Pfortran Reference Manual

a Parallel extension of Fortran

Babak Bagheri
Advanced Science of Design, Inc.

Terry W. Clark
L. Ridgway Scott
The University of Chicago

Release 2.1
April 1999

April 9, 1999
CONTENTS

Contents

1 Introduction .................................................. 2
  1.1 Overview ................................................. 2
  1.2 IPfotran Translator Limitations .......................... 2

2 IPfotran Built-in variables, functions and conventions 2
  2.1 Built-in variables ........................................ 2
  2.2 Built-in subroutines ....................................... 2
  2.3 Process identifiers ........................................ 2
  2.4 Deprecated variables ...................................... 2

3 The Operator @ ................................................ 2
  3.1 Local vs. global values ................................... 2
  3.2 An example: sorting ...................................... 2
  3.3 Ranges of variables ...................................... 2
  3.4 Ranges of processors ..................................... 2
  3.5 Functions of myProc ...................................... 2

4 Reduction Operations ........................................ 2
  4.1 Reduction operations defined ............................. 2
  4.2 Reduction operation procedural interface and parameter typing ... 2
  4.3 Parameter conformance .................................... 2
  4.4 The nondestructive subroutine interface ................... 2
  4.5 Length one arrays ........................................ 2
  4.6 Intrinsic reduction operation ............................. 2
  4.7 Reduction operation applications ........................ 2
  4.8 Compilation of reduction operations ...................... 2

5 Arrays ........................................................ 2
  5.1 Arrays in assignments .................................... 2
  5.2 Arrays as parameters ..................................... 2
  5.3 Performance consideration: Array sections and the communication system .................................................. 2
CONTENTS

6 Sample programs ............................. 2
   6.1 Baton passing .................................. 2
   6.2 Column oriented LU decomposition .......... 2
   6.3 Grid problems ................................ 2
   6.4 Dimensional exchange ......................... 2
   6.5 Gaussian elimination revisited ................. 2
   6.6 Dining philosophers problem .................. 2

7 An extended example: particle dynamics .... 2
   7.1 Pair generation ................................ 2
   7.2 Particle force calculation ...................... 2
   7.3 Exchange of the JNB array ..................... 2
   7.4 The “shake” algorithm ........................ 2

8 I/O .............................................. 2
   8.1 Serial I/O from Disk .......................... 2
   8.2 Single Process I/O from Disk ................. 2
   8.3 Concurrent I/O ................................ 2
   8.4 Intermediate I/O .............................. 2

9 Typical Programming Errors in iPfortran ...... 2
   9.1 Conditional guards ............................ 2
   9.2 Inconsistent source and destination process identifiers ........ 2
   9.3 Processor number out of range ................. 2

10 Users guide .................................. 2
   10.1 Installing .................................... 2
   10.2 Invoking ...................................... 2
   10.3 Include files .................................. 2
   10.4 Switch summary ................................ 2
   10.5 Intrinsic names ................................ 2
   10.6 iPfortran Library ............................. 2
   10.7 Scratch memory and variables ................. 2
   10.8 Using iPfortran ............................... 2
   10.9 Sample Makefile for the iPSC/860 .......... 2
   10.10 iPfortran Programs and the KSR .......... 2
        10.10.1 iPfortran and TCGMSG .................. 2
        10.10.2 iPfortran and SHMSG .................. 2
CONTENTS

10.10.3 Compiling programs for SHMSG ................................. 2
10.10.4 Specifying processor configurations for SHMSG based programs 2
10.10.5 Running SHMSG based programs ................................. 2
10.10.6 Stopping SHMSG based programs ................................. 2
10.10.7 Things to look out for ............................................ 2
10.11 KSR specific limitations ............................................ 2

11 Current limitations and bugs ................................. 2
11.1 Communicating with nonexistent processors ................. 2
11.2 Communicating character strings .................................. 2
11.3 Arrays as expression operands .................................. 2
11.4 Reducing functions as operands in expressions ............. 2
11.5 Reducing functions and arrays .................................. 2
11.6 Reduction operations and processor subranges ............. 2
11.7 Overlapping of communication and computation ............. 2

12 IPfortran extensions to FORTRAN77 grammar ................. 2

13 Glossary ................................................. 2
1 Introduction

Пfortran is a language designed to facilitate the programming of multi-process, data-parallel applications. It is an implementation of the Пlanguage model; in this document we may use the terms interchangably.

1.1 Overview

The Пlanguages provide programmers with two operators to specify off process data. In a serial program, we can write

\[ i = j \]

with the clear intention to move the value at the memory location represented by \( j \) to the memory location represented by \( i \). The Пlanguages allow the same type of assignment, however, the memory need not be local, as in the following example in a two-process system

\[ i@0 = j@1 \]

with the intention to move value at the memory location represented by \( j \) at process 1 to the memory location represented by \( i \) at process 0. One way of looking at this is that the duality of the send and receive operations in the message-passing paradigm [8] is encapsulated in an infix operator. We say more about \( @ \) in Section 3.

The other operator consists of a pair of curly braces with a leading function, \( f\{\} \). This operator lets one represent in one fell swoop the common case of a reduction operation where the function \( f \) is applied to data across all processes. For example, suppose one wanted to find the sum of an array distributed across \( nProc \) processes, with one element per process. One could write

\[ \text{sum} = +\{a\} \]

where \( a \) is a scalar at each process, but can be viewed logically as an array across \( nProc \) processes. We discuss reducing functions in detail in Section 4. It turns out with \( @ \) and \( \{\} \) a rich variety of operations involving off-process data can be concisely formulated.

The Пfortran program is written in the SPMD style of programming. In a nutshell, that means programmers have access to the local process identifier called \( myProc \).
With *myProc*, the programmer distributes data and computational workload.\(^1\) The \[\text{IP} \text{fortran} \] translator generates generic calls to a system-dependent library from user-supplied expressions containing communication operators.

**PLanguage Advantages**

The notation of \[\text{IP} \text{fortran} \] allows parallel algorithms to be expressed quite efficiently and explicitly. It is similar to notation that has been used solely in the context of describing parallel algorithms [11].

Abstracting the send and receive (or put and get) with an operator results in

- streamlined code easier to reason about,
- reduced development time, and
- portability

without degrading performance on message-passing systems [10]. Errors in writing explicit message-passing logic are reduced by passing some of the book keeping and code generation to the translator. System-dependent functionality is limited to a library of routines, facilitating portability of source programs.

**The Model**

\[\text{IP} \text{fortran} \] makes the important assumption that each processor knows the names of the variables in all processors. To this end we require that all processors run the same \[\text{IP} \text{fortran} \] code; our programming model is Single-Program Multiple-Datastream, or SPMD. The programmer, using a *local* view of the data, is responsible for its distribution and access.

\[\text{IP} \text{fortran} \] programs are written using the *local* approach, with explicit logic for each computational element (processor or process) with data decompositions propagated by programmer-supplied control structures. This methodology, while crude relative to globally-viewed, automatic decompositions [2], can provide greater flexibility. Moreover, industrial-strength codes have been ported to \[\text{IP} \text{fortran} \] in a short amount of time [1].

With the \[\text{IP} \text{fortran} \] programming model, the logical processor number pinpoints the data item in an expression through the use of the \(\@\) (Section 3) where \(i\@p\) represents a different variable from \(i\@q\), if \(p \neq q\).

\(^1\)Computational workload translates to loop iterations in the targeted technical applications.
1 INTRODUCTION

Since each Fortran program operates on its own memory the question arises, what can be different at each processor and what can be the same? Prior to the execution of the first instruction, all processes have the same data and instructions, and therefore are in identical states. The key difference between the $P$ processes is the variable $myProc$, the logical processor number. Any logic in the program involving the variable $myProc$ can potentially create a condition whereby two or more processes are in different states, meaning that for at least one variable name in the replicated program (recall the SPMD model), there will be different values. Without the variable $myProc$ and using the the SPMD model on identical processors, we would have rather uninteresting computations.

As a consequence of the logical processor number $myProc$, the values at each processor, corresponding to some variable, can be partitioned into two classes that we call local and global. Local values are (or more accurately, could be) different from other corresponding values at other processors. Global values are the same at all processors (in general, local and global can be applied to the set $Q \subseteq P$). With the current version of Pfortran, the programmer must keep track of which values are global and which are local. In any case, it is useful for the programmer to be aware of the distinction as it can affect efficiency. In general, target and source values must be global for those processors participating in a communication (see Section 9).

This manual gives numerous examples ranging from single lines of code to complete, working parallel programs. The internal workings of the Pfortran system are described slightly beyond what is necessary to write programs; we hope that this information assists the programmer in avoiding common pitfalls in using the language, in addition to arming the programmer for creative application. One can get enough basics to write a parallel program by reading Sections 3, 9 and 10, although a complete reading is recommended.

Finally, one can always incorporate message-passing code with Pfortran code where Pfortran falls short, provided the message-passing model is available on the target system. We are interested in any scenarios requiring this approach and welcome feedback.

1.2 Pfortran Translator Limitations

The current Pfortran translator has some limitations not inherent to the Planguages, but limitations arising from the implementation. We bring these to the reader's attention early in this document. If some of the points are not clear, you can refer back to them later. These limitations are due in part to limited resources at the time of
the translator’s implementation, others to the formulation of the language at that
time. The translator under construction will lift all of the restrictions covered below.
(Also see section 11.)

Some notes about the restrictions. The restrictions 1 and 2 arise from the require-
ment that: all processes must see all off-process data movement statements, and all
processes must agree on the outcome. Think of an invisible counter at each communi-
cation statement that must be incremented in exactly the same way by all processes,
even if the communication does not occur. This is described in detail below. Restric-
tions 3 and 4 are concerned with the places where the \texttt{P} language operators can occur.
One would rightfully expect to be allowed to use an off-process variable expression
(e.g., \texttt{a@p}) in any position a Fortran variable is valid, but this is not always the case.

1. All statements containing \texttt{@} and \texttt{}} must be in the control flow in the same way at all
processes whether a process executes the statement or not.
For example, suppose a 4-process system. The following will cause a problem

\begin{verbatim}
if (myProc.EQ.0.OR.myProc.EQ.1) i@0 = i@1 ! DO NOT DO THIS
j@2 = j@1 ! process 1 and 2 are now not in agreement
\end{verbatim}

Instead, write it this way

\begin{verbatim}
i@0 = i@1
j@2 = j@1
\end{verbatim}

But not all conditionals are prohibited. The following code is fine when executed by all processes:

\begin{verbatim}
read(5,*) v ! all processes agree on the value of v
if (v.GT.10) then
 i@0 = i@1
 j@2 = j@1
endif
\end{verbatim}

The problem arises from the way processes communicate the incarnation of a variable. This is through
a tag and there is only one tag counter throughout the program. There are clever ways to implement
a solution, but a fully general solution requires some machinery, particularly for one-sided synchroni-
ation, but we digress.

2. All statements containing \texttt{@} must have the same values of source and destination pro-
cesses.

\begin{verbatim}
p = 0
if (myProc.EQ.0) p = 1
j = j@p ! DISASTER since p is not uniformly valued!
\end{verbatim}

The exception to this is functions of \texttt{myProc}. The following is fine

\begin{verbatim}
p = 0
j@myProc = j@p ! this is okay
\end{verbatim}

The semantics of the preceding two code pieces are entirely different. The point to notice are the
uniformity issues in the value of the process identifiers,

3. Composition of \texttt{@} with \texttt{}}. In principle, the following should be allowed, but it is currently not
supported:

\begin{verbatim}
i@f(j) = i@g(k) ! DO NOT DO THIS
\end{verbatim}

It will not work.
1 INTRODUCTION

4. Placement of @. At present, @ can only occur in assignment statements. Do not try this, for example:

    do i = 1, n@p      ! DO NOT DO THIS
    a(i) = 0
    enddo

but this is okay

    m = n@p
    do i = 1, m
      a(i) = 0
    enddo

5. For functions of myProc to work, the variable myProc must appear explicitly in the statement.

    p = myProc
    a@p = b@(p-1)      ! DEADLOCKS OR WORSE

Write it the simple way instead:

    a@myProc = b@(myProc-1)

Functions of myProc are described in section 3.5.
2  **P**fortran Built-in variables, functions and conventions

Pfortran defines the following variables for use in the Pfortran source code. These names will also be used throughout this manual.

2.1  Built-in variables

**myProc** is the logical processor number of the processor executing the program. This default name can be modified at compile time.

**nProc** is the total number of processes.

**nProc0** is the total number of processes - 1 (zero based counting).

**cubedim** a architecture-dependent variable defined to be the \[\log_2 nProc\]. The name historically arose from hypercube topologies.

2.2  Built-in subroutines

**psync** an all process barrier synchronization; i.e., no process progresses until all execute the statement.

**pclock(\textless value\textgreater )** returns double precision number with wall clock time.

2.3  Process identifiers

Processes are logically numbered 0,\ldots, nProc0. A process may be referred to in this document as a either processor or a node.

2.4  Deprecated variables

Pfortran no longer support the notion of **host** and **unix process numbers**. The host process can be simulated by using a designated process, for example, the process where **myProc** is equal to zero. By choosing node 0 (or nProc −1) for specialized tasks, a program may be correct with only one process.
3 The Operator @

Accessing variables at other processors can be done in Pfortran using a new binary operator, @, which indicates that a variable is at some processor. For example,

\[ x = y@0 \]

assigns the value of variable \( y \) of processor 0 to \( x \) at every processor (broadcast). Similarly

\[ x = y@ (n + 1) \]

broadcasts the value of variable \( y \) of processor \( n + 1 \) to \( x \) (at every processor). Mixed expressions such as

\[ x = y@p + z \]

assigns to \( x \) (at every processor) the sum of \( z \) at that same processor and \( y \) from processor \( p \).

Multiple uses of @ in a statement are permitted, but only in assignments (i.e. expressions involving =) or in reduction operations (Section 4). (@, in principle, makes sense in any expression, e.g. predicates, but has not been implemented there.)

The operator precedence in Pfortran, from highest to lowest precedence, is

@, **, (*, /), (+, -).

So, for example, \( a = b@p + 1 \) is not equivalent to \( a = b@(p + 1) \) and \( a@p + 1 = b@q \) is meaningless where \( a@(p+1) = b@q \) is not. @ is non-associative, therefore, \( x = y@i@p \) is illegal and \( x = y@(i@p) \) or \( x = (y@i)@p \) are legal.

