

- PTR_SYMB x_lookupSymb(PTR_FILE file, PTR_BFND scope, char* name);

  Look up the *name* in the *scope*, returns a pointer to the entry if found; SMNULL if not found.

  Strickly speaking, even if the *name* is found in *scope* but no *tag* conflict, we should also be able to create a new symbol entry. Among the 16 possible tags of a symbol, CONST_NAME(500) and VARIABLE_NAME(503) will be considered as conflicting tags, namely, if there is a *name* with tag=500 then we can not have the same name with tag=503. But it should be OK if the same name is used both for VARIABLE_NAME and PROCEDURE_NAME. For simplicity, we'll leave this capability out.

- PTR_SYMB x_makeSymb(PTR_FILE file, int tag, char* name);

  Create a new symbol node of *name* with *tag*. Returns a pointer to the node, or SMNULL if no memory available. Scope information, if necessary, is assumed to be set by application after this function.

- void x_enterSymb(PTR_FILE file, PTR_SYMB new);

  Enter a symbol node to symbol table. Appropriate adjustments to hash table and other internal parameters (such as number of symbols) are performed.

- int x_deleteSymb(PTR_FILE file, PTR_BFND scope, char* name);

Remove the symbol with given *name* in the *scope* of the *file* from symbol table, memory associated with the symbol is deallocated. Return -1, if the symbol is not found.

- PTR_LLND x_makeLlnd(PTR_FILE file, int tag, PTR_SYMB symbol, PTR_LLND left, PTR_LLND right);

  Make an expression node with *tag*, taking its subexpression from *left* and *right*, respectively. Returns a pointer to the expression.

- void x_deleteLlnd(PTR_FILE file, PTR_LLND node);

  Remove the low level *node* together with its children low level nodes from AST. Associated memory is released. Leaves (symbol nodes) in the expression tree are kept intact.

- PTR_BFND x_makeBfnd(PTR_FILE file, int tag);

  Memory is allocated for a bif node, initialize the bif node with *tag*.

- void x_insertBfnd(PTR_FILE file, PTR_BFND node, PTR_BFND next);

  Insert a **control structure** headed by bif *node* before *next*. The control structure will be in the same control branch as *next*. Proper link updates are performed, the resulting structure is as if it came from a source program with one additional construct in designated position.

  Note, a control structure may be a single assignment statement or an entire if-then-else-endif structure.

- void x_deleteBfnd(PTR_FILE file, PTR_BFND node);

  Delete the control structure headed by *node* from AST. The associated low level nodes are also deleted. For simplicity at this point, we do not touch related symbols.

  Thus, if *node* = global_bfnd then the entire AST is deleted.

- PTR_BFND x_setBfndList(PTR_BFND first, PTR_BFND second);

  Returns a pointer to a bif node list composed of the *first* and the *second* lists. Note, the bif node lists are implemented by blob node lists, and always associated with some control structure. This function is useful for adding a sequence of statements into some control branch.

- PTR_LLND x_setLlndList(PTR_LLND first, PTR_LLND second);

  Combine two expression lists. Items in an expression list are linked by their second low level node pointers. Expression lists occur in triplets, subroutine parameters, etc.

- PTR_SYMB x_setSymbList(PTR_SYMB first, PTR_SYMB second);

  Combine two symbol lists. Items in a symbol list are linked by their *id_list* fields. Symbol list may occur in variable declarations, etc.

- int x_hash(char * string);

  Compute the hash value for *string*.

- void x_insert_hash(PTR_SYMB symbol, PTR_HASH hash_tbl[]);

  Create a hash table entry corresponding to *symbol*; insert the entry in *hash_tbl*.

- int x_readNodes(PTR_FILE file); Read an entire dep file into memory. Note: the file must be opened before calling the function, and the file handle must have been put in the *fid* field of *file*.

- int x_writeNodes(PTR_FILE file, char * string);

  Write out an internal representation, pointed by *file*, of a program from memory to a disk file named *string*.

  Note, x_writeNodes and x_readNodes observe the same data format. Thus, they should be used together.

To understand how to make use of them, we need to have some idea on what consists of the intermediate representation (IR) of an HPF program under HPFfe and how the intermediate representation is constructed by HPFfe.

For simplicity, let's ignor the complications involved with separate compilation. We in what follows assume an HPF program always corresponds to a single file, though there may be multi program units in the file.

IR of each program file is represented by a data entity with a structure as:

```
struct file_obj {
   char     *filename;            /* filename of the .dep file */
   FILE     *fid;                 /* its UNIX file id */
   int       lang;                /* type of language */
   PTR_HASH *hash_tbl;            /* hash table for this file obj */
   PTR_BFND  global_bfnd;         /* global BIF node for this file */
   PTR_BFND  head_bfnd,           /* head of BIF node for this file */
             cur_bfnd;            /* the last BIF node in bif node list */
   PTR_LLND  head_llnd,           /* head of low level node */
             cur_llnd;            /* the last low lovel node */
   PTR_SYMB  head_symb,           /* head of symbol node */
             cur_symb;            /* the last entry to symble table
   PTR_TYPE  head_type,           /* head of type node */
             cur_type;            /* the last entry to type table */
   PTR_BLOB  head_blob,           /* head of blob node */
             cur_blob;            /* the last blob node */
   PTR_DEP   head_dep,            /* head of dependence node */
             cur_dep;             /* the last dependence node */
   PTR_LABEL head_lab,            /* head of label node */
             cur_lab;             /* the last label node */
   PTR_CMNT  head_cmnt,           /* head of comment node */
             cur_cmnt;            /* the last comment node */
```

```
    PTR_FNAME head_file;
    int       num_blobs,            /* no. of blob nodes */
              num_bfnds,            /* no. of bif nodes */
              num_llnds,            /* no. of ll nodes */
              num_symbs,            /* no. of symb nodes */
              num_label,            /* no. of label nodes */
              num_types,            /* no. of type nodes */
              num_files,            /* no. of filename nodes */
              num_dep,              /* no. of dependence nodes */
              num_cmnt;             /* no. of comment nodes */
};
```

This data entity captures all handles to various data structures representing the underline program. In what follows, we give a description for each of components in the structure. To facilitate the discussion, we first note a few terms (confusing, but frequently used in HPFfe).

- BIF node or bif node — high level node in abstract syntax tree, which normally represents a statement in a program. The linkage of bif nodes in AST represents the *control structure* of a program. Each bif node contains two pointers which possibly point to a list of bif (LOB, or BLOB) nodes as its children, respectively. We call the first pointer *true* or *left* branch, and the second *false* or *right* branch.

  The right branch is null for most statements.

- Low level node or ll node — low level node in abstract syntax tree, which normally represents a expression in a statement. Each bif node contain three pointers to ll nodes.

- BLOB node or blob node — as indicated above, they serve to form a list of bif nodes corresponding a branch of control flow. In other words, a bif node accesses its control children via blob nodes.

- filename — points to the character string of a .dep file to be processed.

- fid — the UNIX file handle of the .dep file

- lang — a code designating the language. We only deal with HPF.

- hash_tbl — hash table is organized as a one dimensional array of pointers which point to a list of hash entries, respectively. This "hash_tbl" is a pointer of pointer, which points to the first element of the array.

- global_bfnd — there is a global BIF node designed for each file. It's left branch corresponds to the list of program units in the file.

- head_bfnd — points to the first bif node of the program, it's the same as global_bfnd in value, but emphasizes different aspect of the intermediate representation, namely,

all bif nodes of a program are also linked as a list, according to lexical order, via the component *thread* in bif node data structure. This way, one may talk about some thing like "the 100th bif node of the program". *id* in bif node data structure indicates the position of a bif node in the list.

cur_bfnd — the last BIF node in the bif node list linked by *thread*.

- head_llnd — similarly, all low level nodes are also linked as a list, headed by head_llnd.

  cur_llnd — the last low level node in the low level node list linked by *thread*.

- head_symb — all symbols are also linked as a list, via *thread* component, headed by head_symb.

  cur_symb — the last entry to symble table (the list linked by *thread*).

- head_type — all type nodes are linked as a list headed by head_type.

  cur_type — the last entry to type table

- head_blob — it looks like all blob node are also linked as a **global** list, though does not make much sense to me.

  cur_blob — the last blob node

- head_dep — this is some thing related to data dependence analysis, we don't address it in this document.

  cur_dep — the last dependence node.

- head_lab — head of label node

  cur_lab — the last label node in the list linked by *next*.

  Label nodes records statement lables in a scope. We include its structure here for easy reference.

```
struct Label {
   int          id;             /* identification tag */
   PTR_BFND     scope;          /* level at which ident is declared */
   PTR_BLOB     ud_chain;       /* use-definition chain */
   unsigned     labused :1;     /* if it's been referenced */
   unsigned     labinacc:1;     /* illegal use of this label */
   unsigned     labdefined:1;   /* if this label been defined */
   unsigned     labtype:2;      /* UNKNOWN, EXEC, FORMAT, and OTHER */
   long         stateno;        /* statement label */
   PTR_LABEL    next;           /* point to next label entry */
   PTR_BFND     statbody;       /* point to body of statement */
   PTR_SYMB     label_name;     /* label name for VPC++       */
};
```

- head_cmnt — head of comment node

  cur_cmnt — the last comment node in the list linked by *thread*, or *next* ?

```
struct cmnt {
   int id;
   int type;
   int counter;                    /* New Added for VPC++ */
   char* string;
   struct cmnt *next;
   struct cmnt *thread;
};
```

- head_file — It seems no use for now.
- the numbers of various nodes mentioned above are recorded in num_blobs, num_bfnds, num_llnds, num_symbs, num_label, num_types, num_files, num_dep, and num_cmnt, respectively.

