Spring 1997

# Parallel programming

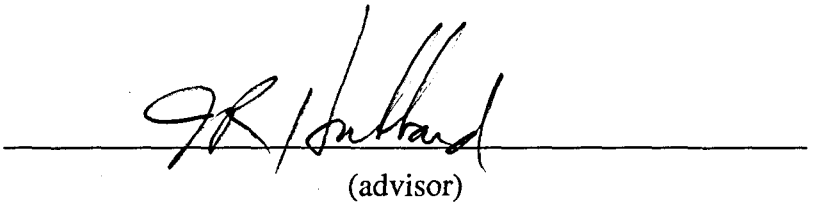
Peter Dailey
*University of Richmond*

# Parallel Programming

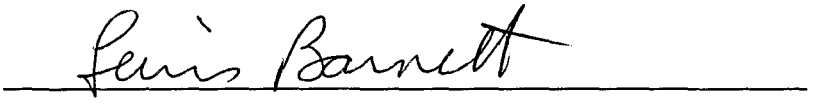
**Peter Dailey**
**Honors Thesis**[1]
**Department of Mathematics and Computer Science**
**University of Richmond**
**April 18, 1997**

1. Under the direction of Dr. John Hubbard

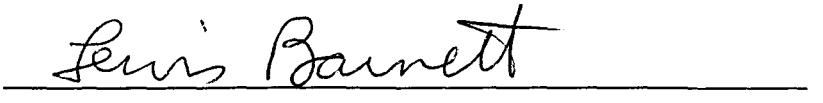
This paper is part of the requirements for the honors program in computer science. The signatures below, by the advisor, a departmental reader, and a representative of the departmental honors committee, demonstrate that Peter Dailey has met all the requirements needed to receive honors in computer science.



(advisor)



(reader)



(honors committee representative)

# Parallel Programming

by

Peter Dailey

April 14, 1997

Dr. Hubbard

The speed of technology is always increasing, especially in the field of computing. Unfortunately, the size of the problems needing to be solved are also growing in many areas. In order to keep up with this, parallel computing has become an important research area. The term parallel computing essentially refers to using multiple processors cooperating to solve a problem. For certain problems this can speed up the solution by a factor of N, the number of processors being used. There are algorithms, for which there is no speed increase due to certain dependencies.

Many different architectures of parallel machines have been developed, the most common of which are SIMD and MIMD. These two acronyms stand for Single Instruction Multiple Data, and Multiple Instruction Multiple Data respectively. The first of these refers to a machine which solves a problem by executing a sequential algorithm simultaneously on many elements of the data using the N processors. This type of computer is often referred to as a vector processor. The second architecture is not bound to any simple pattern as is SIMD. MIMD can execute many different instructions simultaneously on different data. Introduced with this, however, are many new problems such as synchronizing the actions of the processors when necessary, and keeping data consistent. The latter at times can be solved by sharing data. Any data that must be simultaneously accessible to all processors may be kept in a common area often with another node in charge of checking accessibility. The machine I used, an IBM SP2, is of the second type. It is a machine made up of 48 RS/6000 nodes. On this machine, the nodes are split into two pools, 0 and 1. Pool 0 consists of eight processors and is used for interactive processes. The larger pool, of 40 processors, is used for batch processes. I only used pool 0, so I was limited to 8 processors.

In order to program for these computers, new programming languages must be designed which can specify the distribution of data, synchronization of the program, and message passing between processors. The languages I used are MPI, which is actually a language extension, and High Performance Fortran. I chose these languages because they both have compilers on the IBM SP2.

MPI, or Message Passing Interface, is a facilitator for interprocessor communication for use with either the Fortran 77 or C programming languages. This interface was initially begun in April of 1992 at the Workshop on Standards for Message Passing in a Distributed Memory Environment. Two and a half years passed before the fully standardized MPI version 1.0 was released. One of the major goals of MPI was a high degree of portability. For this reason, the MPI library of functions has been implemented for a wide range of systems. The fact that the interface can be used with both Fortran and C extends the bounds of its usage even further.

Two major drawbacks of MPI at the current time are the lack of both explicitly shared memory and support for debugging. Both of these drawbacks affected the work which I have done this year in one way or another. Many parallel algorithms, such as the Fast Fourier Transform, implicitly assume the presence of shared memory in their "Big Oh " estimates. This does not mean that the algorithms cannot be written without it, for it is quite easy to write a procedure which mimics sharing the data. This does, however, place the optimal time estimates out of reach because of the need for additional communication to keep all distributed data consistent. This will be discussed in greater detail later.