We can now give a simple program segment to compute a sum of values at each processor, say in the variable \( y \). First, all processors broadcast their own value and then each sums the values independently.

\[
\text{DO 10 n=0, nProc-1} \\
10 \quad x(n) = y \oplus n \\
\text{sum=0.0} \\
\text{DO 20 n=0, nProc-1} \\
20 \quad \text{sum = sum + x(n)}
\]

3.1 Local vs. global values

This technique for summing values over all processors, while not efficient, serves to illustrate an important point concerning our assumption about communication in
3 THE OPERATOR @

In Fortran. In the first loop in the code above, the processor from which we receive data is a variable. However, at each step in the iteration the value of this variable is the same at each processor. Thus \( n \) denotes a value universally agreed upon at each iteration, a value \emph{global} to all processors, \( P \). If the value of \( n \) were different in different processors, then a \emph{local} value results and a more complicated implementation would be needed.

If values referring to processor numbers are \emph{global}, we can predict the communication pattern required for the data exchange. Otherwise, for processor numbers given by \emph{local} values, the communication pattern must be determined at run time. This is not difficult to do, but it requires a significant amount of overhead.

In a future version of \#Fortran we plan to do compile-time analysis to determine which variables referring to processor numbers are \emph{local} or \emph{global}. At the moment, the programmer should limit the use of variable processor numbers to \emph{global} values (see Section 9). Our experience has shown that this is not a significant hindrance.

An example of converting local variables to global ones is as follows. Consider the code fragment

\[
i = f(\ldots) \\
x = y @ i
\]

In this it is likely that \( i \) will not have the same values at all processors. The correct behavior will be obtained with the current compiler if this is replaced by the (equivalent) code

\[
i = f(\ldots) \\
d o \ j = 0, nProc - 1 \\
 \hspace{1cm} ithere(j) = i @ j \\
enddo \\
d o \ j = 0, nProc - 1 \\
 \hspace{1cm} x @ j = y @ ithere(j) \\
enddo
\]

The set of the values \( ithere \) can be collected in a much more efficient way using various algorithms, to be discussed subsequently.

3.2 An example: sorting

To illustrate the power of the operator \( @ \), we consider the algorithm for a bitonic sort [12]. We consider the simplest case in which each processor begins with only one
value, n, and the intent is for each processor to have only one value at the end, but in sorted order (processor 0 with the largest). We indicate a simple random number generator to seed n, but anything could be used. We have hard-coded this for a number of processors equal to a power of two in this case.

```c
integer i,j,d,locdim,n,x,iswch
n = mod(17*myProc , 63)
locdim = 1
ipadr=myProc
!do 1000 j=0, cubedim -1
!   d = locdim
!   locdim = locdim*2
!   do 1000 i=0, j
!      notd=MINOT(d)
!      ieven=MIAND(ipadr,notd)
!      iodd=MIOR(ipadr,d)
!      x@ieven = n@iodd
!      x@iodd = n@ieven
!      iswch= mod((ibir(myProc,locdim)+ibir(myProc,d)),2)
!      if( iswch .EQ. 1 ) then
!         n = min(x,n)
!      else
!         n = max(x,n)
!      endif
!      d = d/2
100    continue
1000   continue
```

In this, we have used some bit-manipulation routines. Here, MIOR, MINOT and MIAND are the logical “or”, “not” and “and” functions, respectively. Similarly, bit is a bit-manipulation routine that returns the bit value in a given bit location. These can be defined in C as follows, and linked with Fortran on many systems.

```c
int minot_(mask)
   int *mask;
   {return(~*mask);
   }
int miand_(mask,nask)
```
int *mask,*nask;
{return(*mask&&nask);
}
int mior_(mask,nask)
int *mask,*nask;
{return(*mask||nask);
}
ibit_(m,d)
int *m,*d;
{
    if( *m & *d ) { return( 1 );}
    else { return( 0 );}
}

Exercise. Run the bitonic-sort routine with $2^k$ processors for various values of $k$.
Use \( n = \text{mod}(131*\text{myProc} , 16383) \) as your seed.

3.3 Ranges of variables

It is often necessary to exchange collections of variables among different processors, not just singletons. This is allowed in Pfortran using standard array syntax, such as \( x(7:11) \) to denote the range of variables \( x(7), x(8), x(9), x(10), x(11) \). For example, if \( \text{nProc} = 2**\text{CUBEDIM} \) then the array \text{ithere} can be collected via the loop

\[
\text{do j=0, nProc-1}
\quad \text{ithere}(j) = -\text{myProc}
\quad \text{enddo}
\quad \text{ithere}(<\text{myProc})=\text{myProc}
\quad \text{muchmo} = 1
\quad \text{mask} = 1
\quad \text{mmask} = \text{nProc} - 1
\quad \text{ipadr}=\text{myProc}
\text{DO 123 idim = 1, CUBEDIM}
\quad \text{notmask}=\text{MINOT(mask)}
\quad \text{ieven}=\text{MIAND(ipadr,notmask)}
\quad \text{iodd}=\text{MIOR(ipadr,mask)}
\quad \text{ivevbot}=\text{MIAND(mmask,ieven)}
\text{123 CONTINUE}
\]
ievtop = ievbot + muchmo - 1
iodbot = MIAND(mmask, iodd)
iodtop = iodbot + muchmo - 1
ithere(iodbot:iodtop)@ieven = ithere(iodbot:iodtop)@iodd
ithere(ievbot:ievtop)i odd = ithere(ievbot:ievtop)@ieven
mask = mask*2
mmask = mmask*2
123 muchmo = muchmo*2

Here, OR, NOT and AND are the logical "or", "not" and "and" functions, respectively (see Section 3.2). At each step, twice as much is exchanged with the neighbor in the idim-th dimension in a hypercube, so that after \( \log_2(nProc) \) steps, the entire array has been exchanged. We note that this algorithm would execute correctly on any architecture but there could be potential communication contention on ones that are not based on a hypercube network.

Ranges of variables will be discussed at more length in Section 5.

Exercise. Give an algorithm for concatenating ithere suitable for a ring, mesh and fat tree.

Exercise. Modify the bitonic sort algorithm to sort a list of numbers whose length is \( m \) times the number of processors, for some integer \( m \). (Hint: first sort in each processor before exchanging an array of numbers.)

### 3.4 Ranges of processors

There are many instances in which sets of processors or arrays of values are useful objects. We have implemented one class of such sets, namely ranges. The construct

\[
<expr> \oplus (<expr>_1 : <expr>_2)
\]

refers to the value of \(<expr>_1\) in the range of processors from the value of \(<expr>_1\) to the value of \(<expr>_2\). It can appear on the left-hand side of an assignment statement. For example,

\[
x@1:5 = y@0
\]

means: “Assign the value of variable \( y \) at processor 0 to variable \( x \) at processors 1 through 5.” On the right-hand side, the range logically returns multiple values. For expression

\[
x@i:j = y@r:s,
\]
3 THE OPERATOR @

the range \((r : s)\) will map 1:1 onto the range \((i : j)\) on the left-hand side; if \(j - i \neq s - r\), then the statement is not defined with unpredictable results. In addition to a single range or scalar processor identifier, lists of scalars and ranges are allowed, but also with a 1:1 correspondence if more than one processor is specified on the right-hand side. Some examples follow:

\[
\begin{align*}
    a@i, j, k &= b@m, n, o \\
    a@1, 2, 3, 4 &= b@0 \\
    a@(k + m : n) &= b@0
\end{align*}
\]

3.5 Functions of myProc

This section deals with the special case where one of the process identifiers is a function of the process identifier, myProc. Using standard compiler algorithms, analysis can detect the dependence of a variable’s value on myProc. The expression

\[ x = y@\text{mod}((\text{myProc} + 1), n\text{Proc}) \]

means: assign the value of variable \(y\) of processor \((\text{myProc} + 1)\) (modulo \(n\text{Proc}\)) to \(x\) in myProc. We can have any expression which mentions myProc explicitly (including functions which return integers, arithmetic operators, etc.).

An example using functions of myProc can easily be constructed to give a slightly more efficient summation algorithm. Think of our nodes as making up a simple ring. Then we can sum by accumulating and passing partial sums around this ring:

C Ring Sum WITH functions of myProc

```
program ringSum1
    integer n, sumIn, sumOut
    sumOut = myProc
    do n = 1, nProc-1
        sumIn@myProc = sumOut@MOD(myProc+1,nProc) + myProc
        sumOut = sumIn
    enddo
    print*,myProc, sumOut
end
```

This has the effect of computing the \(n\)-th partial

\[
(...(y@me + y@(me + 1)) + \cdots + y@(me + n + 1))
\]
at the \( n \)-th iteration, where node indices in the latter representation must be identified modulo \( n_{Proc} \). Given the program listing above, what conditions allow a single variable to serve for \( sumIn \) and \( sumOut \)?

It's also possible to implement the ring summation algorithm without functions of \( \text{myProc} \) by adding an inner loop over logical processor numbers:

```c
C Ring Sum WITHOUT functions of \text{myProc}
    program ringSum2
    integer n,p,sumIn,sumOut
    sumOut = myProc
    do n = 1, nProc-1
        do p = 0, nProc-1
            sumIn%p = sumOut%MOD(p+1,nProc) + p
        enddo
        sumOut = sumIn
    enddo
    print*,myProc, sumOut
end
```

*Exercise.* Use the technique introduced in Section 3.1 to convert \textit{ringSum1} to a program involving only global values.
4 Reduction Operations

Having introduced a formalism for distributed variables via the @ operator, we can now think of them as a new data type. It is natural to consider appropriate operators on this new data type. In this section we introduce one class of such operations based naturally on corresponding operations on each component of the distributed data structure. Such operations may, moreover, be defined on a restricted set of distributed data. (As mentioned above, an expression such as $x@i:j$ may refer to many variables. However, the current implementation of Pfortran reduction operations only supports the case with the entire set of the distributed values.)

The concept of a reduction operation was introduced to accomplish a specialized operation on a set of data; the operation is a reduction in the sense that a set of data is reduced to a single datum of the same type through the repeated application of a binary operator on pairs of the data. Furthermore, the operation is defined such that the result will itself be distributed in the same way the operands were distributed.

4.1 Reduction operations defined

To allow reduction operations, Pfortran introduces the following constructs:

\[
\begin{align*}
\langle id\rangle & = \langle id\rangle \{ \langle expr\rangle [,.opt\ args]\} \\
\text{call} & \quad \langle id\rangle \{ \langle expr\rangle , \langle id\rangle [,.opt\ args]\}
\end{align*}
\]

where \(\langle expr\rangle\) can reduce to a variable or constant. Functions and subroutines used along with \(\{}\) are called reducing functions and reducing subroutines, respectively. Some examples of reduction operations that one may reduce include: SUM, MIN, MAX, OR, and AND. Pfortran provides the reduction operation + described in Section 4.6. Other binary operators can be cast as reduction operations as described below.

The distributed value is represented by the symbolic name closest to the left curly brace \(\{}\). The set notation emphasizes the notion of a collection of data objects, distributed in some manner across a collection of processors. function_name (respectively, subroutine_name) must be a function (respectively, subroutine) of at least two (respectively, three) parameters. Functions must return a value of the same type as \(\langle expr\rangle\). The target variable is given by the left-hand side when using the function syntax, or as the second parameter, with procedure syntax (following the familiar mnemonic of in,out for parameter order). The designated operator, encapsulated in a
function or subroutine, is then applied to the set of distributed values. As with other
Fortran statements, Fortran90 array syntax may be used (see Section 5 Arrays).

We symbolically write the reduction operation for a $p$-process model
\[ A = \oplus \{ B \} \]
or
\[ \text{call } \oplus \{ B, A \} \]
as
\[ A = B_1 \oplus B_2 \oplus \cdots \oplus B_p . \]
Since the order of application of $\oplus$ to the set of values is not specified, $\oplus$ must be
associative and commutative. (More precisely, we are assuming that the set of values
of the type of $B$ forms an Abelian semi-group with respect to the binary operator $\oplus$.)
Although the order of evaluation may depend on the particular architecture or even
the number of processors, we intend that will be deterministic in the sense that it will
always be done in the same order on a given machine with a given number of nodes.

These concepts are illustrated with a concrete example. Consider a user defined
reduction operation implementing a summation of a set of scalars distributed across
$p$ processors into a single scalar. Expressed mathematically,

\[ \forall j, \quad \text{result}_@j = \sum_{i=1}^{p} \text{scalar}_@i . \]

By defining a function, mysum,

```
REAL*8 FUNCTION mysum(in1,in2)
mysum = in1 + in2
RETURN
END
```

now we can write

\[ \text{result} = \text{mysum}\{\text{scalar}\} . \]

### 4.2 Reduction operation procedural interface and parameter typing

A reduction operation can be associated with either intrinsic functions or user
defined functions; we consider examples of each. Using the intrinsic Fortran functions,
\texttt{AMIN1}, the statement

\[ y = \text{AMIN1} \{ x \} \]
computes the minimum of all values of x in all processors and assigns this value to the variable y (at all processors).

The general form of the reduction procedure invocations are

function: \( \text{out} = f1\{\text{in} \} \)

subroutine: call s1\{in, out\}

where in is the distributed operand. \( f1 \) and \( s1 \) implement some binary operation with the results assigned to the distributed variable \( \text{out} \). The corresponding function and subroutine definitions are of the form

function: "type" FUNCTION f1(in1, in2)

subroutine: SUBROUTINE s1(in1, in2, out)

where "type" denotes the type (e.g. REAL or INTEGER) of the function \( f1 \). With the function, the returned value is passed through the function return mechanism whereas in using the subroutine, the optional parameter \( \text{out} \) is provided. In both cases, the parameters \( \text{in1} \) and \( \text{in2} \) are not used to return results.

The user-defined reduction operations may include optional parameters. These parameters are assumed to be global parameters, the same at all processors, and they are not communicated, but are included in the calling sequence as shown above. These parameters can carry type information for the parameters \( \text{in} \) and \( \text{out} \), or can be used in generating side-effects with the reduction operation.

Optional parameters are often useful. For example in FORTRAN77, without user-defined types, these parameters can carry type information. The contrived statement

\[
\text{CALL opuratur \{ in, out, param1, param2 \}}
\]

illustrates the use of optional parameters. The corresponding subroutine definition is

\[
\text{SUBROUTINE opuratur(in1, in2, out, param1, param2)}
\]

where \( \text{param1} \) and \( \text{param2} \) are optional parameters and \( \text{out} \) is the output parameter. We notice that the parameters supplied to the function or procedure implementing the binary operator may be used nondestructively, that is, read-only.

The reduction of a function, e.g. \( \text{out=opatr\{in, param1, param2\}} \), requires a function with the definition

\[
\text{REAL FUNCTION opatr\{in1, in2, param1, param2\}}.
\]

Example. Recall the array \( \text{ithere} \) from Section 3. Its accumulation may be accomplished as follows.
where the subroutine intplus is defined by

```
subroutine intplus(ithere, jthere, kthere, istart, n)
dimension ithere(1), jthere(1), kthere(1)
do j=istart, istart+n-1
   kthere(j)=ithere(j)+jthere(j)
dendo
end
```

### 4.3 Parameter conformance

The actual parameters in and out must conform in shape, and if out is an array, in must also be an array. The formal parameters in1 and in2 correspond to some pair of operands for an operation $B_i \oplus B_j$, using with the notation of Section 4.