10.5 Internal interface

By *internal interface*, we mean the internal data structures employed by HPFfe. These data structures are defined by the header files in hpfFrontEnd-x.xx/include.

The related header files are:

- typedef.h — type definitions for all major data structure pointers. Especially, the struct file_obj is defined here.
- bif.h — define the data structure for each HPF statement. The structures are "unioned" in a bif node.

  Figure 10.1 shows primary information in a bif node and basic control structure in a program represented in HPFfe. Note, control dependence relations among bif nodes are implemented via blob nodes.
- defnodes.h — some misc. constant definitions.
- ll.h — define the structures for various expressions in an HPF program.

  Figure 10.2 shows primary information in a low level node and an example how a typical expression is represented in HPFfe.
- tags.h — assign a distinct integer (*tag or variant*) for different types of elements in a language. This integer is of primary importance in various nodes.
- dep.h — a minor, define two structures for data dependence related matter.

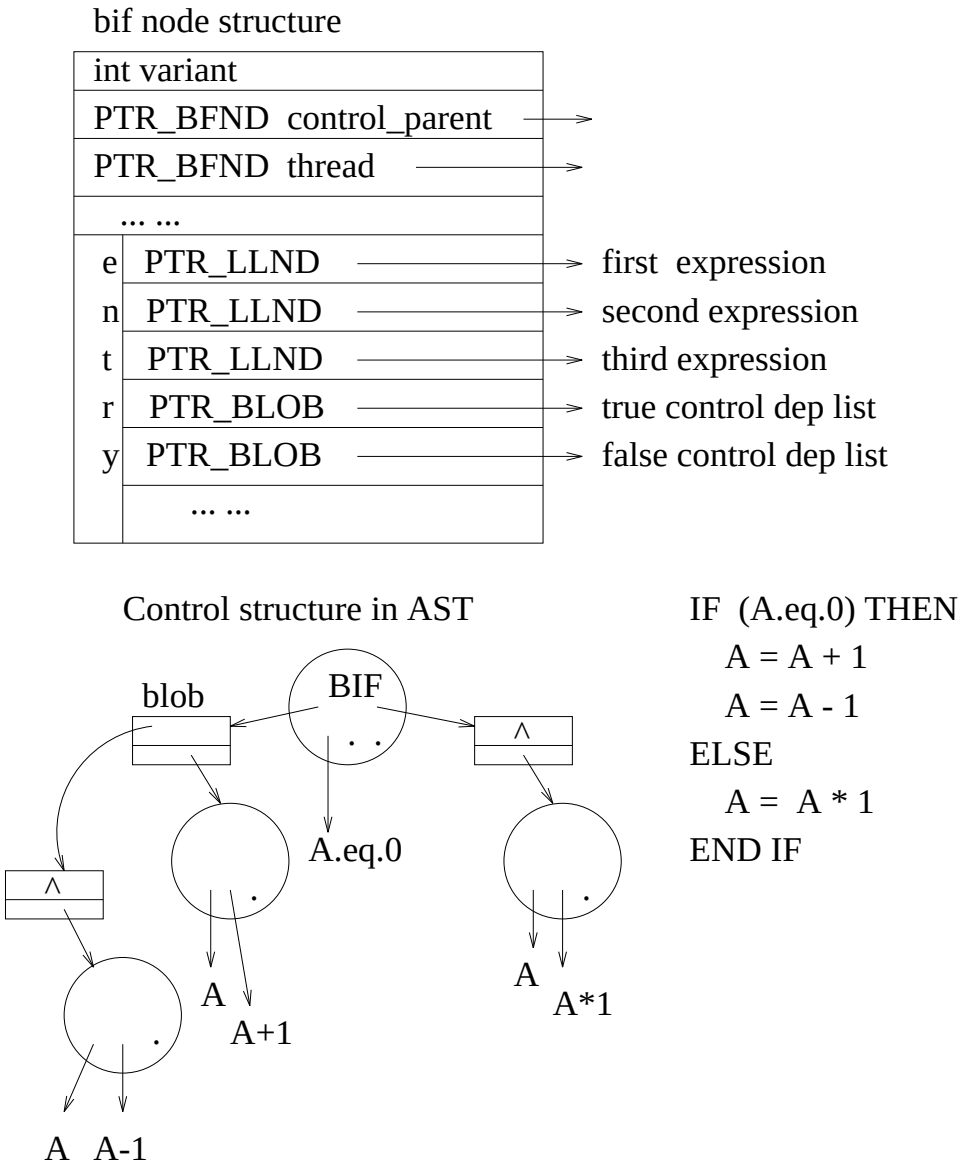


Figure 10.1: Basic control structure in AST of HPFfe

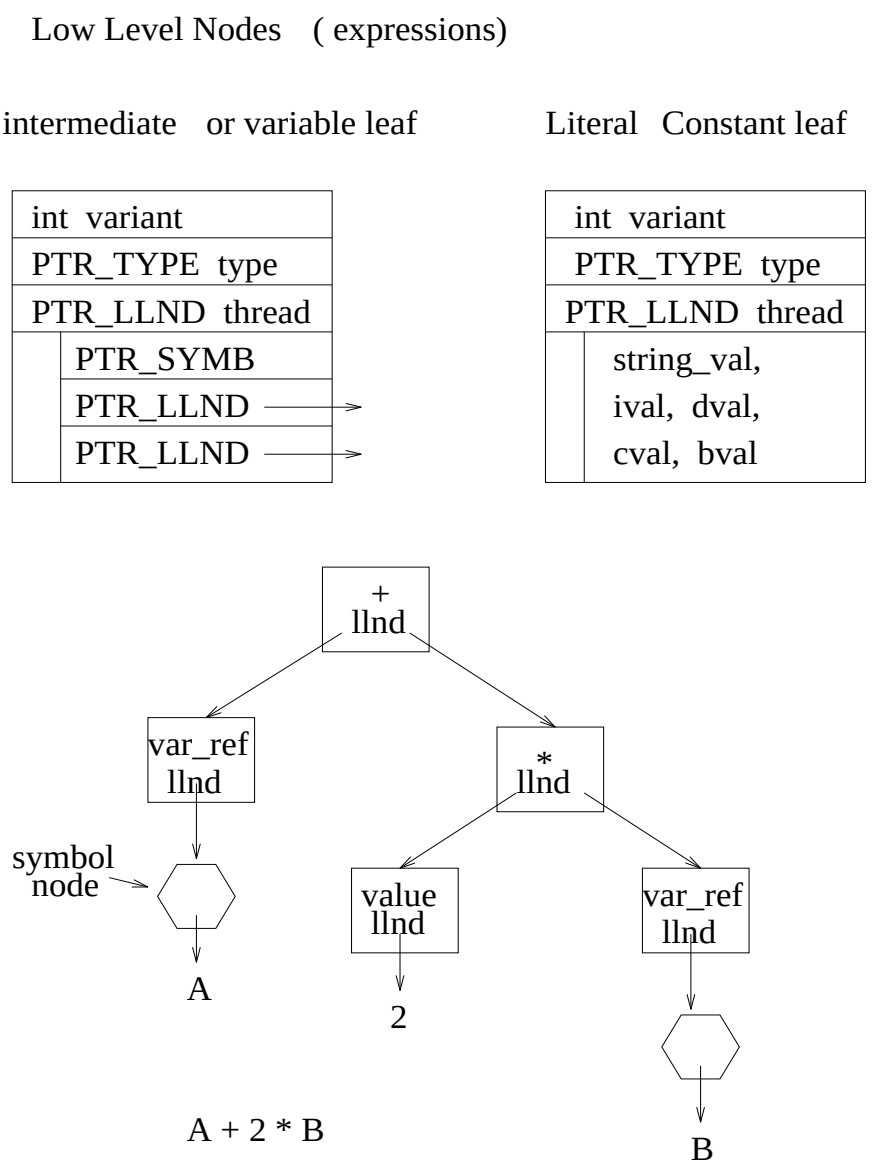


Figure 10.2: Low level node structure in AST of HPFfe

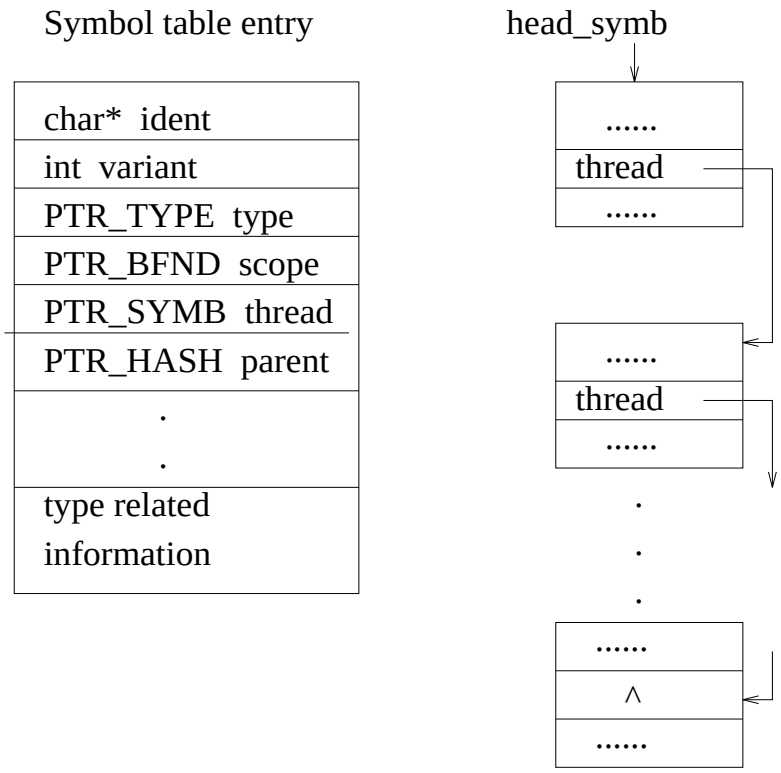


Figure 10.3: Symbol node structure in HPFfe

- decl.h — this is a list of functions used in HPFfe.
- dep_struct.h — defines the structure of .dep files generated by HPFfe.
- symb.h — symbol table and hash table entry structures are defined here.