The nonexistence of debugging facilities also greatly hindered my work with parallel algorithms. Programming for N processors at once greatly complicates the algorithms. Not only would debugging allow for the testing of the logic, but also assure the programmer that sent

message is received correctly. This was perhaps the greatest problem for me. At times I needed to completely rewrite the program sequentially for a local compiler and use the local debugger to test the logic. This was comparatively easy, however in relation to matching corresponding sends and receives.

The creation of MPI datatypes greatly facilitated achieving the goal of portability. Because different machines have distinct representations for types of data, a standardized representation was needed as an intermediary. Once these were created, each machine's implementation of MPI only needed functions to translate the representation of the data back and forth. These MPI datatypes correspond to datatypes existing in Fortran and C. The following is a chart of how MPI types relate to those of C:

| | |
|---|---|
| MPI_CHAR | signed char |
| MPI_SHORT | signed short int |
| MPI_INT | signed int |
| MPI_LONG | signed long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |

All MPI datatypes are preceded by "MPI_." This is true of all datatypes, procedures, and constants provided by the interface. When programming in MPI, this prefix is reserved, and cannot be used by the programmer. The above datatypes are only a subset of those available in MPI, some of which do not correspond to types in the programming language. Some others do not correspond to types in the programming languages. The MPI_BYTE datatype is an example of one. This type essentially tells the interface that the data is devoid of format. Any data sent of

this type will be received exactly the same as it was sent even if the communication is between two vastly different machines. User defined types for C structs can also be added by the programmer. This will be explained later.

Out of all 125 procedures included in the Message Passing Interface, there are six which a MPI programmer cannot do without. The C prototypes of these are the following:

```
MPI_Init(int *argc, char ***argv);
MPI_Comm_size(MPI_Comm comm, int *size);
MPI_Comm_rank(MPI_Comm comm, int *rank);
MPI_Send(void* buf, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm
        comm);
MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
        MPI_Status *status);
MPI_Finalize();
```

MPI_Init must be called before any other MPI procedures can be run. This routine is responsible for setting up the MPI environment. This may mean drastically different things on separate machines. For this reason, much of MPI's portability rests on the shoulders of this routine. Therefore it cannot be stated here what this routine does, because it changes from one environment to another. This procedure is even different depending on the language it is being used with, for example the Fortran prototype for this function has no parameters. When used with C, one can specify on the command line the value of certain environment variables. For example MP_PROCS is the environment variable which dictates how many processors will simultaneously run the program. Alternatively the command-line statement " foo -procs 4 " specifies four processors will run foo.

The second of these functions, MPI_Comm_size(), returns the size of the communication world specified by the first parameter comm. This parameter is of type MPI_Comm which is an addition to the C library of types. MPI_COMM_WORLD is the default communication descriptor which

describes the whole communication world, containing all processors running the current program. When this descriptor is sent, the value of MP_PROCS (or the command line value of -procs) is returned in the second parameter. I have used the variable nproc, a C int, for this purpose in each of my programs. The communication world can be broken up into groups and given topologies, in which case the programmer would define new communicators, but I did not find the need to use these.

Each processor in the communication world is given a rank number. An inquiry to the communication descriptor returns this value. MPI_Comm_rank() executes this inquiry. These rank numbers consist of the integers from 0 to one less than nproc. This convention facilitates the logic of the algorithms, allowing the programmer to specify which processors do certain jobs. This also allows for direct point to point communication, which I will describe in the next couple of paragraphs. I have developed my own convention in calling this variable mytid, for my task identification. There is however a difference between this and nproc, other than the obvious. Mytid contains nproc different values even though it is only a single integer, while nproc remains the same on all processors. This has been a constant point of confusion for me in my work with MPI. In each of my programs, I have also defined a master process, namely where mytid == 0. I have done this in order to have one processor in charge of coordinating the input/ output and data consistency.

Once the communication world and the processors are initialized properly, we can use MPI to send data between the nodes. MPI_Send() and MPI_Recv() are the functions most often used for this. These send and receive data, respectively, while blocking further execution of the program until completion of the functions. MPI_Isend() and MPI_Irecv() correspond to the non-

blocking forms of these functions. The first three parameters of these functions describe the data being transmitted. The first, void *buf, is the address of the buffer to be copied out of or into. The type and number of elements of type are contained in the next two parameters, int count, MPI_Datatype datatype to be communicated. The count does not refer to the number of bytes sent, because that is machine dependent, but instead represents the number of elements of the datatype transmitted. In this scheme, type checking is done three times. The datatypes specified in each of the function calls must correspond to those of the respective send and receive buffers. The types specified by in the send and receive parameter lists must also match. In this way type checking is accomplished between the types of each host language and the MPI datatypes, as well as between the two ends of the communication channel. The count on each end does not figure into the type checking, however. The size of the receive buffer does not have to be the same size as the send buffer, it merely has to be large enough to hold the contents of the send buffer.