The memory for at least one of the two input parameters, perhaps both, comes from a Pfortran scratch array; if possible, the output parameter out is used as a working memory through the duration of the reduction operation. This implementation detail is of practical concern only in assuring that the Pfortran temporary array is of sufficient size to accommodate intermediate results (see 8. User’s Guide).

The array-section actual parameter will be reshaped to rank 1 arrays, where the reshaping will construct the rank 1 array in array order from the array subsection in, provided in is not already rank 1. The array section syntax denotes the description of a subarray with a shape indicated by the subscripts; Section 5 discusses Pfortran array usage in detail.

### 4.4 The nondestructive subroutine interface

A slightly more efficient reduction operation with a subroutine interface can be defined for arrays by using Pfortran function syntax for the global operation. Pfortran does not support anything but scalar-typed functions for global operations using the syntax $a = \oplus \{b\}$ since FORTRAN77 only supports this type of function. However,
it is more convenient to write

\[ A = \oplus \{ A \} \]

than

\text{call } \oplus\{a_{in}, A_{out}\}

Therefore, using the nondestructive procedural interface, the user is forced to supply the additional array \( A_{in} \) even though the destructive use (i.e. read–write), would have been acceptable.

To allow a destructive subroutine interface, with the spinoff of a streamlined syntax, \texttt{F}fortran supports the statement such as \( \text{In} = \text{Globul}\{\text{In}\} \) where the user has defined the subroutine \texttt{Globul}:

\begin{verbatim}
SUBROUTINE Globul(In,InOut)
   Dimension In(1),InOut(1)
   InOut = In \oplus InOut
   RETURN
END
\end{verbatim}

\textit{Example.} Recall the array \texttt{ithere} from Section 3. Its accumulation may be accomplished as follows.

\begin{verbatim}
  izer0=0
  do j=0, nProc-1
     ither(j)=0
  enddo
  ither(myProc)=i
  ither = intplusnd\{ither\}
\end{verbatim}

where the subroutine \texttt{intplusnd} is defined by

\begin{verbatim}
subroutine intplusnd(ither,jther)
   dimension ither(0:nProc-1), jther(0:nProc-1)
   do j=0, nProc-1
      jther(j)=ither(j)+jther(j)
   enddo
end
\end{verbatim}
4.5 Length one arrays

The Pfortran translator generates a Fortran function interface and a subroutine interface for scalar or array parameters, respectively. Some care must be taken with regard to the interpretation of arrays of length one. In particular, note that \texttt{in(1)} denotes a scalar, whereas \texttt{in(1:1)} denotes an array (of length one). Similarly, if there is a statement such as \texttt{Dimension In(1)} then the parameter \texttt{In} (no parentheses) denotes an array. Thus it would not be correct usage to write \texttt{mysum\{In\}} in this case.

4.6 Intrinsic reduction operation

The accumulation of the values of the elements of an array into a target array of the same shape is a common operation worthy of an intrinsic function. We wish to write code for the following sum:

\[
\forall i, j, \quad A(i)@j = \sum_{k=1}^{p} A(i)@k.
\]

Using Pfortran we can write

```fortran
    do k = 1, P
        B = A@k
        do i = 1, n
            A(i) = B(i) + A(i)
        enddo
    enddo
```

Alternatively, we write the more concise expression using the Pfortran intrinsic reduction operator, +\{ \cdot \}, for example

\[
    A = + \{ A \}
\]

Note that the result of the user-defined reduction

\[
    \text{result} = \text{mysum\{scalar\}}
\]

described earlier can be achieved using the intrinsic \texttt{result = +\{scalar\}} just as easily.
4 REDUCTION OPERATIONS

4.7 Reduction operation applications

The reduction of a set of distributed data by a binary operator is a common occurrence in scientific and engineering computations. Having a concise and intuitive notation streamlines programs, resulting in enhanced readability. Furthermore, by flagging the operation in a single statement, efficient algorithms can be implemented by the translator; these operations are efficiently implemented in at most $\log_2 p$ steps on a hypercube or mesh network, where $p$ processors are involved. We now conclude this section with a few examples.

Consider a user defined reduction operation implementing the global $\text{minmod}$ function. In solving discretized hyperbolic partial-differential equations, the flux of a quantity at a point is characterized by the smaller of finite-differences of that quantity in some vicinity of the point because near a shock the larger values may be unreliable. The binary function $\text{minmod}$ is used to find the smaller value. It is defined as follows. Given two input values of the same sign, it returns the value with the minimum modulus (absolute value). When there is a sign disagreement between two values, zero is returned. (This corresponds to the case of a variable that canProcot decide whether it is coming or going.) We define the scalar function $\text{minmod}$ as

```fortran
REAL FUNCTION minmod(s1,s2)
   if (s1*s2 .gt. 0.0 .and. s1.lt.0.0) then
      C s1 and s2 are < 0
      minmod = AMAX1(s1,s2)
   else if (s1*s2 .gt. 0.0) then
      C s1 and s2 are > 0
      minmod = AMIN1(s1,s2)
   else
      C If s1 and s2 are not of the same sign or one is 0
      minmod = 0.0
   endif
   return
end
```

To compute the $\text{minmod}$ of a set of values $\text{myslope}$ over all processors, one simply writes $\text{slope} = \text{minmod}\{\text{myslope}\}$.

A slightly more complicated application of a reduction operation arises in Gaussian elimination with row pivoting. Suppose an $m \times n$ matrix is stored by rows across $P$
processors where \( \text{row}_i \) is assigned to processor \( q = \text{mod}(i - 1, P) \) and that processors are numbered \( 0, \ldots, P - 1 \). Define the space \( \mathcal{X} = \mathbb{Z}^+ \times \mathcal{R}^+ \), where \( \mathbb{Z}^+ \equiv \text{positive integers} \) and \( \mathcal{R}^+ \equiv \text{non-negative real numbers} \). This space represents the Cartesian product of \( \text{row numbers} \) with absolute values of \( \text{pivot entries} \) from the matrix. We now define a function \( C : \mathcal{X} \times \mathcal{X} \rightarrow \mathcal{X} \) where

\[
C((i,x),(j,y)) = \begin{cases} 
(i,x), & \text{if } x > y \\
(j,y), & \text{if } y > x \\
(i,x), & \text{if } i < j, \ y = x \\
(j,y), & \text{if } j < i, \ y = x \\
(i,x) = (j,y), & \text{if } i = j, \ y = x.
\end{cases}
\]

Thus, \( C \) is commutative and associative and can be utilized in a reduction operation. The following subroutine illustrates the use and definition of \( C \) to calculate a row-pivot element among \( P \) processors.

```fortran
SUBROUTINE pivot(B,m,n)
IMPLICIT NONE
PARAMETER (size=64,minp=1)
REAL*8 B(size/minp,size) ! the rows for this processor
INTEGER m,n ! matrix dimensions
INTEGER row,i
INTEGER k ! cycle in Gaussian Elimination

c pivot(1) has the value, pivot(2) the row number as a real
REAL*8 pivot(2),pivot2(2)

c LOOP OVER COLUMNS OF MATRIX
DO 500 k = 1, n

c basis: determine pivot row from local rows
pivot(1) = B(1,k)
pivot(2) = myProc+1
DO 20 irow = 2, m/minp
row = myProc + 1 + (irow-1)*minp
pivot2(1) = B(irow,k)
pivot2(2) = row
CALL Cpivot(pivot2,pivot)
20 continue

c determine pivot row globally
```
call Cpivot {pivot(1:2)}

c output results
   print* ,myProc, k, INT(pivot(2)), pivot(1)
500 continue
END

Note that we could have equivalently used the statement

   call Cpivot {pivot}

above, as pivot has been dimensioned to be of length 2. The definition of Cpivot is as follows.

   SUBROUTINE Cpivot(p1,p2)
   REAL* 8 p1(2),p2(2)
c calculate a pivot where p1 and p2 contain
c row(i,k) element and value k
c output tuple goes to p2
   INTEGER* 4 i,j
   REAL* 4 v,w
   v = p1(1)
   i = p1(2)
   w = p2(1)
   j = p2(2)
   if ( v .gt. w ) then
      p2(1) = p1(1)
      p2(2) = p1(2)
   else if ( v .eq. w ) then
   if ( i.eq.j ) then
      print* , Error: Cpivot() has same row p1 and p2
   else if ( i.lt. j ) then
      p2(1) = p1(1)
      P2(2) = p1(2)
   endif
   endif
   RETURN
END
Quite complex commutative operators can be devised. Consider the following approach to computing a matrix times a vector, `matvec`, that uses the reducing version of `addvec`, a subroutine that adds vectors. For this new data type, we define addition by introducing a subroutine that adds two vectors and returns their sum:

```
SUBROUTINE addvec(a,b,c,D)
  dimension a(1),b(1),c(1)
  DO 1 i=1,D
      1   c(i) = a(i) + b(i)
  RETURN
END
```

We now define matrix multiplication using the reducing version of `addvec`. In the following, note that $D$ is the dimension of the matrix, and $N << D$ is the number of columns stored in each node. The index $global(i)$, $i = 1, \ldots, N$, indicates how the columns are distributed across processors, but the details of a particular choice of such a distribution is left unspecified. One such choice is defined in the following:

```
global(j)=myProc + j * nProc
```

where $nProc = D/N$. The matrix is assumed to be distributed in a similar way.

```
SUBROUTINE matvec(res, mat, vec, D, N)
  dimension res(N), mat(D,N),vec(N), s(1024),t(N)
  DO 1 i=1, D
      s(i) = 0.0
      DO 1 j=1, N
          1      s(i) = s(i) + mat(i,j) * vec(j)
  continue
  addvec{s,t,D}
  DO 2 i=1,N
      2     res(i) = t(global(i))
```

Note that the function `addvec` is clearly associative and commutative; its global version adds all the components among all processors.

### 4.8 Compilation of reduction operations

The optimal implementation of a particular reduction operation will depend, in general, on the size of the data being exchanged. For example, the addition of large vectors...
4 REDUCTION OPERATIONS

\[ A = + \{ B(1:N) \} \]

on a hypercube message-passing system can be done more efficiently by recursively subdividing the vector and adding parts of the vector at each step when \( N \) is sufficiently large [4]. (However, for small \( N \) this approach is suboptimal [13].)

For a user-defined operation such as

\[ A = \text{myvecsum}\{ B(1:N) \} \]

it is beyond the scope of current compilation techniques to attempt to infer whether \( B(1:N) \) could be segmented and still give a correct result. Thus the MPfortran compiler takes a conservative approach when compiling user-defined reduction operations. However, for operations such as \( + \) which are defined in the language, a more efficient implementation can be used.

*Example.* Define a subroutine to do matrix multiplication, \texttt{matmul(A,B,C,N,LDA)}. When applied to a set of commuting matrices, e.g.

\[
\text{Call matmul}\{A,C,1000000,LDA}\}
\]

this operation is suitable as a reduction operation. However, \texttt{canProcot} be segmented in the same way that the \( + \) operation can be.
5 Arrays

In a distributed computing model where parallelism is expressed explicitly, it is essential to have some way of expressing operations on entire arrays or sections of arrays. IPfortran supplements FORTRAN77 array notation with a subset of Fortran 90 syntax. Array names may be used in three settings to reference

1. a single scalar $A(i, j)$
2. a section $A(:, j)$
3. the entire array $A$.

In IPfortran, any of these three references can be used in expressions where @ is used or as actual parameters to reducing procedure invocations; the array extensions are not recognized outside of expressions involving @ or reducing function references.

5.1 Arrays in assignments

IPfortran permits arrays to be referenced in simple assignments of the type $A(m : n) = B(m : n)@p$. In the assignment $A(2 : 4)@q = A(1 : 3)@p$, if $p = q$, then $A$ refers to the same array, and the subarrays, therefore, share storage. The assignment to the left-hand side is made using the values of $A(1 : 3)$ prior to the assignment. If $A(1 : 4) \equiv (1, 2, 3, 4)$ then after the assignment $A(1 : 4) \equiv (1, 1, 2, 3)$ and not $(1, 1, 1, 1)$. These semantics come to bear in, for example, broadcasts where $A(2 : 4) = A(1 : 3)@p$ results in the assignment $A(2 : 4) = A(1 : 3)$ at processor $p$.

5.2 Arrays as parameters

A common programming style in Fortran uses procedure interfaces to rename arrays or to change the offset into an array. With IPfortran global functions and procedures, the subscripted references to the actual parameters $A(i)$ and $B(j)$ are treated as scalar references: there is no guarantee that the actual parameters generated by IPfortran will be arrays.

The following subroutine definition and use, where $A$ and $B$ are rank-1 arrays, will result in incorrect code: call myglob{$A(ioff), B(ioff), n}$ where the definition is
SUBROUTINE myglob(in1,in2,out,n)
REAL in1(1), in2(1), out(1)
INTEGER n
DO i = 1, n
   out(i) = in1(i) + in2(i)
RETURN
END

The functionality that the preceding global definition incorrectly attempts, can be
accomplished with

SUBROUTINE myglob(in1,in2,out,n,off)
REAL in1(1), in2(1), out(1)
INTEGER n,off
DO i = off, n+off
   out(i) = in1(i) + in2(i)
RETURN
END

with the reduction operation invocation call myglob[A,B,n,ioff]. (With future
versions of Pfortran, these semantics may be replaced with those standard to For-
tran.)

While Pfortran does not enforce conformance across procedural interfaces, the
formal parameter at the very least will take the shape into account: Conformance
across the reducing procedure interface is managed through copying array sections.
Shape information is not available in the called routine in the present version of
Pfortran. As a parameter in a reducing function or procedure, the array section will
be copied into a rank 1 array in array-element order corresponding to the shape of
the array section. In effect, this rank 1 array will have the characteristics of the array
section referenced as the actual parameter.

Consider, for example, the array declared as DIMENSION A(3,3). The reference A
which is equivalent to A(:,:) and A(1 : 3, 1 : 3) is not equivalent to
A(1 : 2, 1 : 3); if A(3,3) = /1, 2, 3, 4, 5, 6, 7, 8, 9/ then the elements of A(1 : 2, 1 : 3)
in array element order are /1, 2, 4, 5, 7, 8/. Pfortran assumes that the procedures being
reduced via {} are independent of such index shifts.
5.3 Performance consideration: Array sections and the communication system

The use of array sections may add a copy overhead to the compiled Pfortran program. To transmit a block of data to another processor, the communication subsystem is given an address of a data buffer, a byte count and a destination address. The communication subsystem, expecting to find the data contiguous in memory, will either copy this to a system buffer, or send the data to the destination processor directly from the user’s address space.