  Figure 10.3 shows primary information in a symbol node and how they are linked as a linear list.

  To facilitate operations on the symbol table, a hashing mechanism is also created and associated with symbol table. Figure 10.4 shows information in a hash entry and how a hash table is organized in HPFfe.
- makenodes.h — declares functions in libbasic.a and necessary extern variable to use the functions.

The most important data structures are the abstract syntax tree and symbol table.

The AST is composed of two types of nodes, *bif* nodes and *ll* nodes. Each statement in the program is represented by a bif node, the expressions under a statement are represented by ll nodes.

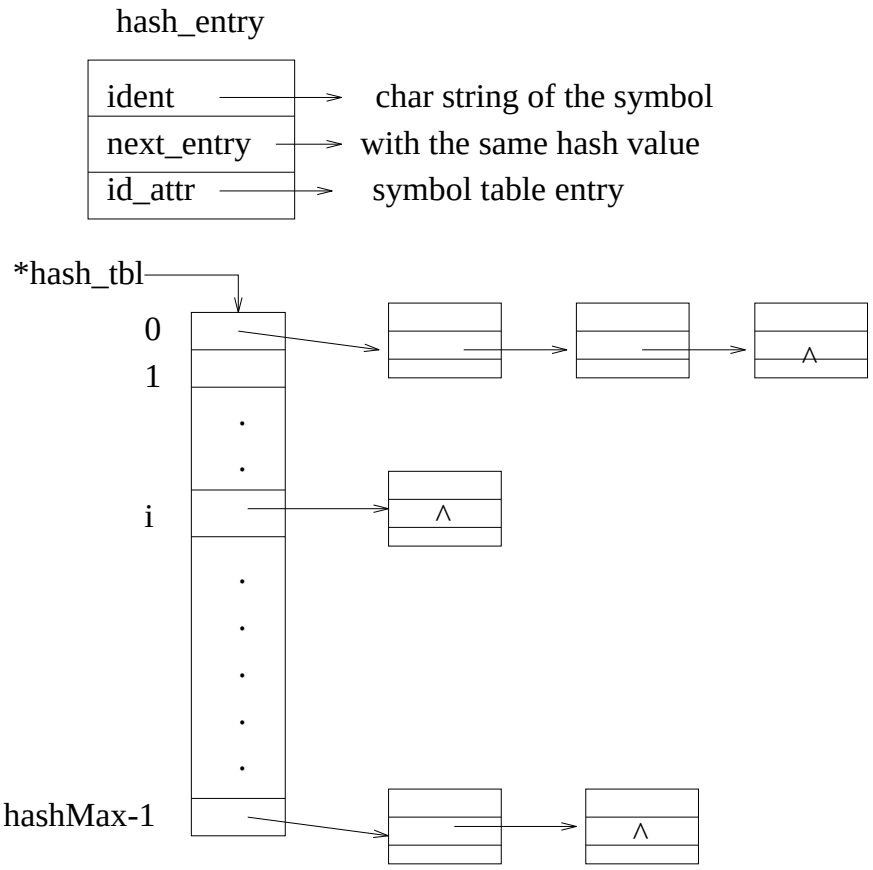


Figure 10.4: Hash table in HPFfe

10.6 A sample session

We present two example applications that use HPFfe. The first one makes use of the class library `xsageop`; the other calls `basicop` directly.

10.6.1 A sample of using `xsageop`

This program performs indentation of programs.

```
#include <stdio.h>
#include "compilerApi.h"

enum FLAG {off, on};

FLAG in_main_program;
FLAG in_procedure;

int tabnumber=0;

void puttab(int number) {
  for(int i=0; i<number; i++)
    printf("  ");
}

void action(SgStatement * s) {
  puttab(tabnumber);
  printf("id:%4d   variant:%4d\n", s->id(), s->variant());
}

void transform(SgStatement * s) {
  switch(s->variant()) {
  case GLOBAL: {
      int org_numberOfChildren=s->numberOfChildrenList1();
      SgStatement * * children = new SgStatement * [org_numberOfChildren];

      for (int i=0; i<org_numberOfChildren; i++)
         children[i]=s->childList1(i);

      for (i=0; i<org_numberOfChildren; i++)
         transform(children[i]);

      delete [] children;
      break;
    };

  case PROG_HEDR: {
      int org_numberOfChildren=s->numberOfChildrenList1();
      SgStatement * * children = new SgStatement * [org_numberOfChildren];
```

```
    for (int i=0; i<org_numberOfChildren; i++)
       children[i]=s->childList1(i);

    in_main_program = on;

    action(s);

    tabnumber++;

    for (i=0; i<org_numberOfChildren; i++)
       transform(children[i]);

    tabnumber--;

    in_main_program=off;

    delete [] children;

    break;
  }

case PROC_HEDR: {
    SgProcHedrStmt * prochedr = (SgProcHedrStmt *)s;

    int org_numberOfChildren=s->numberOfChildrenList1();
    SgStatement * * children = new SgStatement * [org_numberOfChildren];
    for (int i=0; i<org_numberOfChildren; i++)
       children[i]=s->childList1(i);

    in_procedure=on;

    action(s);

    tabnumber++;

    for (i=0; i<org_numberOfChildren; i++)
       transform(children[i]);

    tabnumber--;

    in_procedure = off;

    delete [] children;

    break;
  };
```

```
  case VAR_DECL:
  case ASSIGN_STAT:
    action(s);
    break;

  case FOR_NODE: {

      int org_numberOfChildren=s->numberOfChildrenList1();
      SgStatement * * children = new SgStatement * [org_numberOfChildren];

      for (int i=0; i<org_numberOfChildren; i++)
         children[i]=s->childList1(i);

      action(s);

      tabnumber++;

      for (i=0; i<org_numberOfChildren; i++)
         transform(children[i]);

      tabnumber--;

      delete [] children;

      break;
    }

  case CONTROL_END:
  case ENDDO_STMT:
    break;

  default:
    printf("Not touched statement in transform phase on line %d\n", s->lineNumber());
    break;
  };
};

int main(int argc, char **argv){

  SgProject * project = new SgProject("test", argc, argv);

  for (int i=0; i< project->numberOfFiles(); i++) {

    SgFile * file = &(project->file(i));

    SgStatement * s = file->firstStatement();

    transform(s);
```

```
  };

  // save a copy of the input file, if you like.
  // f->saveDepFile("debug.dep");

  return 0;
}
```

Compile and linking

```
%xlC -o sample sample.C -I../include ../lib/libxsage.a ../lib/libbasic.a
```

Run

```
%sample test.dep
```

10.6.2 A sample of using `basicop`

The following application adds the statement

```
   print *, 'Enter a parallel structure'
```

before each DO statement.

The key here is to make use of x_readNodes() and x_writeNodes() for input and output, and requires the output is a valid dep file recognizable by dumpdep and unparse.

The code

```
#include "hpffe.h"

main(int argc, char* argv[]) {
   FILE *depFile;
   PTR_FILE f = (PTR_FILE) calloc(1, sizeof(struct file_obj));
   PTR_BFND temp,temp1,temp2,parent,new;
   PTR_LLND str,format,expr_list;
   PTR_BLOB foo,trace;

   depFile = fopen(argv[1], "r");
   f->fid = depFile;

   x_readNodes(f);

/*
Starting from head of bif node list, locate the next DO statement
*/

   for (temp1 = f->head_bfnd; temp1 != BFNULL; temp1 = temp1->thread) {
```

```
      if (temp1->variant == FOR_NODE) {

         format = x_makeLlnd(f,KIND_VAL,LLNULL,LLNULL,SMNULL);
         format->entry.var_ref.symbol = x_makeSymb(f,temp1->control_parent,DEFAULT,"*",0);

         expr_list = x_makeLlnd(f,EXPR_LIST,LLNULL,LLNULL,SMNULL);
         expr_list->entry.list.item = x_makeLlnd(f,STRING_VAL,LLNULL,LLNULL,SMNULL);
         expr_list->entry.list.item->entry.string_val = (char*)"'Enter a Do loop'";

         new = x_makeBfnd(f,PRINT_STAT);
         new->entry.print_stat.format = format;
         new->entry.print_stat.expr_list = expr_list;

         temp = temp1->control_parent;   /* find proceeding bif node in the control branch */
         trace = temp->entry.Template.bl_ptr1;
         for (trace = temp->entry.Template.bl_ptr1;
              trace->next->ref->variant != FOR_NODE;
              trace=trace->next);
         temp = trace->ref;

         foo = x_make_blob(f,new,trace->next); /* make and link the blob for the new bif */
         trace->next = foo;

         new->id = temp1->id;   /* insert the new bif in the list */
         new->thread = temp1;
         temp->thread = new;

         for (temp2 = temp1; temp2->thread != BFNULL; temp2 = temp2->thread) {
            temp2->id = temp2->id + 1;
         }  /* update ids */

      }

   }
   x_writeNodes(f,(char*)"test.dep");
}
```

Compiling and linking

```
%cc -o sample sample.c -I../include ../lib/libbasic.a
```

Run

```
%sample test.dep
```

Chapter 11

Issues in dealing with multidimensional arrays

So far, what we have discussed is focused on one dimensional arrays, and we should have got pretty good idea on how to process them, using a combination of runtime and compiler techniques.

