IPfortran: a parallel dialect of Fortran

(Appears in ACM's FORTRAN FORUM, Sept. 1992)

Babak Bagheri
Department of Computer Science, Pennsylvania State University
Terry W. Clark
Department of Computer Science, University of Houston
L. Ridgway Scott
Department of Mathematics, University of Houston, Houston, TX 77204

1 Introduction

IPfortran is a language designed to facilitate the programming of multi-process, data-parallel applications [BCS91]. Based on a message-passing paradigm, IPfortran extends Fortran with a small set of extensions. The duality of the send and receive operations is encapsulated in an infix operator (Section 2) or reducing functions (Section 3). With *system-dependent* message-passing pushed out of sight, code is streamlined and development time reduced. Errors in writing message-passing code are reduced by leaving the generation of the message-passing logic to the translator.

IPfortran 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 IPfortran 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.

Using the *local* approach, IPfortran programs are written with explicit logic for each computational element with data decompositions propagated by programmer-supplied control structures. This methodology, while crude relative to globally-viewed, automatic decompositions [CHKKS92], can provide greater flexibility. Moreover, industrial-strength codes have been ported to IPfortran in a short amount of time [CMS91].

With the IPfortran programming model, the logical processor number pinpoints the data item in an expression through the use of the @ (Section 2) where $i@p$ represents a different variable from $i@q$, if $p \neq q$.

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 *ThisNode*, the logical processor number. Any logic in the program involving the variable *ThisNode* 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 *ThisNode* and using the the SPMD model on identical processors, we would have rather uninteresting computations.

As a consequence of the logical processor number *ThisNode*, the values, corresponding to a variable, at each processor 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 using the current version of IPfortran, the programmer must keep track of which values are *global* and which are *local.*

Some IPfortran constants inspired by commercial hypercubes and used throughout this article are:

ThisNode The Processor number of the processor running the code; also called *TN.*

NumNode The total number of processors; also refered to as *P.*

We make the convention that the processors are identified by consecutive integers, starting at *0* and continuing to *NumNode-1.* This numbering will be referred to as the logical numbering.

2 The Operator @

Accessing variables at other processors can be done in IPfortran 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 3). (@, in principle, makes sense in any expression, e.g. predicates, but has not been implemented there.)

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

$$@ \qquad ** \qquad (*, /) \qquad (+, -).$$

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 example of a code to compute the sum of values contained in each processor, say in the variable y. All processors broadcast their own value and each processor sums the values independently.

```
      DO 10 n=0, NumNode-1
10       x(n) = y @ n
      sum=0.0
      DO 20 n=0, NumNode-1
20       sum = sum + x(n)
```

This technique may not be very efficient, but it serves to illustrate an important point concerning our assumption about communication in IPfortran. 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 in each processor. Thus n denotes a value universally agreed upon at each iteration, a *global* value. If the value of n were different in different processors, then a *local* value results and a more complicated implementation would be needed.

If values refering to processor numbers are *global,* we can predict the communication pattern required for the data exchange. Otherwise, for processor numbers given by *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 IPfortran we plan to do compile-time analysis to determine which variables referring to processor numbers are *local* or *global.* At the moment, the programmer should limit the use of variable processor numbers to *global* values. Our experience has shown that this is not a significant hindrance and results in a distinctive programming style.

2.1 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*> @ (<*expr*>$_1$: <*expr*>$_2$)

refers to the value of <expr> in the range of processors from <expr>$_1$ to <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

$$x@(i:j) = y@(r:s),$$

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:

$$a@(i,j,k) = b@(m,n,o)$$
$$a@(1,2,3,4) = b@0$$
$$a@(k+m:n) = b@0$$

2.2 Functions of *ThisNode*

It is tempting to write $x@ThisNode = y@(ThisNode-1)$, however, this is incorrect in the current version of IPfortran and will result in deadlock. It is not unreasonable to expect the translator to detect the presence of *ThisNode* and generate correct code. The remainder of this section talks about the first case, detecting the direct presence of *ThisNode* in a communication statement. This has been implemented in IPfortran's cousin, PC [BRS90].

Using standard compiler algorithms, analysis can detect the dependence of a variable's value on *ThisNode.*

The expression

$$x = y@mod((ThisNode+1), NumNode)$$

means: assign the value of variable y of processor *(ThisNode+1)* (modulo *NumNode*) to x in *ThisNode.* We can have any expression which mentions *ThisNode* explicitly (including functions which return integers, arithmetic operators, etc.).

An example using functions of ThisNode 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:

```
      sum=y
      DO 10 n=1, NumNode - 1
        sum = sum@mod((ThisNode+1), NumNode)
 10     sum = sum + y
```

This has the effect of computing the n-th partial

$$(\ldots(y@me + y@(me+1)) + \cdots + y@(me+n+1))$$

at the n-th iteration, where node indices in the latter representation must be identified modulo *NumNode.*

3 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 like x@(1:5) refers to many variables (the current implementation of IPfortran 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 replicated 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.