Since the array section is a portion of some parent array, it may not be contiguous in memory. For sections derived from \( rank \geq 2 \) arrays, the array section may be copied into a buffer by the application in order to achieve contiguous storage, although this is done automatically by the Pfortran translator.
6 Sample programs

In this section, some fundamental parallel algorithms cast in Pfortran are presented.

6.1 Baton passing

Baton passing derives its name from a relay where a team of runners run in some sequence chosen by their coach, with the current runner in possession of the single baton. Each runner, except for the first, will wait their turn for the pass of the baton, which signals a runner’s turn to snap into action.

Baton passing is a way to order processes, so they can, for example, perform some activity in some order. The following program segment results in the processes outputting “hello world” according to their logical ordering.

```fortran
SUBROUTINE Baton
  INTEGER baton
  if (myProc.eq.0)
    print*, 'hello from',myProc
  baton = 1
  do i=1, nProc-1
    baton@i = baton@(i-1)
    if (baton.eq.myProc)
      print*, 'hello from',myProc
      baton = baton +1
  endif
enddo
END
```

Exercise. Predict the value of baton@i at the completion of this routine. Extend the routine by adding, just before the END statement, the following code to check your prediction.

```fortran
print*, 'baton at ', myProc, ' = ', baton
```
6.2 Column oriented LU decomposition

This example illustrates prudent application of cyclic assignment for an LU decomposition algorithm. Geist and Romine [6] describe the following algorithm for column-to-processor assignment of a matrix using $LU$ decomposition on an $n \times n$ matrix.

\begin{verbatim}
for k = 0, n-1
  if (I own column k) then
    for i = k+1, n-1
      l(i,k) = a(i,k) / a(k,k)
    end for
    broadcast l
  else
    receive l
    for (all columns j>k that I own)
      for i = k+1 to n-1
        a(i,j) = a(i,j) - l(i,k) * a(k,j)
      end for
    end for
  end if
end for
\end{verbatim}

The natural unit of decomposition, columns of an $m \times n$ matrix, leads to an efficient implementation of this algorithm using cyclic assignment; column $j$ belongs to processor $p$, where $p = \text{mod}(\text{column}_j, n\text{Proc})$ and $0 \leq \text{column}_j < n$. An alternative decomposition, contiguous assignment, leads to increased processor idling since the lower indexed columns are eliminated first.

In the Pfortran $LU$ decomposition example that follows, both the data and iteration distribution are illustrated. The two are intimately related, and depend of course on the arrangement of data in memory.

REAL*8 a(64), m(64)
C distribute columns cyclically
  do i = 1, 64
    if (myProc.eq.mod(i,nProc))
      call getcolumn(i,a)
  enddo
  do k = 1, N-1
C owner computes column of multipliers
  if (myProc.eq.mod(k,nProc)) then
  do i = k+1, n-1
    m(i,k) = a(i,k)/a(k,k)
  enddo
  endif
C broadcast column of multipliers
  m(k+1: N) = m(k+1: N)@mod(k, nProc)
C reduce columns
  do i = k+1, N
    if (mod(i,nProc).eq.myProc) then
      do j = k+1, N
        a(j, i) = a(i,j) - m(j)*a(k,j)
      enddo
    endif
    a(i,j) = a(i,j) - m(i)*a(k,j)
  enddo
enddo

Exercise. Modify this routine to perform banded LU factorization.
Exercise. Modify this routine to perform LU factorization with (partial) row pivoting.

6.3 Grid problems

Grid problems arise frequently in approximating the solution of differential equations via finite difference, finite element or integral equation methods. The simplest example results in the simple iteration

```
REAL*8 newsoln(1000000), solution(1000000), data(1000000)
INTEGER  bignum
bignum = 1000000
delta_x = 1.0/float(bignum - 1)
dxsq = delta_x**2
solution(1) = 0.0
solution(bignum) = 0.0
do i=2, bignum - 1
```
newsoln(i) = (solution(i-1)+solution(i+1)-dxsq*data(i))*0.5
do i=2, bignum - 1
   solution(i) = newsoln(i)
endo

This is the Jacobi iterative method applied to a standard finite difference method
for the two-point boundary value problem

\[-u'' = f \text{ on } [0, 1]\]
\[u(0) = u(1) = 0.\]

It is a very inefficient method, but it serves to illustrate the necessary communication
statements.

There is a simple way to distribute both the work and the data in this loop. Let us
assume that \(\text{bignum} - 1\) is divisible by \(nProc\); let \(k := (\text{bignum} - 1)/nProc\). Let \(x\) denote either \text{solution} or \text{newsoln}. We divide these up as indicated in the following
diagram.

\[
\begin{array}{cccccccccccc}
\ldots & \ldots & \text{processor} j-1 & \ldots & \text{processor} j & \ldots & \text{processor} j+1 & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots
\end{array}
\]

We let \(z\) denote a local copy of either \text{solution} or \text{newsoln}, with \(z(1)\) and \(z(k + 1)\)
reserved to hold values that will be needed from other processors. The above loop becomes

```fortran
REAL*8 newz(10000), oldz(10000), locdata(10000)
INTEGER bignum
bignum = 1000000
k = (bignum-1)/nProc
delta_x = 1.0/float(bignum - 1)
dxsq = delta_x**2
do j= nProc-1,1,-1
   oldz(1)@j = oldz(k)@j-1
endo
```
do j=0, nProc-2
  oldz(k+1)@j = oldz(2)@j+1
enddo

do i=2, k
  newz(i) = ( oldz(i-1) + oldz(i+1) - dxsq*data(i) )*0.5
enddo

Note the reversed order of traversal in the loop

do j= nProc-1,1,-1
  oldz(1)@j = oldz(k)@j-1
enddo

This is because the Fortran compiler implements communication using the *golden rule* on message-passing systems: first a processor sends data to other processors that need it, then it receives the data that it needs. The loop in this order allows the necessary sends and receives to occur simultaneously. In the reverse order, viz.

do j= 1, nProc-1
  oldz(1)@j = oldz(k)@j-1
enddo

all send-receives will occur sequentially. Processor 1 will not send data to processor 2 until it receives data from processor 0, and processor 2 will not send data to processor 3 until it receives data from processor 1, and so on.

Of course, if functions of *myProc* are correctly implemented, an alternate implementation is as follows.

```
REAL*8 newz(10000), oldz(10000), locdata(10000)
INTEGER  bignum
bignum = 1000000
k = (bignum-1)/nProc
delta_x = 1.0/float(bignum - 1)
dxsq = delta_x**2
oldz(1) = newz(k)@max((myProc-1),0)
oldz(k+1) = newz(1)@min((myProc+1),nProc-1)
do i=2, k
```
newz(i) = ( oldz(i-1) + oldz(i+1) - dxsq*data(i) )*0.5
enddo
do i=2, k
  oldz(i) = newz(i)
enddo

Exercise. Do the two-dimensional case. The original sequential code reads as follows.

REAL*8 newx(1000,1000), x(1000,1000), data(1000,1000)
INTEGER  bignum
bignum = 1000
delta_x = 1.0/float(bignum - 1)
dxsq = delta_x**2
do i=1, bignum
  x(1,i) = 0.0
  x(bignum,i) = 0.0
  x(i,1) = 0.0
  x(i,bignum) = 0.0
enddo
do i=2, bignum - 1
  do j=2, bignum - 1
    newx(i,j)=(x(i-1,j)+x(i+1,j)+x(i,j-1)+x(i,j+1)-dxsq*data(i))*0.25
  enddo
enddo
do i=2, bignum - 1
  x(i) = newx(i)
enddo

Use array syntax throughout your code to transmit the border information between processors.

6.4 Dimensional exchange

In this section we take a look at the hypercube dimensional exchange implemented by the Pfortran translator. For a hypercube, it generates the following code for $A = op\{B\}$:
6 SAMPLE PROGRAMS

real*4 o10000
real*4 o10001
integer*4 o10002
integer*4 o10003

Cp
   a = op(b)
   a=b
   o10002=1
   do 10 o10003 = 1, cubedim
      call pf_inctag()
      call pf_snd(xor(myProc,o10002),a,4)
      call pf_rcv(o10000,4)
      a=op(o10000,a)
      o10002=o10002 * 2
   10 continue

where cubedim = log₂P. Here, pf_inctag, pf_snd and pf_rcv are routines that are system dependent and implement tag-incrementing, message-sending and message-receiving, respectively. This is probably isomorphic to what one would write by hand. Without the reduction operations and using the @ one could write

k = 1
   do 60 d = 1, cubedim
      do 50 p = 0,nProc-1
         dst = xor(p,k)
         tmp@dst = a@p
         if (dst.eq.myProc) then
            a = op(tmp,a)
         endif
      50 continue
   k = k * 2
50 continue

Another loop is introduced in this algorithm adding an O(P) amount of work. We see that @ alone is not enough to write efficient algorithms, but coupled with reduction operations implemented via dimensional exchange (or other algorithm well suited to a given network topology, such as a mesh) it is usually enough.

A fan-out is another common operation well suited to hypercube topology. Here a single processor, say 0, sends a datum to processor 1. In the next step, processor 0
sends to processor 2, and processor 1 sends to processor 3, and so on. In \( \log_2 P \) steps all processors will have the datum initially at processor 0. In other words, processor 0 broadcasts its datum to all other processors. \( \text{P} \) fortran uses the broadcast mechanism supplied by the communication system (when available) for \( a = b @ p \).

The inverse of fan-out, fan-in, reduces data from a set of processes to a datum at a single processor. The reduction operation in \( \text{P} \) fortran accomplishes this, but with all processors participating in the exchange. While we intend eventually to produce code for syntax such as \( a = \text{op} \{ b @ (i : j) \} \) in \( \text{P} \) fortran, to communicate efficiently among an arbitrary subset of processes, for now only \( a = \text{op} \{ b \} \) is supported.

Using \( \text{P} \) fortran as it stands, one can design algorithms using the standard reduction operation simply by assigning null data to the non-participating processors’ variable. We consider an algorithm that accumulates values from a set of processors. In the following loop, let \( A(0 : P - 1) \) indicate a processor’s participation in a fan-in or dimensional exchange; if \( A(i) = 1 \) then processor \( i \) will contribute its data.

\[
\begin{align*}
\text{if} & \ (A(\text{myProc}).\text{ne.}1) \ \text{then} \\
& \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \text{contribute} = 0.\text{E}0 \\
\text{else} & \\
& \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \text{contribute} = \text{value} \\
& \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \text{endif} \\
& \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \text{accumulator} = +\{\text{contribute}\}
\end{align*}
\]

For a fan-in to a single processor, \( p \), one can write \( a @ p = +\{\text{value}\} \). For a hypercube topology, a slightly more efficient algorithm exists for a fan-in than the general algorithm (currently dimensional exchange) for \( a = +\{\text{value}\} \). However, it is not used at the moment; \( a @ p = +\{\text{value}\} \) is computed simply by computing \( a = +\{\text{value}\} \) and then specializing the assignment only to processor \( p \).

If the \text{P} fortran mechanisms cannot effectively express what one wishes to do, then the low level sends and receives can be used directly, and if appropriate, may be interspersed with \text{P} fortran statements.

### 6.5 Gaussian elimination revisited

We now consider a more serious application code to solve linear equations by Gaussian elimination (no pivoting) using an algorithm for distributed memory computers which allows the overlapping of communication and computation. This algorithm is quite similar to one described in [6]. We defer to that article for motivation and
explanation of the algorithm. In particular, we present the “pipelined” algorithm for
column storage. Included in the code is a test of the accuracy of algorithm using a
matrix a with a known factorization (given by b in the code). Some care must be
taken for large matrix sizes as the generation of a and b involves exponentially small
quantities and underflow can result.

```
real*8 a(1024,64), b(1024,64), m(1024), nextm(1024)
real*8 ainv, above, mflops, err

integer*4 t

integer*4 i,j,k,l

ipasign(j,nProc) = mod(j-1, nProc)
C processor assigned to store global column no. j of the matrix

iglobal(myProc,j,nProc) = myProc + 1 + (j-1)*nProc
C global column address of the j-th local column held by myProc

local(j,nProc) = ((j-1) / nProc) + 1
C local column address (in proc. ipasign(.j,)) of global
C column no. j in the matrix

matdim = 64
nocols = matdim/nProc

c write(6,101) myProc
101 format(' processor ', I4, ' beginProcing initialization ')
do 20 i=1,matdim
   do 10 j=1,nocols
      a(i,j) = 2**(-iglobal(myProc,j,nProc))
      if( i .eq. 1 .or.
         iglobal(myProc,j,nProc) .le. i ) then
         b(i,j) = a(i,j)
      else
         b(i,j) = 0.75*a(i,j)
      end if
   continue
10 continue
20 continue

t = mclock()

if ( myProc .eq. ipasign(1,nProc) ) then
C this processor computes first set of "multipliers" for Gauss elimination
write(6,103) myProc
103 format(' processor ', I4, ' computing multipliers at step 0')
   l = local(1,nProc)
   ainv = 1.0/a(1,l)

   do 30 i=2,matdim
```
6 SAMPLE PROGRAMS

    a(i,1) = a(i,1) * ainv

end if

m(2:matdim) = m(2:matdim)@(ipasign(1,nProc))

    do 80 k=1,matdim-2
       if (myProc .eq. ipasign(k+1,nProc) ) then
          C this processor computes next set of "multipliers" for Gauss elimination
          write(6,104) myProc,k

104      format(' processor ', I4, ' computing multipliers at step',I4)
           1 = local(k+1,nProc)
           above = a(k,1)
           do 40 i=k+1,matdim
               a(i,1) = a(i,1) - m(i)*above

               ainv = 1.0/a(k+1,1)

               do 50 i=k+2,matdim
                   a(i,1) = a(i,1) * ainv

40             nextm(i) = a(i,1)

               end if

50             nextm(k+2:matdim) = nextm(k+2:matdim)@(ipasign(k+1,nProc))

jstart = local(k+2,nProc)

    if( ipasign(k+2,nProc) .gt. myProc ) jstart = jstart + 1

if( jstart .le. ncols) then
       do 60 j = jstart,ncols
           above = a(k,j)
           do 60 i=k+1,matdim
               a(i,j) = a(i,j) - m(i)*above

60             continue

     end if

    do 70 i=k+2,matdim
        m(i) = nextm(i)

70     continue

    if(myProc .eq. ipasign(matdim,nProc)) then
       a(matdim,ncols) = a(matdim,ncols) - m(matdim)*a(matdim-1,ncols)
    end if

    write(6,111) myProc

111      format(' processor ', I4, ' finished factoring ')
           t = mclock() - t
           mflops = float(matdim*matdim*matdim)/(3000*t)