Multidimensional arrays introduce some distinct problems in a compilation system design.

11.1 Dimension match requirement

The first issue resulting from multidimensional arrays is that we may observe an arbitrary permutation between dimensions of two arrays involved in an operation. This permutation directly affects the communication requirement between the two arrays. Let's see some examples.

1. A 'nice' case.

```
INTEGER X(16,16),Y(16,16)
PROCESSORS P(2,2)
ALIGN X(i,j) WITH T(2*i+1,2*j-1)
ALIGN Y(i,j) WITH T(i+1,3*j+1)
DISTRIBUTE T(BLOCK,CYCLIC) ONTO P
...
X = Y
...
```

We may detect communication requirement of the assignment by calling,

```
status_1 = detect_comm(dim 1 of X against dim 1 of Y)
status_2 = detect_comm(dim 2 of X against dim 2 of Y)
```

Then 9 combinations of cases may result for *(status_1,status_2)*:

```
no comm, no comm
no comm, shift
no comm, remap
shift, no comm
shift, shift
shift, remap
remap, no comm
remap, shift
remap, remap
```

If we consider one 'remap' implies a general remap for the whole array, then compiler would prepare to generate node programs for one of the 5 cases:

```
no comm, no comm
no comm, shift
shift, no comm
shift, shift
remap
```

What we have covered in previous chapters should be enough for generating the node programs.

2. Now, if we change the alignment directive a little bit, as:

```
INTEGER X(16,16),Y(16,16)
PROCESSORS P(2,2)
ALIGN X(i,j) WITH T(2*i+1,2*j-1)
ALIGN Y(i,j) WITH T(3*j+1,i+1)       !!!
DISTRIBUTE T(BLOCK,CYCLIC) ONTO P
...
X = Y
...
```

the situation becomes quite different. There is a non identity permutation between dimensions of X and Y. What's the communication requirement then ? Our detect_comm() algorithm can not apply to this situation. Why ?

A possible treatment is 'remap', once compiler finds out the dimension-permuted situation. 'remap' would then essentially perform some kind of transpose function.

Although some careful analysis is possible that may avoid callings remap, (for instance, if template T happens to be very big so that X and Y are actually held by P(1,1) entirely), we think the analysis may not be worthy to be included in compiler – let runtime (remap) do the job.

To facilitate our discussion, let's use `DMR` for *dimension match requirement*. Each assignment statement induces some correspondence between dimensions of *lhs* and *rhs* of the statement. For instance,

```
X(:) = Y(:)       1 --> 1
X(3,:) = Y(:)     2 --> 1
X(:,3) = Y(4,:)   1 --> 2
FORALL (i=1:100,j=1:100) X(i,j) = Y(j,i)    1 --> 2; 2 --> 1
```

We say a DMR is met, if for all pairs of corresponding dimensions,

- the two dimensions are distributed on the same processor dimension (not template dimension), or
- they are both ultimately (array or template) collapsed

The DMR of this example is not met.

3. Effect of `FORALL` statement.

```
INTEGER X(16,16),Y(16,16)
PROCESSORS P(2,2)
ALIGN X(i,j) WITH T(2*i+1,2*j-1)
ALIGN Y(i,j) WITH T(3*j+1,i+1)
DISTRIBUTE T(BLOCK,CYCLIC) ONTO P
...
FORALL (i=1:16,j=1:16) X(i,j) = Y(j,i)
...
```

compiler should realize the DMR is met. Our detect_comm() then can be employed as

```
status_1 = detect_comm(dim 1 of X against dim 2 of Y)
status_2 = detect_comm(dim 2 of X against dim 1 of Y)
```

and then code will be generated for one of the 5 possible cases. We see alignment directive and `FORALL` statement may affect DMR.

4. Collapsing.

```
INTEGER X(16,16),Y(16,16)
PROCESSORS P(4)
ALIGN X(i,j) WITH T1(2*i+1,2*j-1)
ALIGN Y(i,j) WITH T2(3*j+1,i+1)
DISTRIBUTE T1(BLOCK,*) ONTO P
DISTRIBUTE T2(BLOCK,*) ONTO P
...
FORALL (i=1:16,j=1:16) X(i,j) = Y(j,i)
...
```

we see `DMR: 1 --> 2; 2 --> 1`. Both dim_1(X) and dim_2(Y) are distributed on the same processor dim; both dim_2(X) and dim_1(Y) are collapsed. We can do

```
status_1 = detect_comm(dim 1 of X against dim 2 of Y)
```

and then generate code for 3 possible cases.

5. Not all collapsings make compiler easy.

```
INTEGER X(16,16),Y(16,16)
PROCESSORS P(4)
ALIGN X(i,j) WITH T1(2*i+1,2*j-1)
ALIGN Y(i,j) WITH T2(3*j+1,i+1)
DISTRIBUTE T1(BLOCK,*) ONTO P
DISTRIBUTE T2(*,BLOCK) ONTO P
...
FORALL (i=1:16,j=1:16) X(i,j) = Y(j,i)
...
```

DMR: 1 $\longrightarrow$ 2; 2 $\longrightarrow$ 1. dim_1(X) is distributed, dim_2(Y) is collapsed; dim_2(X) is collapsed, dim_1(Y) is distributed;

The DMR is not met. Whether a DMR is met or not can be detected by compiler, and we see the notion of DMR suggests a strategy: if not met, go for `remap`; otherwise, call `detect_comm()`.

11.2 Rank-reduced sectioning

Before we see some examples, let's introduce a notion of 'stridden' or 'amplified' data distribution. (This allows us to avoid a discussion of general `BLOCK(n)/CYCLIC(n)` distributions.)

We say a dimension of template elements is `BLOCK/CYCLIC` distributed on processors with *stride* k, if two consecutive template elements assigned to the same processor are k locations apart in local memory. We use `BLOCK_k/CYCLIC_k` to denote the situation. `BLOCK_1` and `CYCLIC_1` are equivalent to normal `BLOCK` and `CYCLIC`, respectively. Note, we are not trying to modify the language. This notion is only for convenience when we talk about the effect of rank reduced sectioning. As an example, for

```
INTEGER X(16)
PROCESSORS P(4)
DISTRIBUTE X(BLOCK_3) ONTO P
```

we would see the following picture in local memory:

```
x--x--x--x--          (for every processor, in this example)
```

The extent of local memory is to be 12, instead of 4. Another example,

```
INTEGER X(16)
PROCESSORS P(4)
TEMPLATE T(40)
ALIGN X(i) WITH T(2*i+5)
DISTRIBUTE T(BLOCK_3) ONTO P
```

The situation on template T looks like,

```
T: ------x-x-|x-x-x-x-x-|x-x-x-x-x-|x-x-x-x---
```

Then local extent would be 30 (instead of 10), and we see the local memory profile:

```
P(1): ------------------x-----x-----
P(2): x-----x-----x-----x-----x-----
P(3): x-----x-----x-----x-----x-----
P(4): x-----x-----x-----x-----------
```

Note, this situation can be equivalently formulated as

```
TEMPLATE T(120)
ALIGN X(i) WITH T(6*i+13)
DISTRIBUTE T(BLOCK) ONTO P
```

We always have the equivalence for `BLOCK` type distribution, the rule is:

```
t_extent(BLOCK) = t_extent(BLOCK_k)*k;
a_stride(BLOCK) = a_stride(BLOCK_k)*k;
a_offset(BLOCK) = a_offset(BLOCK_k)*k - (k-1);
```

where `a_stride` and `a_offset` are stride and offset specified in `ALIGN` directive, they are different from `t_stride` and `t_offset` in a DAD.
An example of `CYCLIC` distribution.

```
INTEGER X(16)
PROCESSORS P(4)
TEMPLATE T(40)
ALIGN X(i) WITH T(2*i+5)
DISTRIBUTE T(CYCLIC_3) ONTO P

T: ------x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x---
   1234123412341234123412341234123412341234
```

The local extent is still 30, and we see the local memory profile:

```
P(1): ------x--x--x--x--x--x--x--x--
P(2): ----------------------------
P(3): ---x--x--x--x--x--x--x--x-----
P(4): ----------------------------
```

Note, this situation can *not* be equivalently formulated as some `CYCLIC` distribution.

Anyway, we need this notion to deal with rank reduced sections. To this end, we use -k for `BLOCK_k`, k for `CYCLIC_k`, and keep 0 for `*` for *dist_code* in DAD.