3.1 Reduction operations defined

To allow such operations, IPfortran introduces the following constructs:

<id> =<id> { <expr> [,opt args]}
call <id> { <expr> , <id> [,opt args]}

where <expr> can reduce to a variable or constant.

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 <expr> . 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 IPfortran statements, Fortran90 array syntax may be used (see Section 4. Arrays).

We symbolically write the reduction operation for a p-process model

$$A = \oplus\{B\}$$

or

$$call \quad \oplus\{B, A\}$$

as

$$A = B_1 \quad \oplus \quad B_2 \quad \oplus \quad \cdots \quad \oplus \quad B_p.$$

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 $\oplus$.)

3.2 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. We begin with intrinsic Fortran functions. The statement

y = AMIN1 { x }

computes the minimum of all values of x in all processors and places it in the variable y (in all processors). Functions and subroutines used along with {} are called *reducing functions* and *reducing subroutines*, respectively.

The user-defined reduction operations may include optional parameters. These parameters are local parameters to a processor, i.e. they are not communicated, but are included in the calling sequence as shown above. These parameters can carry type information for the parameters `in` and `out`, or can be used in generating side-effects with the reduction operation. (Detailed semantics for the reducing function and subroutine interface can be found in the *IPfortran Reference Manual* [BCS91,IPfortran].)

Now consider a user defined reduction operation implementing a summation of a set of scalars distributed across p processors into a single scalar. Mathematically, we wish to implement

$$\forall j, \quad result@j = \sum_{i=1}^{p} scalar@i.$$

By defining a function, `mysum`,

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

we can write

result = mysum{scalar}.

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

```
call opuratur { in,out,local1,local2 }
```

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

SUBROUTINE opuratur(in1,in2,out,local1,local2)

where *local1* and *local2* are optional parameters.

3.3 Intrinsic reduction operations

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 IPfortran we can write

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

Alternatively, we write the more concise expression using the IPfortran intrinsic reduction operator, $+\{\}$:

$$\mathrm{A} = +\{\mathrm{A}\}$$

Note that the result of the user-defined reduction

$$\mathrm{result} = \mathrm{mysum}\{\mathrm{scalar}\}$$

described earlier can be achieved using the intrinsic

result $= +\{$scalar$\}$ just as easily. Further reduction operation applications are discussed in the *IPfortran Reference Manual.*

4 Arrays

In a 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 extends, albeit modestly, FORTRAN77 with Fortran90 syntax. (Unless otherwise stated, the array terminology used here follows that recommended by the X3J3 Fortran90 committee.)

IPfortran permits arrays to be referenced in simple assignments of the type

$$A(m:n) = B(m:n)@p.$$

The array extensions are not recognized outside of expressions involving @ or reducing function references.

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 an expression, or as an *actual parameter* to a subroutine or function, where @ or {} are used.

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 IPfortran. 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 $DIMENSION \quad A(3,3)$. 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 $/1,2,4,5,7,8/$. IPfortran assumes that the procedures being reduced via {} are independent of such index shifts.

The use of array sections may add a copy overhead to the compiled IPfortran 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 IPfortran translator.

5 Sample programs

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

5.1 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 runners 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.

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

5.2 Column oriented LU decomposition

This example illustrates prudent application of cyclic assignment for an LU decomposition algorithm. Geist and Romine [GR88] describe the following algorithm for column-to-processor assignment of a matrix using *LU decomposition* on an $n \times n$ matrix.

```
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
```

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 $p = mod(column_j, numnode)$ and $0 \leq column_j < n$. An alternative decomposition, *contiguous assignment*, leads to increased processor idling since the lower indexed columns are eliminated first.

In the IPfortran *LU decomposition* example that follows, both the distribution of, and the computation with data 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 (TN.eq.mod(i,numnode)
.        call getcolumn(i,a)
   enddo
```

```
   do k = 1, N-1
C owner computes column of multipliers
      if (TN.eq.mod(k,numnode)) 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, numnode)
C reduce columns
      do i = k+1,N
        if (mod(i,numnode).eq.TN) 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
```

5.3 Dimensional exchange

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

```
    real*4 o10000
    real*4 o10001
    integer*4 o10002
    integer*4 o10003
Cpf     a = op{b}
    a=b
    o10002=1
    do 10 o10003 = 1, cubedim
       call pf_inctag()
       call pf_snd(xor(thisnode,o10002),a,4)
       call pf_rcv(o10000,4)
       a=op(o10000,a)
       o10002=o10002 * 2
 10 continue
```

where $cubedim = log_2 P$. 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,numnode-1
         dst = xor(p,k)
         tmp@dst = a@p
         if (dst.eq.thisnode) then
           a = op(tmp,a)
         endif
50      continue
        k = k * 2
60   continue
```

Another loop is introduced in this algorithm adding an $\mathcal{O}(P)$ 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. IPfortran 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 IPfortran accomplishes this, but with all processors participating in the exchange. While we intend eventually to produce code for syntax such as `a=op{b@(i:j)}` in IPfortran, to communicate efficiently among an arbitrary subset of processes, for now only $a = opb$ is supported.