C Check that the answer is correct
   err = 0.0
   do 90 i=1,matdim
```fortran
   do 90 j=1,nocols
      err = dmax1( err , dabs(b(i,j) - a(i,j)) )
   90
   write(6,100) myProc, mflops, err
   100   format( ' processor ', i4, ' achieved ', e13.3,
              1               ' Mflops with an error of ', e13.3)
   end

The overlapping of communication and computation occurs because the variable
nextm (which holds the precomputation of the multipliers for the next, not the cur-
rent, step of elimination) is available long before it is needed (nearly one full step
of elimination in advance). This can be seen from the text of the code, as there are
several lines of code after the definition of nextm in statement 50 before its use in
statement 70.

A further transformation can be applied to this code whereby the send correspond-
ing to nextm is done as soon as it is possible (after statement 50 in this case) and the
receive is postponed until as late as possible (before statement 70 in this case). At the
moment, the send/receive pair appears adjacent in the code generated by the trans-
lator for message-passing systems. The code generated by the Pfortran translator for

   nextm(k+2:matdim) = nextm(k+2:matdim)%ipasign(k+1,nProc)

is similar to what appears in section (6.4) on dimensional exchange and will not be
repeated here.

6.6 Dining philosophers problem

With this example we veer from our numerical-applications trajectory and apply
Pfortran to a classic problem in concurrency control. Our objective is not to advocate
Pfortran for this type of parallel programming problem. Instead we include this
example to show how the inner workings of Pfortran can be used to achieve things
beyond the obvious. We do so in the spirit that Fortran is used by some as a macro-
processor. More precisely, we are referring to situations when one uses not only the
formal properties of the language but also knowledge of how its compiler works. Thus
we will show how the Pfortran translator works to some degree by showing how it
can be used to implement a procedural-parallelism solution to the dining philosophers
problem (a complete program text is included at the end of this section).

The dining philosophers are a group of five philosophers who from time-to-time
compete for chopsticks [3]. The philosophers spend their time alternating between
eating and thinking. When during thinking hunger strikes, the philosopher will enter
the dining room and take a position at a round table. On this table surrounded by
five chairs, are five plates and five chopsticks. A plate is located at each of the five
positions with a single chopstick placed in the spaces between plates. In order to eat,
a philosopher must have two chops in hand, taken from either side of the plate. It
is easy to see that problems could arise if all philosophers sat down simultaneously
and picked up the chopstick to their right; this is just one of many potential 
deadlock
scenarios. If the philosophers are not going to starve, then at least one must relent,
put down a chopstick, and try again later.

Solutions to the dining philosophers problem pose clever ways to manage the shared
and finite resource of chopsticks such that deadlock will not occur and such that no
philosopher will starve to death (bounded waiting). There are analogous cases with
operating systems where processes may compete for a finite resource, such as a cpu
or a printer. Counterparts also occur with numerical programming where processes
may be dedicated to performing some specialized task such as sorting output. Strictly
speaking, the SPMD model does not encompass this procedural-parallelism program-
ning model. With the SPMD model, all processors execute the same code, with
variations in branches of if statements or number of iterations for a loop, for in-
stance. In our implementation of Pfortran we take the SPMD model to heart and
require the symmetric execution of communication statements in order to produce
correct message passing code.

Various concurrency constructs and their ability to deal with the subtleties of dead-
lock and progress have been tested against the dining philosophers problem – mon-
itors, semaphores, Hoare’s CSP, to name a few. We present a Pfortran solution
whereby ten processes implement the five chopsticks and five philosophers.

Staying within the SPMD model, we use a single program. Chopstick processes will
execute one portion of the program, philosopher processes the other. In their respec-
tive code sections, each philosopher will move between thinking, becoming hungry
and trying to acquire the two chopsticks alongside their plate, and upon acquiring
two chopsticks, eating. The chopstick processes will take requests from the philoso-
phers, but will only allow a single philosopher to use them at any time. Two problems
challenge an SPMD solution: symmetry is violated and philosophers are hungry at
arbitrary intervals.

Chopsticks and philosophers communicate, but from separate program sections. So
how can we guarantee that messages will be read by the respective recipient? Correct
communication in Pfortran requires that each send be matched with a corresponding
receive, greatly simplifying the programmer’s task. The sends and receives originate
from the @, which are placed both where data is needed and where it is available. Since messages may be buffered, we need some way of labeling or tagging them. Herein lies an important ramification of the asynchronous SPMD model.

It should not be a secret that processes can astutely exchange data on a message passing machine with complete confidence about message origins in the program and at what incarnation of a cycle or cycles, through message tagging. With each send/receive pair generated by IPfortran, there is an accompanying tag incrementing operation. The tag is global to the program and is incremented once at each communication statement. The tag becomes part of a message sent and will be used by a receiving process to request the correct message for its point in execution. This allows processes to proceed asynchronously. For the scheme to work, however, all processes must execute all tag-increment operations.

If a process should miss a tag increment, then all bets are off. This is why symmetry is so important and why the SPMD model is so appealing. We’re fortunate that scientific applications fit so well into this model. The solution to the dining philosophers problem that we’ve outlined violates this symmetry, but since we know about message-tags, we can construct corresponding code sections for chopsticks and philosophers where the tags are incremented, but in different statements; chopsticks increment in their subroutines, and philosophers increment in their subroutines.

The program segment that follows uses the pair of communication statements

\[
\text{message}@\text{bamboo} = \text{activity}@\text{human}
\]

\[
\text{message}@\text{human} = \text{reply}@\text{bamboo}
\]

for philosopher to chopstick and chopstick to philosopher communication in the philosopher subroutine.

The left-hand side of the first statement and the right-hand side of the second statement are sleights of hand crafted for the IPfortran translator. The values of message and reply at bamboo in this subroutine (i.e. at this place in the code) are undefined – the chopstick processes never enter this subroutine. This is fine because in the chopstick subroutines, another pair of IPfortran statements have equally bogus right-hand and left-hand sides ostensibly claiming that data assignments are made using variables from the philosopher process.

Using the SPMD, data-parallel model to simulate a style of programming involving procedural parallelism has resulted in some obscure code. In addition, we are using a deterministic model for a fundamentally non-deterministic problem. Our dining philosopher solution works, however, because the programmer built in the necessary symmetry with an understanding of message tags. The key is that the message-number (tag) identifies the message needed for a correct algorithm. And what about
the sends and receives that are generated with the undefined data? Fortran generates the following code for the two statements in the philosopher subroutine

```fortran
Cpf message@bamboo = activity@human
    call pf_inctag()
    if (myProc.eq.bamboo.and.myProc.eq.human) then
        message=activity
    else if (human.eq.myProc) then
        call pf_snd(bamboo,activity,4)
    else if (bamboo.eq.myProc) then
        call pf_rcv(message,4)
    endif
Cpf message@human = reply@bamboo
    call pf_inctag()
    if (myProc.eq.human.and.myProc.eq.bamboo) then
        message=reply
    else if (bamboo.eq.myProc) then
        call pf_snd(human,reply,4)
    else if (human.eq.myProc) then
        call pf_rcv(message,4)
    endif
```

In the first translation, the receive by bamboo (chopstick) will never be executed, here. And likewise, the send in the second translation will always be skipped. However, the other send and receive will be executed as we want.

If one takes the straight and narrow with Fortran, these gymnastics are not of concern. This shows, however, why one should not write the statement