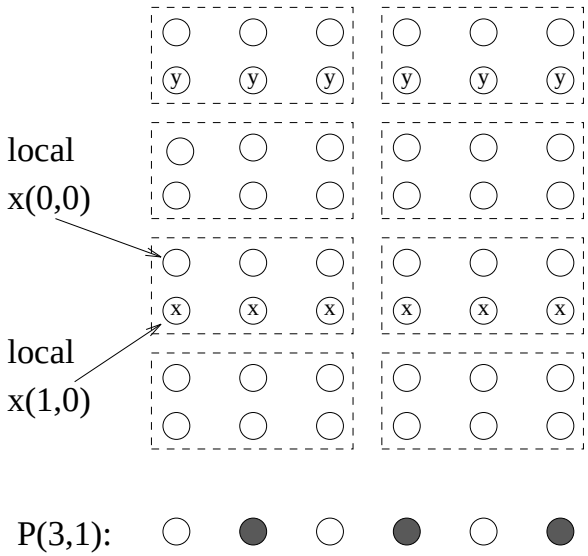


Figure 11.1: Effect of rank reduced sectioning

11.3 Subgrid of a processor grid

Meeting DMR is not sufficient for reaching a valid conclusion for communication requirement. The assignment,

```
X(2,:) = Y(6,:)
```

has DMR $2 \longrightarrow 2$. If DMR is met, is there any special problem in communication detection ? See,

```
   INTEGER X(8,6),Y(8,6)
(6)PROCESSORS P(4,2)
   ALIGN X(i,j) WITH T(i,j)
   ALIGN Y(i,j) WITH T(i,j)
   DISTRIBUTE T(BLOCK,BLOCK) ONTO P
   ...
   X(6,:) = Y(2,:)
   ...
```

The data distribution is depicted as in Figure 11.1.

The DMR is met, if you call our `detect_comm()` for dim_2 of X and dim_2 of Y, it will return 'no comm'. But row 6 of X and row 2 of Y reside on different rows of processor grid, which implies communication. This could be discovered by calling `detect_comm()` with (dim_1 of X,6) and (dim_1 of Y,2), a shift communication requirement should be returned. Nevertheless, a more effective approach may be employed.

With the help of DAD, we would like to treat the result of a *rank reduced sectioning* as some array of lesser rank. That is, we would like to view X(6,:) as some one dimensional

array X' of 6 elements, this X' will be described by dad_X'. Let's see how to adjust dad_X to form dad_X'.

```
dad_X:
    g_extent  8   6
    t_extent  8   6
    t_stride  1   1
    t_offset  0   0
   dist_code -1  -1   we use <1 for BLOCK, 0 for *, >1 for CYCLIC
    on_pdim   1   2
    l_extent  2   3
    l_lb      0   0
    l_ub      1   2
    l_stride  1   1
   ghostsize  0   0
    p_shape   4   2
    my_coord  *   *
 slice_coord -1  -1
              1   2  2  comm  major
              base_address = address of local X(0,0)
```

Pay special attention to *slice_coord*, *rank*, and *base_address* in the following DAD.

```
dad_X(6,:):
    g_extent  6
    t_extent  6
    t_stride  1
    t_offset  0
  dist_code  -2 <- dist_code(2)*l_extent(1)
    on_pdim   2
    l_extent  6 <- l_extent(2)*l_extent(1)
    l_lb      0 <- l_lb(2)*l_extent(1)
    l_ub      4 <- l_ub(2)*l_extent(1)
    l_stride  2 <- l_stride(2)*l_extent(1)
    ghstsize  0
    p_shape   4   2
    my_coord  *   *
 slice_coord  3  -1
              1   1  2  comm  major
              base_address = address of local X(1,0) !!!
```

Figure 11.1 also indicate the result of this slicing. It's not difficult to see it can be seen as the DAD for

```
INTEGER X'(6)
PROCESSORS P(2)
DISTRIBUTE X'(BLOCK_2) ONTO P
```

What we should realize is that X' is indeed equivalent to X(6,:), in terms of elements layout in memory, namely the memory location of X(6,i) is indeed that of $X'(i)$, no matter where it is.

With this concept, we can always view the result of rank reduced sectioning as some array with reduced rank. This concept can be applied to higher dimension, too. For X declared in

```
INTEGER X(16,12,8)
PROCESSORS P(4,2)
DISTRIBUTE X(BLOCK,*,BLOCK) ONTO P
```

we have

```
 dad_X:
 g_extent : 16  12   8
 t_extent : 16  12   8
 t_stride :  1   1   1
 t_offset :  0   0   0
dist_code : -1   0  -1
  on_pdim :  1  -1   2
 l_extent :  4  12   4
     l_lb :  0   0   0
     l_ub :  3  11   3
 l_stride :  1   1   1
ghostSize :  0   0   0
  p_shape :  4   2
 my_coord :  1   1
slice_coord : -1  -1
              1   3   2  comm major
              base_address = address of local X(0,0,0)
```

Then for X(2,:,:), we see a DAD for a two dimensional array.

```
 dad_X'(1:12,1:8) = dad_X(2,:,:)
 g_extent : 12   8
 t_extent : 12   8
 t_stride :  1   1
 t_offset :  0   0
dist_code :  0  -1
  on_pdim : -1   2
 l_extent : 48   4
     l_lb :  0   0
     l_ub : 44   3
 l_stride :  4   1
ghostSize :  0   0
  p_shape :  4   2
 my_coord :  1   1
```

```
slice_coord :  1  -1     note change here
               1   2   2  comm major
               base_address = address of local X(1,0,0)
```

Further,

```
    dad_X''(1:8) = dad_X'(6,:) = dad_X(2,6,:)
    g_extent :   8
    t_extent :   8
    t_stride :   1
    t_offset :   0
   dist_code : -48
     on_pdim :   2
    l_extent : 192
        l_lb :   0
        l_ub : 144
    l_stride :  48
   ghostSize :   0
     p_shape :   4   2
    my_coord :   1   1
 slice_coord :   1  -1    no change on slice coord.
                 1   1   2  comm major
                 base_address = address of local X'(5,0) or X(1,5,0)
```

In general, we have the following rule to update DAD upon *array slicing* or *rank reduced sectioning*:

If the slicing occurs at dimension d with scalar index s,

step 1. find the processor coordinate that holds the index s, set associated slice_coord accordingly. (if dimension d collapsed, no change on slice_coord)

step 2. multiply dist_code(d+1),l_extent(d+1),l_lb(d+1),l_ub(d+1), and l_stride(d+1) by l_extent(d);

step 3 replace dad(1:11,i) by dad(1:11,i+1), i=1,...,rank

step 4. rank = rank - 1

step 5. base_address = local X(0,0,global2local(s),0,0)

How do we know this is right ?

First, we need to ensure node program (runtime) can still correctly traverse local elements. Clearly, the following loop accesses local element correctly,

```
DO i = l_lb(1), l_ub(1), l_stride(1)
  DO j = l_lb(2), l_ub(2), l_stride(2)
    ... X(i,j) ...
  END DO
END DO
```

or

```
DO i = l_lb(1), l_ub(1), l_stride(1)
  DO j = l_lb(2), l_ub(2), l_stride(2)
    ... X(base + i + (j-l_lb(2))*l_extent(1))
  END DO
END DO
```

Second, we should see `global2local()` and `local2global()` index conversions can be performed correctly. For instance,

global2local(i): assume i is a normalized (0-based) global index

```
dist_code=0:
   return l_lb+i*l_stride; (for either collapsing case).

dist_code < 0:
   j = i*t_stride+t_offset,
   j = j mod "block-size" for the processor having the j,
   i = floor(j/t_stride),
   return l_lb+i*l_stride;       ! no need for abs(dist_code) here

   If we make use of abs(dist_code), the algorithm would be
   j = i*t_stride+t_offset,  position on template
   j = j mod "block-size" for the processor having the j,
   return j*abs(dist_code);

dist_code > 0:
   j = i*t_stride+t_offset,
   j = floor(j/p) for the processor having j.
   i = floor(j/t_stride),
   return l_lb+i*l_stride;       ! no need for abs(dist_code) here

   Again, if we make use of abs(dist_code), the algorithm would be
   j = i*t_stride+t_offset,  position on template
   j = floor(j/p) for the processor having j.
   return j*abs(dist_code);
```

11.4 Broadcasting/replication

Multidimensional arrays also require some special attention when doing communication. See,

```
INTEGER X(4,8),Y(4,8)
PROCESSORS P(2,2)
ALIGN Y(i,j) WITH X(i,j)
DISTRIBUTE X(BLOCK,BLOCK) ONTO P
...
```

```
FORALL (i=1:4,j=1:8) X(i,j) = Y(i,3)       (a)
FORALL (i=1:4,j=1:8) X(i,j) = Y(3,j)       (b)
FORALL (i=1:4,j=1:4) X(i,j) = Y(j,3)       (c)
FORALL (i=1:4,j=1:8) X(i,j) = Y(3,i)       (d)
...
END
```

We'll investigate the node program segments for each of the FORALLs,

```
(a) the third column of Y is replicated along second dim of X.
   DO i = 1, 4
     CALL bcast(t,y,dad_y,(/i,3/))
     CALL loop_bounds(dad_x,2,1,8,1,l,u,s)
     DO j = l, u, s
       it = global2local(dad_x,1,i)
       IF (it.NE.-1) X(it,j) = t
     END DO
   END DO
```