Using IPfortran *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.

```
if (A(thisnode).ne.1) then
  contribute = 0.E0
else
  contribute = value
endif
accumulator = +{contribute}
```

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

If the IPfortran 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 IPfortran statements.

5.4 Dining philosophers problem

With this example we veer from our numerical-applications trajectory and apply IPfortran to a classic problem in concurrency control. Our objective is not to advocate IPfortran for this type of parallel programming problem. Instead we include this example to show how the inner workings of IPfortran 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 IPfortran 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 [DIJ65]. 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 programming 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 instance. In our implementation of IPfortran 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 deadlock and progress have been tested against the dining philosophers problem – monitors, semaphores, Hoare's CSP, to name a few. We present a IPfortran 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 respective 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 philosophers, 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 IPfortran 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

message@bamboo = activity@human
message@human = reply@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? IPfortran generates the following code for the two statements in the philosopher subroutine

```
Cpf message@bamboo = activity@human
 call pf_inctag()
 if (TN.eq.bamboo.and.TN.eq.human) then
    message=activity
 else if (human.eq.TN) then
    call pf_snd(bamboo,activity,4)
 else if (bamboo.eq.TN) then
    call pf_rcv(message,4)
 endif
Cpf message@human = reply@bamboo
 call pf_inctag()
 if (TN.eq.human.and.TN.eq.bamboo) then
     message=reply
 else if (bamboo.eq.TN) then
     call pf_snd(human,reply,4)
 else if (human.eq.TN) 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 IPfortran, these gymnastics are not of concern. This shows, however, why one should not write the statement

if(ThisNode.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 IPfortran listing for the dining philosophers problem follows.

```
   PROGRAM Philosophize
c  Dining Philosophers Problem using {\Plang}
c
   if (ThisNode .lt. 5) call philosopher(ThisNode)
   if (ThisNode .gt. 4.and.ThisNode.lt.10) then
       call chopstick(ThisNode)
   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 ',ThisNode
    else if (orate.eq.1.and.activity.eq.eats) then
      print,' I continue to EAT',
  .   ' Philosopher ',ThisNode
   endif
   RETURN
   END

   SUBROUTINE go_think(activity,iseed)
   IMPLICIT NONE
   REAL fate,enough
   INTEGER activity, iseed
   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 ',ThisNode
      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 bamboo1,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
```

6 References

Note that University of Houston Research Reports are available via anonymous *ftp* from *karazm.math.uh.edu* in the directory *pub/Reports.*

[BCS91] Babak Bagheri, Terry W. Clark and L.Ridgway Scott, "IPfortran (a parallel extension of Fortran) reference manual," *Research Report UH/MD-119*, Dept. of Mathematics, University of Houston, 1991.

[BRS90] B. Bagheri, B. Raghavachari and L.R. Scott, "Sharing variables in parallel programs," *Research Report UH/MD-79*, Dept. Math., University of Houston, 1990.

[CHKKS92] Terry W. Clark, Reinhard v. Hanxleden, Ken Kennedy, Charles Koelbel, and L. Ridgway Scott, "Evaluating Parallel Languages for Molecular Dynamics Computations," *Proceedings of the Scalable High Performance Computing Conference the Scalable High Performance Computing Conference*, April 26-29, 1992, Williamsburg Hilton, Williamsburg, VA., (to appear).

[CMS91] Terry W. Clark, J. Andrew McCammon and L. Ridgway Scott, "Parallel Molecular Dynamics," *Proc. Fifth SIAM Conf. on Parallel Proc. for Sci. Comp., 1991*, to be published by SIAM.

[DIJ65] E.W. Dijkstra, "Cooperating Sequential Processes," *Technical Report EWD-123*, Technological University, Eindhoven, The Netherlands, 1965.

[GR88] Geist and Romine, "LU factorization algorithms on distributed-memory multiprocessor architectures," *SIAM J. Sci. Stat. Comput. 9*, 1988, 639-649.

[IPfortran] For more information about IPfortran, send e-mail to scott@uh.edu.

Acknowldegments

We thank J.A. McCammon and the Institute for Molecular Design for providing helpful discussions, "real-world" computational systems, and support in the development and application of IPfortran. The Institute for Molecular Design is supported in part by the National Science Foundation with additional support from Intel Corporation. Scott was supported in part by the National Science Foundation through award number DMS-8903548.