```fortran
if(myProc.eq.p) a@p=b@q
```

without good reason. We have explored a limitation of this programming model and its implementation; while a programming style using procedural parallelism can be developed, it is unnatural. Moreover, our solution to the dining philosophers problem falls short of the mark. There is progress without starvation and deadlock is avoided, but not without a price. In order to keep message tags in sync with the chopsticks, the philosophers are required to participate in table activities even while thinking, a violation of the original problem as formulated by Dijkstra.

The Fortran listing for the dining philosophers problem follows.
PROGRAM Philosophize
c Dining Philosophers Problem using \{P lang\}
c if (myProc .lt. 5) call philosopher(myProc)
if (myProc .gt. 4.and.myProc.lt.10) then
    call chopstick(myProc)
endif
stop
END

SUBROUTINE go_eat(activity,iseed)
IMPLICIT NONE
REAL fate, enough
INTEGER activity,iseed
PARAMETER(Enough=0.50)
INTEGER orate,null_message,busy,not_busy,
    . hungry,putdown,thinks,eats,true,false
PARAMETER(orate=1, null_message=0, busy=1,
    . not_busy=2, hungry=3, putdown=6, thinks=7,
    . eats=8, true=1, false= 0)
    if (fate(iseed) .gt. ENOUGH) then
        activity = putdown
    else
        activity = eats
    endif
    if (orate.eq.1.and.activity.eq.putdown) then
        print*, ' I am nourished and ready to THINK,'
        . ' Philosopher ',myProc
    else if (orate.eq.1.and.activity.eq.eats) then
        print*, ' I continue to EAT,'
        . ' Philosopher ',myProc
    endif
RETURN
END

SUBROUTINE go_think(activity,iseed)
IMPLICIT NONE
REAL fate, enough
INTEGER activity, ised
INTEGER orate,null_message,busy,not_busy,
    . hungry, putdown, thinks, eats,true,false
PARAMETER(ENOUGH=0.50)
PARAMETER(orate=1, null_message=0, busy=1,
    . not_busy=2, hungry=3, putdown=6, thinks=7,
    . eats=8, true=1, false= 0)
    if (fate(iseed).gt.enough) then
        activity = hungry
    else
        activity = thinks
    endif
if (orate.eq.1.and.activity.eq.hungry) then
  print*,’by virtue of wisdom I perceive HUNGER’,
  ’philosopher ’,myProc
endif
RETURN
END

REAL FUNCTION fate(itn)
IMPLICIT NONE
REAL a, ran
INTEGER itn
fate = ran(itn)
RETURN
END

SUBROUTINE philosopher(human)
IMPLICIT NONE
C node number of this philosopher
INTEGER human
C left and right chopsticks
INTEGER bamboo, bamboo0
C current chopstick index
INTEGER bamboo
C chopstick count, must be 2 to eat
INTEGER chopsticks
DATA chopsticks/0/
INTEGER message, reply
INTEGER activity
INTEGER orate, null_message, busy, not_busy,
  hungry, putdown, thinks, eats, true, false
PARAMETER(orate=1, null_message=0, busy=1,
  not_busy=2, hungry=3, putdown=6, thinks=7,
  eats=8, true=1, false= 0)
INTEGER debug, iseed
data debug/.false./
data activity/thinks/
iseed = human
bamboo = human + 5
bamboo0 = human + 5
bamboo1 = MOD((human + 1),5) + 5

c forever a philosopher
100 continue
  if (bamboo .eq. bamboo0 ) then
    bamboo = bamboo1
  else
    bamboo = bamboo0
  endif

message@bamboo = activity@human
message@human = reply@bamboo
if (activity.eq.thinks) then
    call go_think(activity, iseed)
else if (activity.eq.hungry) then
    if (message.eq.not_busy) then
        chopsticks = chopsticks + 1
    else
        chopsticks = 0
    endif
    if (orate.eq.1 .and. chopsticks.eq.2) then
        print('Philosopher ',human,' eats now'
    endif
    if (chopsticks .eq. 2 ) activity = eats
    if (chopsticks .eq. 1 ) activity = hungry
    if (chopsticks .eq. 0 ) activity = putdown
else if (activity.eq.eats) then
    call go_eat(activity, iseed)
else if (activity.eq.putdown) then
    if (chopsticks.eq.2) then
        chopsticks = chopsticks - 1
    else
        chopsticks = 0
    endif
    if (chopsticks .eq. 1 ) activity = putdown
    if (chopsticks .eq. 0 ) activity = thinks
endif
goto 100
C for philosophers lifetime
RETURN
END

SUBROUTINE chopstick(bamboo)
IMPLICIT NONE
INTEGER bamboo
INTEGER message,activity,reply
C philosopher adjacent to this chopstick
INTEGER human
INTEGER human0,human1
INTEGER orate,null_message,busy,not_busy,
    hungry,putdown,thinks,eats,true,false
PARAMETER(orate=1, null_message=0, busy=1,
    not_busy=2, hungry=3, putdown=6, thinks=7,
    eats=8, true=1, false= 0)
INTEGER ibusy
DATA ibusy/0/
human = bamboo-5
human0 = bamboo-5
human1 = MOD( human0+4, 5)

C always a chopstick
100  continue
if (human.eq.human0) then  
    human = human1  
else  
    human = human0  
endif

message@bamboo = activity@human

if (message.eq.hungry) then  
    if (ibusy.eq.1) then  
        reply = busy  
    else  
        reply = not_busy  
    endif  
    ibusy = 1  
else if (message.eq.putdown) then  
    reply = null_message  
    ibusy = false  
else if (message.eq.eats) then  
    reply = null_message  
else if (message.eq.thinks) then  
    reply = null_message  
endif  
message@human = reply@bamboo  
goto 100  
RETURN  
END
7 An extended example: particle dynamics

We now consider an extended application arising in the simulation of many-particle systems governed by Newtonian mechanics. We assume that there is a function $f(x, y)$ that determines the force between two particles at positions $x$ and $y$. For simplicity of notation, we will ignore the implementation details regarding the fact that the positions are themselves multidimensional. Let $x_i$ denote the position of the $i$-th particle in the system, with $i = 1, \ldots, n$. We will also assume that the total force on the $i$-th particle is given by the sum of pairwise forces, namely,

$$F(x_i) := \sum_{j=1}^{n} f(x_i, x_j).$$

The motion of the system of particles is then governed by the system of ordinary differential equations

$$F(x_i) = m_i \frac{d^2x_i}{dt^2}$$

where $m_i$ denotes the mass of the $i$-th particle. To simplify notation in the code, the pairwise force function will be denoted by ourforce.

The computational problem associated with approximating the solutions of this system of equations is quite simple if $n$ is small. However, for large $n$ the computational effort becomes prohibitive and several simplifications are utilized to reduce the cost. One such technique is to use a cut-off radius $R$. The force $F(x_i)$ is split into two parts as follows:

$$F(x_i) := \sum_{|x_i-x_j|<R} f(x_i, x_j) + \sum_{|x_i-x_j|\geq R} f(x_i, x_j).$$

The part of the force arising from particles outside the cut-off radius is updated less frequently, reducing the computational cost per time-step to $O(nR^3 + cn^2)$ from $O(n^2)$.

Another simplifying technique is to constrain certain particles to move as a group, rather than separately. Such an example might be the three particles comprising a water molecule. This turns the system into an algebraic o. d. e. and complicates the code significantly, but it allows much larger time steps to be utilized in many cases. In molecular dynamics, this technique is known as the shake algorithm (see Section 7.4). We will consider the effect both of these techniques has on the development of a parallel code for particle dynamics.
7.1 Pair generation

The use of a cut-off radius requires the generation of a “pair list” \( JNB \) which keeps track of which particles are within the cut-off radius. \( JNB \) can be determined via the following pseudo code. The array \( INB \) records the numbers of particles within the cut-off radius for each particle.

C Pairlist computation: original version
J0 = 0
JJ = 0
for I = 1, N
    for J = I+1, N
        if( \( | X(I) - X(J) | < R \) ) THEN
            JJ = JJ + 1
            JNB(JJ) = J
        endif
    endfor
INB(I) = JJ - J0
J0 = JJ
endfor I

The \( JNB \) array lists all neighbors in a linear data structure that is very efficient to use. Additional arrays, \( start \) and \( end \), are needed in order to access it, as follows.

\[
\begin{align*}
\text{start}(i) &= 1 \\
\text{end}(i) &= \text{INB}(i) \\
\text{for } I = 2, N \\
\quad \text{start}(i) &= \text{end}(i-1) + 1 \\
\quad \text{end}(i) &= \text{end}(i-1) + \text{INB}(i) \\
\text{endfor } I
\end{align*}
\]

Thus the neighbors of particle \( i \) are to be found in positions \( \text{start}(i) \) to \( \text{end}(i) \) in the pair list, i.e., \( JNB(\text{start}(i):\text{end}(i)) \).

The critical difficulty with parallelizing this algorithm is the dependence in the \( JNB \) array on previous iterations of the \texttt{for} \( I \) loop. We note that this sort of dependency could not be dealt with by existing compiler techniques. However, the dependency does not occur in the \textit{values} of \( JNB \) but rather in the \textit{locations} of the values. The values of \( JNB \) are independent of each other, but the locations of the values are dependent on other locations, which are computed at the same time as the values in the code above. However we may compute each iteration separately and figure out later where to put
them. By introducing the notion of a "scratch space" JNBL we can easily remove the dependencies as follows.

C Pairlist computation: transformed version
for I = 1, N
    JL = 0
    for J = I+1, N
        if( | X(I) - X(J) | < R ) THEN
            JL = JL + 1
            JNBL(JL,I) = J
        endif
    endfor J
    INB(I) = JL
endfor I
.
.
for IALL = 1, N
    JNB( start(IALL) : end(IALL) ) = JNBL( 1:INB(IALL) , IALL)
endfor IALL

Now each iteration of the I loop is independent of the others and can be computed in parallel. Note that this parallelism has been created at the expense of subsequent communication and at the expense possibly of increased storage. One difficulty in doing this automatically is that there is no a priori bound on the size of the scratch copies.

The above approach translates directly into the model needed in Pfortran. Each processor will compute a collection of the iterations, ISTARTp ≤ I ≤ IENDp. (In the following code, we will assume this information is in the arrays ISTART(p) and IEND(p), known at all processors.) The number may depend on p due to the triangular shape of the double loop [1]. Each processor will have a (local) copy of part of the the JNB array. In fact, the original code can be used, the only exception being that it may be preferable to keep the local INB array distinct.

C Pairlist computation: |Pfortran version
J0 = 0
JJ = 0
for I = ISTART(myProc), IEND(myProc)
    for J = I+1, N
if( \(|X(I) - X(J) | < R\) \) THEN
JJ = JJ + 1
JNB(JJ) = J
endif
endfor J
INB(I) = JJ - J0
J0 = JJ
endfor I

Since \(JNB\) is the largest data structure in the problem, it is likely that it would not ever be collected in one place, but rather it would be left distributed. However, it would be essential for load balancing to know the values of \(INB\), which could be collected in IPfortran as follows.

\[
\text{DO 55512 } p = 0, \text{NPROCS} - 1 \\
55512 \quad \text{INB(ISTART(p):IEND(p)) = INB(ISTART(p):IEND(p)) @ p}
\]

This exchange of information is much less than optimal in terms of communication. An approach that is much better is described in Section 7.4 on the “shake” algorithm. Also see the code in Section 3.3 for concatenating there.

### 7.2 Particle force calculation

The sequential form of the particle force calculation can be written as

\[
\begin{align*}
\text{for } I = 1, N \\
F(I) &= 0.0 \\
\text{for } J = \text{start}(I), \text{end}(I) \\
\text{tmpforce} &= \text{ourforce}(X(I), X(JNB(J))) \\
F(I) &= F(I) + \text{tmpforce} \\
F(JNB(J)) &= F(JNB(J)) - \text{tmpforce} \\
\text{endfor } J \\
\text{endfor } I
\end{align*}
\]

where we are assuming that the \(JNB\) array points only to neighbors with larger particle index, in order to avoid duplicate calculations of \(\text{ourforce}\). Note also that the force \(F\) and the positions \(X\) are vectors in \(\mathbb{R}^3\).

It is easy to see how to load-balance the above. It can be written for \(I = 1, N\)
do $C(end(I) - start(I) + 1)$ units of work
endfor $I$

for some constant $C$. Thus once we know $INB(I) = end(I) - start(I) + 1$ (which may change in an outer loop), we can schedule the separate tasks appropriately. In the

IPfortran implementation of UHgromos [1], all of the $INB$ array is kept at each processor. Each processor then participates in the load balancing step, determining a contiguous set of indices $Istart(myProc):Iend(myProc)$ for which to compute the above. The most time consuming step in the load balancing is the exchange of the $JNB$ array values for these $I$. This will be discussed in a subsequent section. However, once the processor has the relevant data, it simply computes

\begin{verbatim}
DO 1 I=1,N
   F(I)=0.0
1 DO 2 I=Istart(myProc),Iend(myProc)
   DO 2 J=startL(I), endL(I)
      tmpforce = ourforce( X(I) , X(JNB(J)) )
      F(I)=F(I) + tmpforce
  2 F(JNB(J))=F(JNB(J)) - tmpforce
\end{verbatim}

Note that new names ($JNBL$, $startL$, $endL$) are introduced only to clarify the fact that we do not want to store the entire $JNB$ array locally and thus these represent pointers.

Due to the nature of the above algorithm, at the end of the loop, no processor has fully computed $F$. Thus it is necessary to accumulate all the values, which is easily done in IPfortran via the reduction operation

\begin{verbatim}
F = +{F} .
\end{verbatim}

Note that the sequential semantics of the original code have not been kept, since the order of summation may be different. At this point the positions $X$ can be updated via a time-stepping scheme together with an algorithm designed to preserve certain constraints (see Section 7.4).

Note that this implementation is somewhat communication intensive in that the forces are not computed locally. However, this avoids duplicate force computations and is in keeping with the spirit of the sequential algorithm. The experience with the UHgromos code parallelized in this way [1] indicates that the communication overhead is tolerable for modest numbers of processors. However, for large numbers of processors, the cost of collecting $F$ exceeds the cost of computing it in a distributed fashion.
To be more precise, let

\[ \text{INB}_{\text{ave}} := \frac{1}{N} \sum_{i=1}^{N} \text{INB}(i). \]

Then [2] both the computation of \( \text{F} \) and the collection of \( \text{F} \) scale linearly with \( N \), and the ratio of computation cost to communication cost is proportional to

\[ \frac{\text{INB}_{\text{ave}}}{P}. \]

Thus for fixed \( \text{INB}_{\text{ave}} \), the cost of communication will dominate for large \( P \), independent of \( N \). In the experiments with gromos parallelized in this way, for \( \text{INB}_{\text{ave}} \) about one hundred, the cross-over point occurs near 100 processors on the Intel iPSC/860. However, note that \( \text{INB}_{\text{ave}} \) increases cubicly in the cut-off radius, \( R \), so that the break-even number of processors, given by \( P \propto \text{INB}_{\text{ave}} \), increases cubicly with the cut-off radius. In the limit of having an infinite cut-off radius (that is, no cut-off at all), we have \( \text{INB}_{\text{ave}} = N \) and the communication costs are swamped by the computational cost: the algorithm is fully scalable as \( N \) increases.

One way to achieve a scalable code for a fixed cut-off radius appears to be to allow each processor to compute all contributions to part of \( \text{F} \), to eliminate the global reduction operation \( \text{F}=+\{\text{F}\} \). This is done by having \( \text{JNB} \) include all neighbors, not just those with larger particle index. This eliminates the assignment

\[ \text{F}(\text{JNB}(J))=\text{F}(\text{JNB}(J)) - \text{ourforce}( X(I), X(\text{JNB}(J)) ) \]

as well as the communication cost of \( \text{F}=+\{\text{F}\} \), but at the expense of doubling the amount of computation.

Note that the Pfortran implementation mixes the ownership of the force array \( \text{F} \). Any part of the array may be computed by a large number of processors.

### 7.3 Exchange of the JNB array

The exchange of the \( \text{JNB} \) array is described in detail in one case here. The basic algorithm is to line up all the work in a row, divide the length of the row roughly equally, and then push jobs in either direction as necessary to achieve a balance.

```fortran
SUBROUTINE BalJnb(inb,jnb,inbx,jn,nr,segb,myi,myj,njnb)
c for use with NTNi=1
implicit real*8 (a-h,o-z)
integer*4 inb(1),jnb(1),inbx(1),jn(1)
integer*4 segb(2,*),nr
```
integer*4 myi,myj,njnb

integer*4 lj,loweri,upperi,pseg,jindex
integer*4 lis,uis,jta,js,allocate

call inb_exchange(segb,inb,nr,jn)

lj = 0
do 796 i=1,nr
   lj = lj + inb(i)
   inbx(i) = inb(i)
796 continue

do 797 ii = 1,njnb
   jn(ii) = jnb(ii)
c see here
   jnb(ii) = 0
797 continue

lis = 1
upperi = 0
pseg = 1
js = 1
inbeave = 0
allocate = lj/nProc

do 878 i = 1, nProc
   if (inbeave.eq.0) then
      loweri = upperi + 1
   else
      loweri = upperi
      inbx(loweri) = iover
      if (myProc.eq.i-1) inb(loweri)=inbx(loweri)
   endif
   icount = 0
   upperi = loweri - 1
   do while (((icount.lt.allocate).or.
     .  .  .  (i.eq.nProc.and.upperi.ne.nr)).and.upperi.lt.nr)
      upperi = upperi + 1
      icount = icount + inbx(upperi)
   end do
   if (i.eq.nProc) upperi = nr

c count until i's allocation filled for myProc,if myProc

c pseg initialized to 1
   jta = 1
   uis = 0
   lis = loweri
   do while (uis.lt.upperi)
      if (segb(2,pseg).lt.lis) then
         pseg = pseg + 1
         js = 1
      endif
   enddo

c how much of its jnb will processor pseg send to i
7 AN EXTENDED EXAMPLE: PARTICLE DYNAMICS

if (upperi.lt.segb(2,pseg)) then
  uis = upperi
else
  uis = segb(2,pseg)
endif

c count jnb items to be communicated
jnb = 0
do 876 ii=lis,uis
  jnb = jnb + inbx(ii)
876 continue
if (pseg.eq.i-1) then
  do 877 ii = js, js+jnb-1
877 c see here
  if (myProc.eq.i-1) jnb(jta) = jn(ii)
    jta = jta + 1
  continue
  js = js + jnb
else
  jnb(jta:jta+jnb-1)@((i-1) = jn(js:js+jnb-1)@((pseg-1)
    jta = jta + jnb
    js = js + jnb
endif
lis = uis + 1
end do
if ((i-1).eq.myProc) then
  myi = loweri
  myj = upperi
endif
878 continue
RETURN
END

subroutine inb_exchange(segб,инб,nr,tmp)
  implicit real*8 (a-h,o-z)
  include 'gromos.h'
  c
c segб: indices into inб for p of P
c inб: what is being exchange and updated
c nr: number of atoms --> size of inб in words
c tmp: scratch
  c
c logP exchange of inб.
  c
  integer*4 segб(2,*),инб(1),nr,tmp(1)
  integer*4 i,j,k,p,dst,d
  integer*4 map(MAINODES),imap(MAINODES)

do 10 i = 1,nProc
10 map(i) = 0
  map(myProc+i) = 1
  k = i
do 60 d = 1, cubedim
do 50 p = 0,nProc-1
dst = xor(p,k)
tmp(1:nr)@dst = inb(1:nr)@p
imap(1:nProc)@dst = map(1:nProc)@p
if (dst.eq.myProc) then
   do 40 j = 1,nProc
      if (imap(j).eq.1 .and. map(j).eq.0 ) then
         map(j) = 1
      end if
   end do
   do 30 i = segb(1,j),segb(2,j)
      inb(i) = tmp(i)
   end do
end if
70
k = k+2
60 continue
return end

7.4 The “shake” algorithm

The “shake” algorithm utilizes a typical form of relaxation to solve a system of constraints regarding the distance (or angles) between particular particles. Mathematically, there is a system of constraints, $\gamma_i(x_1, \ldots, x_n) = 0$, $1 \leq i \leq k$, on the positions, $x_j$, of particles. Typically, this is extremely sparse, with $\gamma_i$ depending only on a small number of $x_j$. Although the constraints are not linear (for example, they involve the distance between two points which includes expressions quadratic in the coordinates), any given constraint can be satisfied exactly by moving only one particle (with the others fixed). Thus, the shake algorithm iterates on $i$ and moves a particular particle (say, particle number $j_i$) so that $\gamma_i(x_1, \ldots, \hat{x}_{j_i}, \ldots, x_n) = 0$. (Note that the algorithm utilizes the previously computed value, $x_{j_i}$.) Since the previously computed values of $x_j$ are used, this can be viewed as a nonlinear Gauss-Seidel method.

To make the analogy precise, consider a system of linear equations,

$$a_{i1}\xi_1 + \cdots + a_{in}\xi_n - f_i = 0, \quad i = 1, \ldots, n$$

(think $\xi_i = x_{j_i}$ and $\gamma_i \approx a_{i1}\xi_1 + \cdots + a_{in}\xi_n - f_i$). Then the $i$-th step of a Gauss-Seidel iteration is

$$\xi_i \left( \sum_{j \neq i} a_{ij}\xi_j - f_i \right) / a_{ii}.$$
It is well known that the Gauss-Seidel method does not parallelize well due to the fact that each step of the iteration depends on previous ones. A simple solution would be to use instead the Jacobi iteration

$$
\xi_{i}^{\text{new}} = \left( \sum_{j \neq i} a_{ij} \xi_{j}^{\text{old}} - f_{i} \right) / a_{ii}, \; i = 1, \ldots, n.
$$

At the completion of this one assigns $\xi_{i}^{\text{new}} \leftarrow \xi_{i}^{\text{old}}$. This is now $n$ independent tasks, but the Jacobi iteration is (usually) a more slowly convergent algorithm than Gauss-Seidel [5].

A typical solution is to introduce a compound algorithm for parallel computation which involves a Jacobi iteration across processors but interior to each processor, the Gauss-Seidel iteration is used. More precisely, the equations are partitioned into $P$ sets, $\mathcal{I}_{p}, p = 0, \ldots, P - 1$, and the iteration in each processor becomes

$$
\xi_{i} = \left( \sum_{j \neq i} a_{ij} \xi_{j} - f_{i} \right) / a_{ii}, \; i \in \mathcal{I}_{p}.
$$

At the completion of this, all processors exchange values of $\xi_{j}$ as necessary. Suppose that the sets $\mathcal{I}_{p}$ were the ranges $1 \leq j - (pn/P) \leq n/P$ where for simplicity we assume $n$ is an integer multiple of $P$. In Pfortran this can be written