The `loop_bounds()` call may be moved outside of the i loop to improve performance. It is possible to devise another runtime function that broadcasts an array (section) in one call.

```
(b) the third row of Y is replicated along first dim of X.
   CALL loop_bounds(dad_x,1,1,4,1,l,u,s)
   DO j = 1, 8
     CALL bcast(t,dad_y,(/3,j/))
     DO i = l, u, s
       it = global2local(dad_x,2,j)
       IF (it.NE.-1) X(i,it) = t
     END DO
   END DO
```

Note the j loop now is the outer loop.

```
(c) the third column of Y is replicated along first dim (first half) of X.
   CALL loop_bounds(dad_x,1,1,4,1,l,u,s)
   DO j = 1, 4
     CALL bcast(t,dad_y,(/j,3/))
     DO i = l, u, s
       it = global2local(dad_x,2,j)
       IF (it.NE.-1) X(i,it) = t
     END DO
   END DO

(d) the third row (first half) of Y is replicated along second dim of X.
   CALL loop_bounds(dad_x,2,1,8,1,l,u,s)
   DO i = 1, 4
     CALL bcast(t,dad_y,(/3,i/))
```

```
  DO j = l, u, s
    it = global2local(dad_x,1,i)
    IF (it.NE.-1) X(it,j) = t
  END DO
END DO
```

As a final example, we see a possible treatment for diagonal access.

```
INTEGER X(8,8),Y(4,8)
PROCESSORS P(2,2)
ALIGN Y(i,j) WITH X(2*i,j)
DISTRIBUTE X(BLOCK,BLOCK) ONTO P
...
FORALL (i=1:8) X(i,i) = Y(3,i)
...
```

we may have

```
DO i = 1, 8
   CALL bcast(t,dad_y,(/3,i/))
   CALL global2local_2(dad_x,i,i,it1,it2)
   IF (it1*it2.GE.0) X(it1,it2) = t
END DO
```

where `global2local_2()` presumably returns two values that constitute an index to a two dimensional array element.

This solution is not very efficient (loop is sequentially executed), but quite general. Clearly, we can have a general global2local() to handle multi dim arrays.

To be more efficient, we may do something like

```
remap Y(3,i) into TEMP, the distribution of which is the same
as first dim of X;
CALL loop_bounds(dad_x,1,1,8,1,l1,u1,s1)
CALL loop_bounds(dad_x,2,1,8,1,l2,u2,s2)
CALL fake_remove(dad_x,1,8,1,1,8,1,l1,u1,s1)
j = l2
DO i = l1,u1,s1
   X(i,j) = TEMP(i)
   j = j + s2
END DO
```

Note we assumed some function `fake_remove()`. This is due to `loop_bounds()` as we defined only works for one dimension structures, and the diagonal can not be expressed as a *linear* composition of two 1-dimensional structures. As a result, four processors all return 'good' l1:u1:s1 and l2:u2:s2. We need to exclude P(1,2) and P(2,1) in this case. Presumably `fake_remove()` will adjust l1:u1:s1 properly to 0:-1:1 for them.

Clearly, this is not only solution. We may well have another runtime function that figures out the bounds directly (something like `global2local_2()` above). A point I am trying to make here is that the translation scheme may generate requirement for runtime functions.

Chapter 12

Compiling irregular problems

Many important computational problems give rise to *irregular data access patterns*. The term *irregular problem* in the context of compiler/runtime technologies usually refers to the application expressed as a program that demonstrates some *irregular data access pattern*.

This chapter introduces the concept and related compilation issues.

12.1 What are irregular problems ?

We see three examples.

1. Algebraic operations involving sparse matrices.

   Suppose we want to compute `Y = AX`, where A is an $n \times n$ sparse matrix, X is a vector of n elements. The following storage scheme may be used for A.

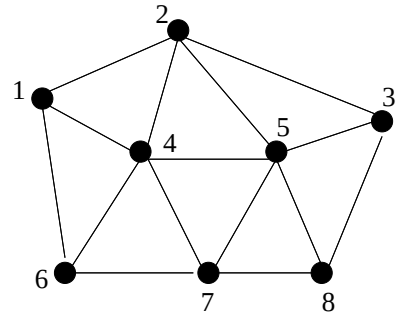
   counter(1:n) — record number of non zero elements in each row.

   index(1:m) — column indices of non zero elements, where m is the total number of non zero elements in A. For sparse matrices, m is normally in the same order as n.

   value(1:m) — the values of non zero elements.

   For instance, for the following sparse matrix A,

$$A = \begin{pmatrix} 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 9 & 4 & 0 & 0 & 0 & 5 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 0 & 2 & 0 & 0 \end{pmatrix}$$



edge1	1	1	1	4	6	4	5	4	2	2	2	3	5	3
edge2	2	4	6	6	7	7	7	5	4	5	3	5	8	8

Figure 12.1: An irregular computation graph

we will have,

counter = (/2,3,1,2,2,1,1,1,1,2/)
index = (/1,4,3,4,8,1,2,3,5,7,9,4,3,6,4,8/)
value = (/1,2,9,4,5,2,1,1,3,8,4,5,4,7,3,2/)

The multiplication procedure would then be

```
l = 1
DO i = 1, n
  u = l + counter(i) - 1
  Y(i) = 0
  DO j = l,u
    Y(i) = Y(i) + value(index(j))*X(index(j))
  END DO
  l = l + counter(i)
END DO
```

The access to *value* via indirection array *index* is known as "irregular".

2. Consider a computational graph of 8 nodes and 15 edges, as shown in Figure 12.1, (taken from reference [18]).

   The graph is represented by the set of its edges, which in program is implemented by two 1-dimensional arrays `edge1()` and `edge2()`. Assume `X(1:8)` and `Y(1:8)` are data arrays associated with nodes, the following program segment is typical,

```
DO j = 1, n_steps
  ...
  DO i = 1, 15
     Y(edge1(i)) = Y(edge1(i)) + f(X(edge1(i),X(edge2(i))
```

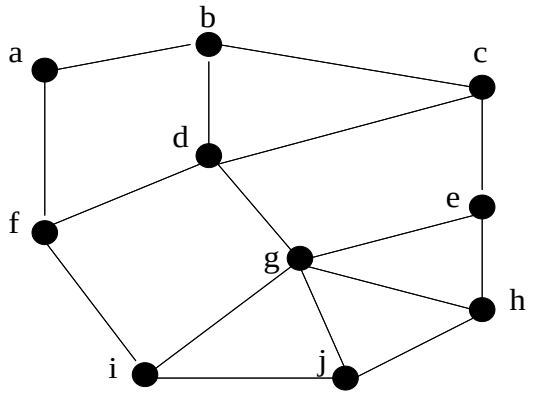


Figure 12.2: An unstructured mesh

```
      Y(edge2(i)) = Y(edge2(i)) + g(X(edge1(i),X(edge2(i))
    END DO
    ...
  END DO
```

where f and g are some application dependent functions. To get a sense of what this segment is doing, take `f(a,b) = b`, and `g(a,b) = a`, the effect of this loop is to set Y of a node to be the sum of it's neighbors' X values — isn't it meaningful ?

We say the loop sweeps over edges.

Once again, data arrays X and Y are accessed/referenced via indirection arrays.

3. Consider an unstructured grid represented by a graph as in Figure 12.2, (taken from reference [15]).

   It may be represented by its adjacent matrix as,

```
   a  b  c  d  e  f  g  h  i  j
a  0  1  0  0  0  1  0  0  0  0
b  1  0  1  1  0  0  0  0  0  0
c  0  1  0  1  1  0  0  0  0  0
d  0  1  1  0  0  1  1  0  0  0
e  0  0  1  0  0  0  1  1  0  0
f  1  0  0  1  0  0  0  0  1  0
g  0  0  0  1  1  0  0  1  1  1
h  0  0  0  0  1  0  1  0  0  1
i  0  0  0  0  0  1  1  0  0  1
j  0  0  0  0  0  0  1  1  1  0
```

   Let X and Y be data arrays associated with nodes. Typical applications compute node-data as a function of the node-data of their neighbors.

The adjacency matrix is sparse. To be efficient both in space and performance of memory hierarchy, we may represent it as

```
numEdges: 2 3 3 4 3 3 5 3 3 3
index1: 1 3 6 9 13 16 19 24 27 30
index2: 2 6;1 3 4;2 4 5;2 3 6 7;3 7 8;1 4 9;4 5 8 9 10;5 7 10;6 7 10;7 8 9
         a    b     c      d      e     f       g         h      i     j
```

Then both X and Y have 10 elements. We observe the following program segment,

```
DO i = 1, numNodes
  X(i) = 0
  DO j = 1, numEdges(i)
    X(i) = X(i) + Y(index2(index1(i)+j))
  END DO
  X(i) = X(i)/numEdges(i)
END DO
```

Once again, we see two levels of indirection in this case, AND some 'hybrid' situation, i.e. array indexed by an expression involving an indirection vector and an index variable.

From the above examples, we summarize the following.

- A program presents itself as an *irregular problem*, if it has an array indexed by a vector;
- We may distinguish two kinds of irregular problems, *pure* and *hybrid*, such as `X(V(i))` and `X(V(i)+j)`, respectively.
- Applications associated with sparse matrices as underlying data representation that are implemented as a set of 1-dimensional arrays often result in irregular problems.

12.2 Recognizing compilation issues by examples

We start from simple (trivial ?) cases. These cases may not be realistic in the sense that we do not see them often in real programs, but I see them demonstrate some essential issues in dealing with irregular problems, in addition to that they seem a natural extension from what we have done in previous chapters.

In particular, we discuss the following 3 cases within the framework we have established so far.

(1) `X(V) = s`, where V is an index vector, s is some scalar constant.

(2) `X(V) = Y(W)` , where both V and W are indirection arrays, and Y and X are different arrays.

(3) `FORALL (i=1:n) X(V(i)) = i` , where we assume `1:n` covers entire V, though X may have more elements.

Let's assume all arrays are regularly distributed (as in HPF) or replicated.

12.2.1 Brute-force approach

We first investigate some translation schemes that merely demonstrate semantical equivalence between HPF code and `FORTRAN 77` code (multi copies).

```
X(V) = s   ===> DO i = 1,n
                  X(V(i)) = s
                END DO
```

The assignment is viewed as a DO loop (note: not all parallel assignment statements can be treated this way). Then node program may look like

```
DO i = 1, n
  CALL bcast(index,V,DAD_V,i)
  it = global2local(DAD_X,1,index)
  IF (it.NE.-1) X(it) = s
END DO
```

It's not difficult to realize this node program will work, though at any time only one processor is doing the assignment — no parallel computation.

```
X(V) = Y(W) ===> DO i = 1,n
                   X(V(i)) = Y(W(i))
                 END DO
```

Note this reduction is safe only if $X \neq Y$. We then have node program,

```
DO i = 1, n
  CALL bcast(index_l,V,DAD_V,i)
  CALL bcast(index_r,W,DAD_W,i)
  CALL bcast(s,Y,DAD_Y,index_r)
  it = global2local(DAD_X,1,index_l)
  IF (it.NE.-1) X(it) = s
END DO
```

Once again, very inefficient, but works. A similar treatment may be applied to the `FORALL` statement,

```
FORALL (i=1:n) X(V(i)) = i  ===>  DO i = 1,n
                                    X(V(i)) = i
                                  END DO
```

with node program,

```
DO i = 1, n
  CALL bcast(index,V,DAD_V,i)
  it = global2local(DAD_X,1,index)
  IF (it.NE.-1) X(it) = i          ! no local2global
END DO
```

Yes, we don't need `local2global`, since the DO index is in fact global.

12.2.2 Try to do better

The above approach is neat and can be easily generalized, but obviously inefficient in many ways. For instance, are the communications necessary ?

```
           weak
X(V) = s   ====>    DO i = 1,n          X: * * - * - - * * *
         reduction    X(V(i)) = s           | |   |     | | |
                    END DO                  s s   s     s s s
```

We realize if V is replicated, communication can be removed easily, as in,

```
DO i = 1, n
  it = global2local(DAD_X,1,V(i))
  IF (it.NE.-1) X(it) = s
END DO
```

though the assignment is still executed sequentially.

For distributed V, things may be quite different. For instance, is the following correct ?

```
CALL loop_bounds(DAD_V,1,1,n,1,l,u,str)
DO i = l, u, str
  X(V(i)) = s
END DO           ! No -- V(i) is global quantity
```

Consider `V = (/1,6,2,10/)` block-distributed on 2 processors and X is X(1:10), also block-distributed. The loop_bounds() will return 1,2,1 for both processor; Processor 1 will try to reference X(1) and X(6) — global ; Processor 2 will try to reference X(2) and X(10) — global ; — but local array X has only 5 elements.

How about the following ?

```
CALL loop_bounds(DAD_V,1,1,n,1,l,u,str)
DO i = l, u, str
  it = global2local(DAD_X,1,V(i))
  IF (it.NE.-1) X(it) = s
END DO           ! No -- i is local quantity
```

Wrong again, consider the same example, `global2local()` will return 0, -1 and -1, 4 for 2 processors, respectively. As a result, only global X(1) and X(10) get assigned.

Some additional runtime functions may help ! In fact, to deal with irregular problems effectively, people have developed some useful runtime functions as well as approaches. We'll develop some of the runtime functions as we analyze the requirement and try to do a better job than the above.

We first note that for `V(1:n)`, `X(V)` at left hand side implies n distinct elements of X are updated in parallel (no particular order, or any order).

Thus, we may design a runtime function

```
reorder(V0,V,DAD_V,DAD_X,u)
```

where V0 is a replicated array of n elements, the function reorders elements of V into V0 with respect to DAD_X, and also gives local element count u.

For instance, X has 12 elements, `V = (/1,3,10,2,6,4/)`, both block-distributed on 2 processors. After reorder(),

```
Processor 1                    Processor 2
V0 = (/1,3,2,6,4,*/)           V0 = (/10,*,*,*,*,*/)
u = 5                          u = 1
```

If X is cyclic-distributed, we would have

```
Processor 1                    Processor 2
V0 = (/1,3,*,*,*,*/)           V0 = (/10,2,6,4,*,*/)
u = 2                          u = 4
```

Then,

```
CALL reorder(V0,V,DAD_V,DAD_X,u)
DO i = 0,u-1
   it = global2local(DAD_X,V0(i))
   X(it) = s   ! we don't have to test "it == -1"
END DO     ! assignment is done in parallel, though load may be inbalanced.
```

Note, `reorder()` needs communication and may be costly. The advantage of this approach becomes evident if X(V) is referenced multi times, say in a repeated loop. Then V0 can be reused.

Also note: size of V0 in this case is actually the local size of X, since the values in V are not repeated as in X(V) = s.

A further improvement is to remove the global2local() call in the loop. This can be done by enhancing reorder() to

```
reorder_localize(V0,V,DAD_V,DAD_X,u)
```

such that V0 holds local indices. For the above example, we then have

```
Processor 1                    Processor 2
V0 = (/0,2,1,5,3,*/)           V0 = (/3,*,*,*,*,*/)
u = 5                          u = 1
```

and

```
Processor 1                    Processor 2
V0 = (/0,1,*,*,*,*/)           V0 = (/4,0,2,1,*,*/)
u = 2                          u = 4
```

for block and cyclic distributed X, respectively. Node program would be

```
CALL reorder_localize(V0,V,DAD_V,DAD_X,u)
DO i = 0,u-1
   X(V0(i)) = s
END DO
```

It looks like we have no way to remove communication completely.
For `X(V) = Y(W)`, we try to extend the above idea and the function of remap.

- `reorder_localize(V0,V,DAD_V,DAD_X,u)`
- Invent `irremap(T,DAD_X,V,DAD_V,Y,DAD_Y,W,DAD_W)`

  T is a temporary array distributed the same as X. Then the node program segment,

```
CALL reorder_localize(V0,V,DAD_V,DAD_X,u)
CALL irremap(T,DAD_X,V,DAD_V,Y,DAD_Y,W,DAD_W)
DO i = 0,u-1
   X(V0(i)) = T(V0(i))
END DO
```

(We note if it is only for X(V) = Y(W), `irremap(X,DAD_X,V,DAD_V,Y,DAD_Y,W,DAD_W)` suffices. The `irremap()` may be a general function.)

For instance, both X and Y have 12 elements, V = (/1,3,10,2,6,4/), W = (/2,10,12,1,3,5/), all block distributed. After reorder_localize(),

```
Processor 1                    Processor 2
V0 = (/0,2,1,5,3,*/)           V0 = (/3,*,*,*,*,*/)
u = 5                          u = 1
```

`irremap(T,...)` will produce,

```
T = (/Y2,Y1,Y10,Y5,*,Y3/)      T = (/*,*,*,Y12,*,*/)
```

This approach would allow us to handle `X(V) = Y(W) + Z(U)`, by

```
CALL reorder_localize(V0,V,DAD_V,DAD_X,u)
CALL irremap(T1,DAD_X,V,DAD_V,Y,DAD_Y,W,DAD_W)
CALL irremap(T2,DAD_X,V,DAD_V,Z,DAD_Z,U,DAD_U)
DO j = 0,u-1
   X(V0(i)) = T1(V0(i)) + T2(V0(i))
END DO
```

though it appears not as efficient as we may want, in particular the use of temporary arrays T1 and T2 are not only space consuming, but also time consuming.

For the FORALL statement,

```
                          weak
FORALL (i=1:n) X(V(i)) = i  ====>     DO i = 1,n      X: * * - * - - * * *
                          reduction     X(V(i)) = i      | |   |     | | |
                                        END DO           3 1   6     4 2 5
```

Again, easy for replicated V,

```
DO i = 1, n
  it = global2local(DAD_X,1,V(i))
  IF (it.NE.-1) X(it) = i         ! no local2global
END DO       ! loop executed sequentially
```

For distributed V, the correspondence between i and V(i) needs to be recorded somehow. A runtime function map() may be designed for this purpose.

```
CALL reorder_localize(V0,V,DAD_V,DAD_X,u)
CALL map(g_index,V,DAD_V,DAD_X)      ! record (i,V(i)) relation
DO i = 0,u-1
   X(V0(i)) = g_index(i)
END DO
```

(We assume V is completely referenced, namely 1:n covers all elements.) This map() function should reflect the result of reorder_localize().