```
much = n/nProc
myslice = myProc*much
C Perform Gauss-Seidel locally
DO 31416 i=myslice+1,myslice+much
   XI(i) = 0.0
   DO 2718 j=1,n
   2718 XI(i) = AMATRIX(i,j)*XI(j)
31416   XI(i) = (XI(i) - F(i))/AMATRIX(i,i)
C Exchange values with other processors (Jacobi update)
DO 911 IP = 0, nProc-1
911   XI( (1+IP*much):((IP+1)*much) ) =
1ti5   XI( (1+IP*much):((IP+1)*much) )@IP
```

Note that the entire XI array is replicated on each processor, necessitating the broadcast of the individual sections to all processors (DO 911 ...). If the AMATRIX is full, such an approach is appropriate. This exchange of XI can be accomplished more efficiently in Pfortran. For clarity, we wrote the exchange step in Pfortran as
7 AN EXTENDED EXAMPLE: PARTICLE DYNAMICS

C Exchange values with other processors (Jacobi update)

DO 911 IP = 0, nProc-1

911 XI( (1+IP*much):((IP+1)*much) ) =

1 XI( (1+IP*much):((IP+1)*much) )@IP

However, this involves $O(n \log P)$ communication, since each broadcast involves $\log P$
steps. The same exchange can be accomplished in $O(n)$ communication via a slightly
more complicated algorithm using divide and conquer [4]. Moreover, it may be nec-
essary to tailor the algorithm for particular architectures. For example, an algorithm
designed for hypercubes is the following. We assume that $P = 2**$CUBEDIM. We also
note that this example assumes that translation of functions of myProc has been
correctly implemented in the compiler. See the code in Section 3.3 (concatenating
ithere) for details.

C Exchange values with other processors (Jacobi update)

mask = 1
muchmo = much

DO 911 idim = 1, CUBEDIM

XI(1+XOR(myProc,mask)*muchmo :

1 (1+XOR(myProc,mask))*muchmo)
2 = XI(1+XOR(myProc,mask)*muchmo :
3 (XOR(myProc,mask)+1)*muchmo )@XOR(myProc,mask)

mask = mask*2

911 muchmo = muchmo*2

Here, XOR is the exclusive-or function. At each step, twice as much is exchanged
with the neighbor in the $idim$-th dimension in a hypercube, so that after $\log(nProc)$
steps, the entire array has been exchanged. We note that this algorithm would execute
correctly on any architecture but there could be potential communication contention
on ones that are not based on a hypercube network.
8 I/O

In this section, parallel I/O in the Pfortran programming environment is introduced. Multiple processors, when accessing the same file, introduce complications beyond the usual language and operating system dependent features of single processor applications. With multiprocessor systems, the options for distributing the data to processors are numerous. Often, answers to the I/O question are not clear, a priori.

A significant complicating factor is that I/O system of a given computer model can vary dramatically depending on the intended use. One system may have only one disk subsystem attached to a single processor, whereas the same model from the same company could have the same disk subsystem attached to every processor at another site. Thus the I/O system cannot be modeled in a simple way that is universal for all computers of a given type.

In order to find a suitable I/O strategy, both the application and the computer architecture must be taken into account. Numerous questions can be asked. Will the application tolerate serial access to the file, or does that result in an unreasonably high IO-to-computation ratio? A 128 node computation, effectively parallelized and with a large data set, might require a more sophisticated I/O strategy. If so, how will the disk subsystem perform with simultaneous accesses to the same file? Should one perform concurrent I/O from the same file, or should a single processor access files, using the communication network to distribute the data to other processors? The use of the communication network to distribute the data from a single process might be the preferred method. For outputting, is post sorting of data feasible, or can the application tolerate seeks to the same file by multiple processors in order to write the data in place, or should a single processor collect the data and write to the file? Most I/O questions can be answered using insight about the application and architecture, along with simple experimentation.

In the remainder of this section, three parallel I/O strategies will be discussed within the Pfortran program model. Pfortran assumes a separate namespace and a separate address space at every process, requiring that data be explicitly assigned to all nodes. We will examine, with examples, the parallel I/O methods alluded to above:

- a single node reading the data from disk and communicating that data to the other nodes,
- all nodes directly reading data from disk,
• all nodes reading data concurrently.

In each case, the Pfortran program model, very close in form to "node code," requires that data be distributed explicitly as a function of the logical processor identifier, myProc. The examples illustrate P processors, reading from the file, coord, into the array, X, with the N data accessed, later in the calculation, by a blockwise distribution. The time for I/O to be performed from a single process is written as $T_s$.

### 8.1 Serial I/O from Disk

This method serializes multiprocess access to the file. Processes can use direct access methods to read their assigned data, or they can read through the entire file, discarding unnecessary data. This method can be unacceptably slow for many process applications. In the sample code section that follows, the serial I/O requires time, $P \times T_s$, and space, N/P.

```fortran
  do p = 0, nProc-1
    if (p.EQ.myProc) then
      OpenFile(coord,unit)
    c*
    c* skip data
    c*
      do b = 0, myProc-1
          READ(unit,* ) (X(i),i=1,N/P)
      enddo
    c*
    c* get data block for calculation
    c*
      READ(unit,* ) (X(i),i=1,N/P)
    c*
    c* Each file closes its unit, conserving file
c* descriptors, a finite system resource.
      CLOSE(unit)
    endif
    c*
    c* This serializes the disk file access. See example below.
c*
      call synchronize
```
enddo

The synchronize subroutine may be a call to a machine dependent synchronization routine, for example, the Intel iPSC subroutine, gsync(). Alternatively, a variation of the baton passing algorithm from Section 6 can be used. A Pfortran synchronization algorithm is given below. This algorithm guarantees that processes will not proceed beyond the call until all other processes have entered this point.

```fortran
subroutine mysync()
    integer p,src,dst,baton
    do src=0,2*(nProc-1)
        dst = mod(src+1,nProc)
        baton@dst = baton@src
    enddo
    return
end
```

A more efficient algorithm uses a dimensional exchange.

```fortran
subroutine mysync2()
    integer baton(1)
    baton = +{baton}
    return
end
```

8.2 Single Process I/O from Disk

This method assigns the I/O task to a single process. The process used should be either process 0, or nProc-1, in order that the number of processors can vary and have the code correct, even for one node. The single process will read the data from disk and distribute it, using the computer’s communication network. Ideally, data will be distributed in blocks, to reduce latency, and those blocks will be distributed as they are read from disk, to reduce the memory requirements of the I/O process.

The following code segment performs I/O in space, N/P, and time

\[ T_s + \alpha(N - N/P) + \beta(P - 1), \]

where \( \alpha \) and \( \beta \) are machine-dependent constants for throughput and latency, respectively.
if (myProc.EQ.nProc-1) OpenFile(coord,unit)
do p = 0, nProc-1
   if (myProc.EQ.nProc-1) READ(unit,*) (X(i),i=1,N/P)
   X(1:N/P)@p = X(1:N/P)@(nProc-1)
endo
if (p.EQ.myProc) CLOSE(unit)

8.3 Concurrent I/O

With concurrent I/O, processes read data simultaneously from disk. Intel provides support for this via their Concurrent File System. At a first glance, this technique appears to be the preferable method, especially if there are multiple disks attached to multiple processors with files striped across these disks. However, disk contention could make this method inferior to single process I/O. The following code fragment illustrating concurrent I/O requires space, N/P, and time, \( AT_s \), where \( A \) is some machine and application dependent parameter, and file positioning is assumed to be free.

call OpenFile(unit,coord)
call PositionFile(unit,(N/P)*myProc-1)
READ(unit,*) (X(i), i=1,N/P)
CLOSE(unit)

8.4 Intermediate I/O

By intermediate I/O, we mean the bits of I/O from occasional print statements, usually of a non-critical, diagnostic flavor. This is I/O that takes place after program initialization, but before termination where large amounts of I/O may be required. It could be be the outputting of a single energy value or an entire configuration, in the case of particle dynamics. Intermediate I/O should be blocked so that I/O buffers are well utilized and latency effects are reduced.
9 Typical Programming Errors in IPfortran

To insure correct communication using a IPfortran statement, the communicating processes must both execute the communication statement, agreeing on the source and destination values. Furthermore, any processes participating in any communication must execute all communication statements in order to maintain consistency with message tags.

9.1 Conditional guards

One source of difficulty with any parallel programming system concerns communication statements that occur inside blocks that are “guarded” by conditional expressions. For example, the statement

\[
\text{IF( myProc .EQ. 7 ) x = y@11}
\]

is not the same as

\[
x@7 = y@11
\]

in IPfortran. The latter correctly assigns the value of the variable \( y \) in processor 11 to the variable \( x \) in processor 7, whereas the former will simply cause processor 7 to deadlock waiting for processor 11 to execute the same statement, which it will never do. Variants of the former statement can be correct, e.g.,

\[
\text{IF( myProc .EQ. 7 .OR. myProc .EQ. 11 ) x = y@11}
\]

will set \( x \) at both processors 7 and 11 to the value of \( y \) at 11. However, the latter programming style in IPfortran is discouraged. Rather, the expression \( x@7 = y@11 \) is preferred.

9.2 Inconsistent source and destination process identifiers

In the current compiler implementation, a faulty communication expression results for \( a@p = b@q \) if the values of \( p \) and \( q \) do not agree at processors \( p \) and \( q \), that is if

\[
p@p \neq p@q \quad \text{or} \quad q@p \neq q@q.
\]

See Section 3.1 for a work-around.
9.3 Processor number out of range

A statement such as

\[ x = y^p \]

is incorrect (with unpredictable results) unless

\[ 0 \leq p < \text{nProc}. \]

See Section 6.3 for examples of how to limit processor values appropriately.
10 Users guide

10.1 Installing

The Pfortran default target machine can be hard-wired at compiler time with the Makefile.pf variable TARGET where valid machine types may be taken from: IPSC, NCUBE, CM5, PVM3, KSR, and SGI. The default machine can be overridden with compiler switches (see below).

10.2 Invoking

The Pfortran translator pfc can be invoked with

\[ \text{pfc} \ [\text{switches}] \ f_1.pf \ [f_2.pf \ [\ldots]] \]

or alternatively

\[ \text{pfc} \ [\text{switches}] \ *.pf \]

where:

- \( f_d \) Name of the Pfortran (or FORTRAN77) source file; this filename must have .pf as a suffix.
- switches Optional command line switches listed below.

10.3 Include files

Optional include file search directories can be used with the syntax:

\[ \$\text{pfc} \ -\text{Id1} \ -\text{Id2} \ldots \ -\text{Idn} \ \text{myprog.pf} \]

for directories d1 ... dn.

Up to 10 directories can be specified with one directory per I flag. The directories are searched in order of occurrence; the current working directory is always searched and it is searched last unless -I. is the first include search directory. The maximum nesting depth of include files is 10.
10.4 Switch summary

The following switches select the indicated target system for code generation:

-ips  Intel iPSC/860, Touchstone Delta, or Paragon.
-kwr KSR-1; uses TCGMSG.
-sgi SGI; uses SHMSG.
-cm CM5.
-pvm PVM.
-ncube NCUBE.

The following switches are architecture independent.

-I Search this directory for include files.
-scratch integer Dimension for Pfortran buffer (default 100000).
-T string Use string as variable name for myProc (default).
-i integer Make the default integer size integer-bytes.
-rp integer Make the default real size integer-bytes.
-dp integer Make the default double precision size integer-bytes.

10.5 Intrinsic names

In addition to the intrinsic functions used by the target processor, Pfortran has a reserved variable-name set.

myProc  Processor number of the processor running the code.
THISPROC Process number of the process running the code.
nProc  Total number of processors.

The current implementation of the compiler also imposes some additional reservations, although these may be cancelled in a later version.
pflib  Pfortran library module.
pf_ib  Integer buffer.
pf_sb  Integer2 buffer.
pf_rb  Real buffer.
pf_db  Real8 buffer.
pf_cb  Character buffer.
pf_xb  Complex buffer.
pf_inctag  Tag increment subroutine (pflib).
pf_snd  Send subroutine (pflib).
pf_rcv  Receive subroutine (pflib).
iqdim  Cube dimension subroutine interface.

10.6  Pfortran Library

System-dependent sends and receives are made from the Pfortran library, called pflib. There are several libraries supported by this version:

pflibintel  Intel hypercube or simulator, Touchstone Delta, and Paragon.
pflibksr  KSR-1 TCGMSG.
pflibshm  KSR-1 shared-memory message passing.
pflibsni  SGI multiprocessor shared-memory message passing.

Consult your local pfc man page for details on the naming conventions and library locations.

The Pfortran library, or its equivalent, must be linked to (loaded with) a program translated by the Pfortran translator.
10.7 Scratch memory and variables

Temporary memory for arrays uses a block of storage declared with the array `pf_lib`. The default size of this storage can be modified using the `-scratch` switch. This storage is re-typed as needed through the use of equivalence statements.

Pfortran generates scalar variables for buffering or naming quantities. These variables are named from the base string `o10000` with successive variables incrementing the suffix `10000`. Clashes can occur if these variable names are used as implicitly typed variables in your program.

Labels are also generated by Pfortran in generating FORTRAN77 loops. These are guaranteed to be unique to the program section.

10.8 Using Pfortran

Pfortran translates the input `.pf`, producing output into a `.f` file. Code not involving Pfortran extensions is output "as is," including comments and blank lines. Code generated from translating statements involving `{}` or `@` adheres to the rules of FORTRAN77.

There are three levels of advisory messages with Pfortran: fatal, error and warning. *Fatalds* result in immediate termination of the translator with a diagnostic message and `.f` file removal. *Errors* also result in `.f` file removal, but the translator will continue processing the input. *Warnings* result in advisorys alone with output written as usual to the `.f` file.

We emphasize that the intrinsic variable set for Pfortran includes variables such as `myProc`. These can be used freely in any subroutine or function that has been processed by Pfortran – redeclaration of these intrinsic variables will result in error. Pfortran can be used on one, or all, of the modules composing a program with the exception of the entry point to the Fortran program: *The main program (entry point to a program) must be processed by the Pfortran processor.*
10.9 Sample Makefile for the iPSC/860

Having added another file-type suffix slightly complicates the compilation process. The make program, however, can be used to automatically invoke the correct sequence of compilations. With the following Makefile, the make program will automatically check dates for the file sequence `planet.pf`, `planet.f`, `planet.o` and the executable `planet`, and recompile as necessary.

sample makefile:

```
PFC = pfc
F77 = if77
PFLAGS = -scratch 1000000
FFLAGS = -02 -node
LFLAGS = -node
LIBS = -lpf
.SUFFIXES: .pf .f .s .o
.f.o: $(F77) $(FFLAGS) -c $*.f
.pf.f: $(PFC) $(PFLAGS) $*.pf
.pf.o: $(PFC) $(PFLAGS) $*.pf
$(F77) $(FFLAGS) -c $*.f

planet: $$@.o
$(F77) $(LFLAGS) -o $$ $$@.o $(LIBS)
```

10.10 IFortran Programs and the KSR

The KSR language and operating system directly support parallelism in threads executing within a shared address space [9]. The separate address space feature of the SPMD model can be achieved with a multiprocess implementation of the application communicating through shared-memory. Currently IFortran uses two such systems: TCGMSG [7] and SHMSG (see below).

10.10.1 IFortran and TCGMSG

To run a compiled program on the KSR, the program is passed as an argument to the TCGMSG program `parallel`. Suppose we want to run the program `baton` from the current working directory. In this directory, the file `baton.p` must be present, or created, prior to issuing
$ parallel baton

The file baton.p contains:

# This template is piped through sed to replace
# a with the name of the program.
# Add other hosts and processes as desired.
clark bubba 2 baton /home/clark/cwd

giving the user name, the host name, the number of processors, the executable name and the directory from which to run the program. Be sure to check the local system policies for processor set allocation.

A checklist for making the KSR executable with Pfortran are:

- Use the -ksr flag with Pfortran translator,
- Use the KSR version of pflib.f (or libpf.a),
- Compile with the -para and -i4 flags,
- ld with the libraries -ltcmsg and -lrpc.

A sample Makefile, for the Pfortran program baton, follows.

LIBDIR = /usr/local/lib
TCGMSG = $(LIBDIR)/libtcmsg.a
LIBS = -lrpc -lpthreads
FFLAGS = -i4
LFLAGS = -para
PFFLAGS = -ksr
F77 = f77
PFC = /usr/local/bin/pfc

.SUFFIXES: .pf .f .pc .c .s .o
.pf.o: $(PFC) $(PFFLAGS) *.pf
   $(F77) $(FFLAGS) -c *.pf
.f.o: $(F77) $(FFLAGS) -c *.f
.pf.f: $(PFC) $(FFLAGS) *.pf

baton: baton.o pflib.o
   $(F77) $(LFLAGS) -o baton baton.o pflib.o $(LIBS) $(TCGMSG)
Note that the -i4 switch is critical. If you find that using 4-byte integers on the KSR limits your applications, please submit your comments to the system administrator. There are working examples on the TCAMC KSR at /usr/local/src/pfl.x/suite. A common mistake in using the KSR Fortran is to execute the application without the parallel command described above. A Fortran program will execute apart from parallel, but not correctly.

10.10.2 Fortran and SHMSG

The SHMSG system is a stand-alone library of message-passing routines for shared-memory architectures. Internally, it is similar to TCGMSG, however, there are differences in the user interface. The usage described here applies to the KSR-1 and SGI multiprocessors.

10.10.3 Compiling programs for SHMSG

A sample Makefile follows, however you are encouraged to check your local machine’s man page for the location of test programs and Makefiles.

INC = .
LIB =
CC = cc
F77 = f77
SHMSG = /home1/clark/lib/shmsg.a
LIBS = pflibshm.o $(SHMSG)

CFLAGS =
FFLAGS = -r8 -i4
LFLAGS = -para

.SUFFIXES: .pf .f .pc .c .s .o
c.o:;$(CC) $(CFLAGS) -c $*.c
.f.o:;$(F77) $(FFLAGS) -c $*.f
.pf.f:; $(PFC) $(PFLAGS) $*.pf
.pc.c:; $(PC) $(CFLAGS) $*.c
.pf.o:; $(PFC) $(PFLAGS) $*.pf
$(F77) -I$(INC) $(FFLAGS) -c $*.f
PFC = pfc

# note that -kser will also interface with SHMSG
PFLAGS = -shm -Tme

pivot: pivot.o
    $(F77) $(FLAGS) -o @@ @.o $(LIBS)

10.10.4 Specifying processor configurations for SHMSG based programs

A configuration consists of the number of processes (P) and the number of cells (C)
on which the processes will execute. Processes will be bound to some cell. If P > C,processes will be cell mapped many-to-one.

A third parameter here is the number of cells in your pset (actual.C), or the “machine”that your application sees. If C > actual.C, SHMSG will adjust and informyou that intra-application pset timesharing will occur.

Intra-application timesharing is not recommended at this time.

Applications can have problems with intra-application pset timesharing, especiallywhen P > 2C. We have found that either it works, or it doesn’t. We discourage thisusage.

The configuration, then, consists of two integers contained in a file called Config(upper case 'C'). This file must be in the directory where program executes. Thevalues of P and C are integers on the first line of this file, e.g., a first record of “8 8”specifies 8 processes to be executed on 8 processors.

Processor configurations should be powers of two for Pfortran comaptibility. Thatlimitation will disappear with an expanded Pfortran communication library. Non-power of two processor operations will work with SHMSG in any case, and withPfortran provided global exchanges are not used there.

10.10.5 Running SHMSG based programs

Type the program name in the directory where the Config file resides. Input andoutput behave as usual.

10.10.6 Stopping SHMSG based programs

So that your application releases system resources, a cleanup routine must be invokedprior to normal termination. The Pfortran compiler will translate a Fortran stop toa pfstop which in turn invokes the routines to release resources. A Fortran programshould terminate with a Fortran stop.
For interactive processes, use `ctl-c`. If the process is not attached to a terminal, send it the SIGINT signal, i.e. `kill -2 <pid>`. If all else fails, use `kill -9`, with the disadvantage that the processes cannot release system resources when hit with the SIGKILL signal.

### 10.10.7 Things to look out for

**Hung program:** If your program hung, or abnormally terminated, take a look at system ipc resources using the command “ipcs.” If this seems like too much trouble, just run bubba: `~clark/bin/rmem`. This will remove the shared-memory segments that the message-passing system used, and did not release, due to drastic termination, e.g. `kill -9`.

**Intra-application pset timesharing:** When an application’s processes timeshare cells, a less efficient algorithm is used for coordinating access to shared data in the message-passing system. This stems from a bug in a hardware mechanism used for enforcing mutual exclusion.

### 10.11 KSR specific limitations

Using `P`fortran on the KSR results in efficient, and portable programs. There are, however, some limitations that you should be aware of. If any of these limitations impede your work, then again, submit comments to the system administrator; software solutions are available for all of the limitations listed below.

- Using `IMPLICIT NONE` will result in compiler warnings as the KSR switch does not produce declarations for some intrinsic KSR functions. Workaround: put the declarations in the `P`fortran source.

- Global sum algorithms require power of two processors, e.g. 1,2,4, ... . Using other than powers of two processors will result in erroneous results for `P`fortran global sum operations. Workaround: use powers of two total number of processors, or do not use the `P`fortran intrinsic global sum operations.

- The KSR debugger is not operational for `P`fortran programs. Workaround: Use print statements or debug on a Intel hypercube.
11 Current limitations and bugs

This section describes issues in addition to those described in section 1.2.

11.1 Communicating with nonexistent processors

If the value of an expression following an @ sign is not in the range $0, \ldots, nProc - 1$, deadlock results.

11.2 Communicating character strings

The communication of character strings is not supported. Character arrays, on the other hand, are supported to the extent that other types, e.g. reals, integers, etc., are.

11.3 Arrays as expression operands

Operations on entire arrays or array subsections are limited to simple assignments of the type: $A = A@p$ or $A = +\{A\}$. Arithmetic operations on entire arrays or array subsections, e.g. $A = A@p + B$, will result in a core dump.

11.4 Reducing functions as operands in expressions

As with arrays or array subsections, global functions are only permitted in simple assignments, regardless of their type.

11.5 Reducing functions and arrays

Reducing functions are supported with scalar arguments only since we have not incorporated a typing mechanism for functions. For the adventurous user, the syntax 

```latex
out = op \{ \text{in} \}
```

where `in, out` are arrays, is translated into a subroutine call `call op(in, out)` where `out` is both an input and output parameter.
11.6 Reduction operations and processor subranges

At present, reduction operations are only defined on the entire set of processors: processor subranges are not permitted for selecting a subset of processors participating in a global operation. It is valid, however to, specify a processor subrange for the target. Nonetheless, all processors will participate in the reduction.

11.7 Overlapping of communication and computation

Code movement of sends (to be done as soon as possible) and receives (to be postponed until as late as possible) based on the definition and use of variables involved is not yet implemented.
12 IPfortran extensions to FORTRAN77 grammar

The operator precedence in IPfortran, from highest to lowest precedence, is

\[ @ \quad ** \quad (\ast,/) \quad (+,-) \]

So for example, \( a = b@p + 1 \) is not equivalent to \( a = b@((p + 1) \) and \( a@p + 1 = b@q \)

is meaningless where \( a@((p + 1) = b@q \) is not.

\( @ \) is non-associative, therefore, \( x = y@i@p \) is illegal and \( x = y@((i@p) \) or \( x = (y@i)@p \)

are legal.

The following productions summarize the IPfortran extensions to the FORTRAN77 grammar:

\[
\text{expr} \rightarrow \text{expr} \@ \text{expr} \mid \text{expr} \@ \text{subs_list} \mid \text{expr} \@ \text{idx_list}
\]

\[
\text{expr} \rightarrow \text{name} ( \text{subs_list} )
\]

\[
\text{subs_list} \rightarrow \text{expr} : \text{expr} \mid \text{subs_list} , \text{expr} : \text{expr}
\]

\[
\text{expr} \rightarrow \text{name} \{ \text{arglist} \}
\]

\[
\text{statement} \rightarrow \text{call} \ \text{name} \ \{ \ \text{arglist} \}
\]

where \( \{\} @, ; () \) and \( \text{call} \) are literals.

The second rule corresponds to subarrays; the third rule covers both subarrays and processor ranges. The fourth and fifth rules represents global functions and procedures, respectively.
13 Glossary

@ The “at” sign; a binary, in-fix operator specifying the processor location of a variable. Read $x@p$ as variable $x$ at processor $p$.

{} Curly braces enclose the parameters for a reduction operator e.g., $mars\{p1\}$. We say that function (or subroutine) $mars$ is globalized in this context.

: The colon is used in constructing a processor range or subarray (see grammar).

**actual parameter** The variables used in a function or subroutine invocation (see formal parameter).

**asynchronous execution** Processes proceed without regard for the state of other process(es). A Pfortran process proceeds asynchronously between receives. (Asynchronous receives are available, on the Intel hypercube, e.g., however, these are not used in Pfortran.)

**asynchronous communication** A send is asynchronous if a process proceeds while the message is transferred to the destination process. A receive is asynchronous if after posting a receive without an incoming message, the process continues to execute.

**array-element order** The sequence of elements derived from an array by running through the innermost subscripts fastest.

**array shape** A rank 1 array where each integer element specifies the extent of that dimension.

**array subscript** An array element is specified with a list of subscripts, one for each dimension of the array (together called an array selector). An array subscript may be a range or a scalar.

**chaotic environment** External effects can arrive at arbitrary intervals. Behavior can not be guaranteed to be repeatable and may depend on arbitrary factors beyond the programmer’s control. Pfortran uses the SPMD model where processes exercise influence over one another by mutual execution of message-passing statements; we call this environment deterministic.

**conformable (arrays)** Two arrays are conformable if they have the same shape.

**deadlock** A deadlock occurs in a system when that system can no longer change its state. In order for deadlock to occur all of the following conditions must hold: at least one resource is held in a non-sharable mode (mutual exclusion), one process holds one resource and is waiting to acquire other resources held by other processes (hold and wait), a resource can only be released voluntarily (no preemption), and there exists a circular wait. Consider two processes, A and B. If A waits for a message from B,
while B waits for a message from A, the four necessary conditions for deadlock are satisfied.

**deterministic** A process is said to be deterministic if its behavior is repeatable. Fortran computation has been devised to be deterministic. For the contrary, see *chaotic.*

**dimensional exchange** Simultaneous, bi-directional exchanges along all communication links (dimensions) in a hypercube permit $\log_2 P$ algorithms for many serial algorithms. In the $\log_2 P$ steps, data from any one node will have reached all other nodes (also referred to as *global exchange*).

**extent** Number of elements (for an array dimension).

**formal parameters** The “dummy parameters” used in a subroutine or function definition (see *actual parameters*).

**FORTRAN77** ANSI X3.9-1978 Fortran programming language. (The language without a *do while* loop.)

**Fortran90** X3J3 Technical Committee X3.9-198x Fortran Draft.

**global value** For any variable in the SPMD model, there will exist $P$ processor instances of that variable. If assignments to that variable have not involved *myProc* or functions of *myProc*, we say that the value of that variable in any processor is a global value. If, on the other hand, the converse applies at some processor, we say that the value of the variable is a *local value*.

**hypercube** Refers to a processor interconnect scheme where for a $P$ processor computer, each processor will have $\log_2 P$ connections to neighboring processors; these neighbors will be all processors at a Hamming distance of 1 (where we have assigned a logical numbering scheme $0 \ldots P - 1$ to the set of processors).

**local value** See *global value*.

**message-passing** A method used by processes to both communicate data and synchronize on that data. Communication is accomplished because a *receiving* process receives values sent from the *sending* process. Synchronization occurs because a message can be received only after it has been sent.

**node program** The instance of an SPMD program in execution with the value of *myProc* distinguished from other node programs in the set of *nProc* processes.

**procedure** A Fortran subroutine or function.

**process** A program in execution.

**rank** The number of dimensions of an array.

**range** An interval of values on the integers where the notation $(i : j)$ denotes the sequence $i, i+1, \ldots, j-1, j$.

**reduction operation** The process by which a set of values corresponding to a
distributed object (an array or scalar) are reduced to a single instance of that object through the repeated application of a binary operator.

**send/receive**  See *message-passing*.

**synchronous communication**  When both the send and receive are blocking, communication is synchronous (see *asynchronous communication*).

**subarray**  A rectilinear section of an array.

**SPMD**  Single Program, Multiple Data.

**synchronous**  See *asynchronous execution* and *asynchronous communication*.

**myProc**  The unique, logical processor number for a process.
References