For instance, X has 12 elements, V = (/1,3,10,2,6,4/), both block distributed on 2 processors. (1,2,3, 4,5,6)

```
FORALL (i = 1:6) X(V(i)) = i
```

After the FORALL statement, global X should be `(/1,4,2,6,*,5,*,*,*,3,*,*/)` and after reorder_localize(), we see

```
Processor 1                  Processor 2
V0 = (/0,2,1,5,3,*/)         V0 = (/3,*,*,*,*,*/)
u = 5                        u = 1
```

and after map(),

```
g_index = (/1,2,4,5,6,*/)        g_index = (/3,*,*,*,*,*/)
```

Finally after the DO loop,

```
X = (/1,4,2,6,*,5/)              X = (/*,*,*,3,*,*/)
```

The three cases we've analyzed, though kind of superficial, should have exposed us with typical problems that one must encounter when dealing with irregular problems.

12.2.3 One step further, "real stuff"

Consider the second example in the first section again. The computational graph has 8 nodes and 15 edges with the following program segment,

```
DO i = 1, 15
   Y(edge1(i)) = Y(edge1(i)) + f(X(edge1(i),X(edge2(i))
   Y(edge2(i)) = Y(edge2(i)) + g(X(edge1(i),X(edge2(i))
END DO
```

Suppose `X, Y, edge1`, and `edge2` are all block-distributed on 2 processors. We want to investigate how this loop is executed parallelly on the processors.

In particular, we want the following to be the core of node program.

```
DO i = 1, u       ! (u=8 for proc 1, u=7 for proc 2)
  Y(edge1(i)) = Y(edge1(i)) + f(X(edge1(i)),X(edge2(i)))
  Y(edge2(i)) = Y(edge2(i)) + g(X(edge1(i)),X(edge2(i)))
END DO
```

To start easier, let's first consider a simplified situation

```
DO i = 1, 15
  Y(edge1(i)) = f(X(edge1(i)),X(edge2(i)))
END DO
```

Data layout

```
    X: x1,x2,x3,x4                  X: x5,x6,x7,x8
    Y: y1,y2,y3,y4                  Y: y5,y6,y7,y8
Local: (1),(2),(3),(4)          local: (1),(2),(3),(4)

edge1: 1 1 1 4 6 4 5 4                  2 2 2 3 5 3 7
               -   -                    - - - -   -
edge2: 2 4 6 6 7 7 7 5                  4 5 3 5 8 8 8
           - - - - - - -                -   -
```

"–" indicates off-processor references. We observe,

- To compute the Y, off-processor X(5),X(6),and X(7) are needed for processor 1, X(2),X(3),X(4) are needed for processor 2.
- Processor 1 also computes Y(5) and Y(6), which owned by processor 2, while processor 2 also computes Y(2) and Y(3) owned by processor 1.

Thus, the node program with communication would look like,

```
gather off-processor X into local buffer;
DO i = 1, u
  Y(edge1(i)) = f(X(edge1(i)),X(edge2(i)))
END DO
scatter_add off-processor Y from local buffer;
```

For the above to work,

- a communication schedule needs to be established before the "gather";
- allocate local buffer properly — this can be done by extending local arrays X and Y (a similar idea to ghost area we've discussed for regular problems);
- contents of edge1 and edge2 need to be adjusted to reflect local index, *and* the buffers (a similar idea to `reorder_localize()`;
- a schedule may be also needed for "scatter_add".

Thus, we will have the following before the loop begins.

```
   X: x1,x2,x3,x4,x5,x6,x7          X: x5,x6,x7,x8,x2,x3,x4
   Y: y1,y2,y3,y4, -, -             Y: y5,y6,y7,y8, -, -
Local: (1),(2),(3),(4),(5),(6),(7)     (1),(2),(3),(4),(5),(6),(7)

edge1: 1 1 1 4 6 4 5 4                  5 5 5 6 1 6 3
               *   *                    * * * *   *
edge2: 2 4 6 6 7 7 7 5                  7 1 6 1 4 4 4
          * * * * * *                   *   *
```

After the loop, local Y(5) and Y(6) are sent off from processor 1 to processor 2, *and* added to those in processor 2. Note, processor 2 also computes `Y(5)` and `Y(6)`.

To implement this scheme, we need a few more runtime functions.

- localize(edge1,DAD_edge1,DAD_Y,edge2,DAD_edge2,DAD_X,schedule)

  adjusts indirection arrays, as well as producing a communication schedule; the "schedule" returned can be viewed as a handle to some runtime data structure describing how communication is to be done.

- gather(X,DAD_X,schedule)

  Fetch off-processor data before computation;

- scatter_add(Y,DAD_Y,schedule)

  Send off-processor result after computation, and the receiver accumulates the result to its own.

This scheme is commonly known as *inspector-executor* model. In this model, a global parallelizable DO loop with irregular data access patterns is turned into two sections of code in corresponding node program. The first section is referred as *inspector* phase which generates necessary communication schedule. The second section is called *executor* phase which uses the schedule to perform communication and computation. What's a *schedule* ? From runtime point of view, a schedule is a data structure containing enough information for directing communication. As an instance, we may see the schedule for the `gather` operations in our example as,

```
schedule for gather {
   fetch_size: specify the # of elements needed from each processor;
   send_size: specify the # of elements to be sent to each processor;
   send_list: list of indices to data elements to be sent, one list
              per processor;
   recv_list: specify indices to put the coming data in the buffer,
              one list per processor;
}
```

In particular, we have,

```
                P1                  P2
fetch_size: [0,3]               [3,0]
send_size:  [0,3]               [3,0]
send_list:  P1 -> NULL          P1 -> 0,1,2
            P2 -> 1,2,3         P2 -> NULL
recv_list:  P1 -> NULL          P1 -> 4,5
            P2 -> 4,5,6         P2 -> NULL
```

Chapter 13

Concluding remarks

After 12 lectures, the course was completed, and the students implemented the little `minihpf` compiler successfully. Reviewing what has been done and considering what can be improved as an education process, I see the following:

- Compilation technologies for data parallel programming languages constitute a set of distinct and coherent knowledge units that are well suited for a senior or graduate course in computer science.

  `HPF` provides a sound basis for discussion of various language features in terms of compilation approaches. It is rich enough for instructors to tailor in unlimited number of ways.

- *Runtime oriented* compilation approach, (yes, I call it runtime 'oriented', instead of 'supported'), is of value, not only in compilation system research and development, but also as an education practice. It always emphasizes the 'bigger' picture, without getting lost in fine points.

- In terms of contents, besides some of the covered topics should be enhanced, I'd like to add at least one more topic: subprogram interface analysis, in which we'll see how `DAD` plays an essential role in carrying information across subprogram boundaries.

  I will not consider things like dependence analysis, DO loop restructuring techniques, interprocedural analysis in this course. For one thing, they seem not fit well with the spirit of data parallel languages, which advises the programmer: if you think it can be parallel, tell me explicitly. For another, I think they deserve a separate course on their own.

- A substantial term project is not only essential, but also very feasible. With network of workstations and MPI, plus some existing infrastructure, some meaningful data parallel compilation system can be constructed with modest effort in one semester. Although I did not do it this time, I think it's feasible to include more general right hand side expressions, multidimensional arrays, and procedures in the project design.

Bibliography

[1] Andrew Meltzer, "Kernel HPF", Version 3.0, Cray Research Inc., October, 1995, presented on HPFF.

[2] Z. Bozkus, A. Choudhary, and G. Fox, et al, "Fortran 90D/HPF compiler for distributed memory MIMD computers: design, implementation, and performance results," Supercomputing'93, pp. 351-360.

[3] Message Passing Interface Forum, "MPI: A Message Passing Interface Standard", May 5, 1994.

[4] W. Gropp, E. Lusk, and A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface. The MIT Press, Cambridge, MA, 1994.

[5] HPFF, High Performance Fortran Language Specification (version 1.0). May 3, 1993.

[6] C. Koelbel, D. Loveman, et al., The High Performance Fortran Handbook. The MIT Press, Cambridge, MA, 1994.

[7] International Standards Organization, ISO/IEC 1539: 1991, Information Technology — Programming languages — Fortran, Geneva, 1991.

[8] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, et al, Fortran 90 Handbook. McGraw-Hill Book Company, 1992.

[9] Walter S. Brainerd, Charles H. Goldberg, and Jeanne C. Adams, Programmer's Guide to Fortran 90. McGraw-Hill Book Company, 1990.

[10] Janes F. Kerrigan, Migrating to Fortran 90. O'Reilly & Associates, Inc, 1993.

[11] Digital, DEC Fortran User Manual, March 1992.

[12] NPACT, "HPFfe: a Front-end for HPF," NPAC Technical Report SCCS-771, NPAC at Syracuse University, May, 1996.

[13] NPACT, "The Implementation of an HPF Front-end," NPAC Internal Technical Report ISCCS 7xx, NPAC at Syracuse University, February, 1996.

[14] D. Gannon, J. K. Lee, et al, "SIGMA II: A tool kit for building parallelizing compilers and program analysis systems," IFIP Transactions, A-11, Programming Environments for Parallel Computing, N. Topham, et al, Ed. North-Holland, 1992, pp17-36.

[15] Antonio Lain, Compiler and Run-time Support for Irregular Computations. Ph.D. Thesis, UIUC, Oct. 1995, aka Technical report, UILU-ENG-95-2236.

[16] V. Balasundaram, "A mechanism for keeping useful internal information in parallel programming tools: the data access descriptor," J. of Parallel and Distributed Computing, 9(2) 154-170, June 1990.

[17] Utpal Banerjee, Dependence Analysis. Kluwer Academic Publishers, 1997.

[18] Ravi Ponnusamy (ed), A Manual for the CHAOS Runtime Library, Computer Science Department, University of Maryland, August 12, 1994.

Index