There are two ways to ensure that a receive procedure acquires the correct message. This is done by matching the third and fourth parameters of the send and receive calls, int source (or destination), int tag. A message sent to a particular destination process, specified by rank number, cannot be received by any other processor. Likewise a receive specifying a particular source cannot receive a message from another processor (unless other non-blocking receives are pending). Since the programmer can never be positively sure which order processes will execute sends and receives in, one must be careful or the program will be blocked from completing. For this reason tags can be used. A tag is merely an integer which provides additional information other than the source processor. If every message has a unique tag, then a receive function doesn't need to know the source of the message it is waiting on. In such a case, the constant

MPI_ANY_SOURCE can be sent as the source rank, and the tag determines which message is received. Similarly, there is also a MPI_ANY_TAG constant when the tag value doesn't matter. The two constants can be used simultaneously to receive any message currently in transit. In the case of a successful transmission, all MPI communication procedures will return the constant MPI_SUCCESS. Notice that for blocking functions error checking is not necessary since the program could not possibly continue until both routines finish properly. Error codes are returned in other cases, but they depend on the implementation. The last parameter MPI_Status &status of MPI_Recv()contains the error codes and other information about the message acquired. If MPI_ANY_SOURCE or MPI_ANY_TAG were sent as parameters, the source and tag of the corresponding send routine could be extracted from status.

At the end of the program, the routine MPI_Finalize() is called. No messages can be in transit when this procedure is called because no MPI routine can be called after it. The communication is shut down and cannot be reopened except by restarting the program. This does not have to be the final line of the program, however. The processes will still be running (all nproc of them) subsequent to this function call, but under an MISD paradigm, which is essentially useless.

Broadcasting is another concept which I found particularly useful, especially in regard to data consistency. This is implemented by the following function:

```
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

This function works simultaneously as a multiple send and receive. In only this one line of code (actually executed nproc times), processor root's contents of buffer are sent to all other processors in the communication world described by comm which store them in their own buffer. This

process does not need to be accompanied by a corresponding receive. This is used mostly to make all data consistent.

These are by no means the only functions which I have found useful. Timing the execution of the program and creating new MPI datatypes facilitated not only my programming, but also the analysis of the algorithms. There is not much difference between the timing features included in MPI, and those in the C language. Times in MPI are returned by the following function:

```
double MPI_Wtime(void);
```

The return value of this function is a double precision float corresponding to the number of seconds, not ticks, which have passed since some fixed time in the past. The distinguishing feature between this and the C timing functions is that this returns a time to all processors. There is no guarantee that all processors will have the same time elapsed, and in fact they will most likely be different. Calling this function at the beginning and at the end of the function and printing the difference will then show the user the time elapsed.

In my earlier programs, I only transmitted data of predefined datatypes. As my work progressed, I incurred the need to pass C structs, especially in the programming of the Fast Fourier Transform, which I will discuss in greater detail later. I will use the example of the complex datatype which I used in the Fast Fourier Transform to assist in my presentation of this important concept. The C Complex datatype is created by the following declaration:

```
Struct Complex {
        double re;
        double im;
};
```

We now have a C type which we can use to build a corresponding MPI type. The first step in this process is giving our new type a name, for which I used m_complex, to help me remember that it was the MPI datatype, without using the forbidden "MPI_" prefix. To name our type we declare it as a variable of type MPI_Datatype, another new addition to the C type library. In order to describe our new datatype, we must create some arrays to hold information about the new type. These arrays must have as many elements as there are components in the C derived type. These declarations are shown below.

```
MPI_Datatype m_complex;
MPI_Datatype type[2] = {MPI_DOUBLE, MPI_DOUBLE};
int blocklength[2] = {1, 1};
MPI_Aint disp[2] = {0, sizeof(double)};
```

The names of these variables need not have any relation to the name of the MPI type being built, as they will be bound to it later. The second declaration, that of type[2] contains the datatypes of the subtypes within the Complex structure. For this reason, it is initialized to {MPI_DOUBLE, MPI_DOUBLE}. Since the MPI environment only needs to know the standardized representation, the MPI types are used, facilitating representation conversions. Blocklength[2] is set to {1, 1}, because both of the subtypes are single elements, not arrays. Should the derived datatypes consist of arrays, then this array would hold the size of the arrays. MPI_Aint is the C type of the last necessary variable. This type is used for arbitrary addresses, or offsets. In this case the variable disp[2], contains the offsets of the two MPI_DOUBLE pieces within m_complex. The first will be 0, because the address of the buffer of type m_complex should contain the re component of the first complex variable. The offset for the im component, will then be sizeof(double). There is a function available to return this information to facilitate programming. It is:

```
int MPI_Address(void* location, MPI_Aint *address);
```

This feature is available to facilitate portability of programs. It will simply return the offset of the second parameter with regard to the first for the machine it is executed on.

Once these descriptors of are initialized, steps can be taken to commit these changes to the MPI environment. This is done using the two routines shown below:

```
MPI_Type_struct(int count, int * array_of_blocklengths, MPI_Aint *array_of_displacements,
      MPI_Datatype *array_of_types, MPI_Datatype, newtype);
MPI_Type_commit(MPI_Datatype *datatype);
```

The first function, MPI_Type_struct(), allows us to describe the data as a set of "blocks." The first block of data for example Complex.re is represented by the first element of the arrays array_of_types, array_of_blocklengths, and array_of_displacements. We can see that it has type MPI_DOUBLE, blocklength one, and displacement 0. In order to render this new type useful to the MPI environment, we must save it. This is achieved by the second routine, MPI_Type_commit(). It is at this time that the system analyzes to see if any optimizations can be made. We are now free to send variables of type m_complex between processors.

The other language I programmed in was HPF, or High Performance Fortran which is an extension to the Fortran 90 programming language. While Fortran 90 did include some features for parallel programming, they were not extensive. The High Performance Fortran Forum (HPFF), an international group of some 500 interested programmers cooperated in an effort to extend those capabilities. The official specifications for this language are copyrighted by Rice University. HPF was planned to facilitate data distribution and computationally intense programming. Its capabilities include use with MIMD and SIMD machines.

High Performance Fortran allows for parallelism through the use of some preprocessor directives. While it has some capabilities lacking in MPI, it is not nearly as flexible. MPI does not support shared memory, while HPF does. Parallelism and distribution in HPF, however are not defined by the programmer as they are in MPI. In HPF, in most instances, the programmer merely places specially formatted comments in the code which tell the compiler that a particular section of the code is parallelizable. In HPF, data can be distributed onto a topology of processors using the following declarations:

```
DOUBLE PRECISION, DIMENSION(N)       :: Y
!HPF$ PROCESSORS, DIMENSION(M)       :: PROCS
!HPF$ DISTRIBUTE(CYCLIC) ONTO PROCS :: Y
```

After declaring the array, in this case Y, the two preprocessor directives specify how the array shall be distributed. Every directive in the HPF language must begin with !HPF$, CHPF$, or *HPF$. The first directive in the code above tells the preprocessor that you have a linear arrangement of M processors running the program. Next, the DISTRIBUTE command is used to describe the actual distribution of the data. In this case it is to be done in a CYCLIC manner. This scheme is shown below, with N = 16 and M = 4:

| Processor | Elements of Y |
|---|---|
| 1 | 1, 5, 9, 13 |
| 2 | 2, 6, 10, 14 |
| 3 | 3, 7, 11, 15 |
| 4 | 4, 8, 12, 16 |

This means that whenever Y is used in a parallelizable computation, each processor will do the computations on its assigned elements. CYCLIC is not the only alternative for the distribution directive. BLOCK is another scheme which I found useful. This scheme is as follows, with the same parameters:

| Processor | Element |
|---|---|
| 1 | 1,2, 3, 4 |
| 2 | 5, 6, 7, 8 |
| 3 | 9, 10, 11, 12 |
| 4 | 13, 14, 15, 16 |

In addition to the DISTRIBUTE command, there is also an ALIGN directive. This works is almost like the FORTRAN 77 EQUIVALENCE routine. When aligning two pieces of data, however, it does not mean that they refer to the same memory location, but instead that they are distributed to the same processor. This is important, because when used correctly, it will minimize the interprocessor communication traffic.

Once data is distributed, much of the parallelization is specified by the programmer. Using the INDEPENDENT directive, "!HPF$ INDEPENDENT," the preprocessor knows that the loop to follow can be executed in parallel. The work will be parallelized according to the specified scheme. This statement can include a DO or a FORALL. DO is a holdover from the Fortran 77 language. FORALL is a new construct which is automatically parallelized. It has a form very similar to a DO loop. The importance of this construct lies in the simplicity it creates for array computations. Any data needed in a FORALL statement which is not distributed, or not correctly distributed, will be sent via a message to the processor performing the computation. In parallelized loops, functions or subroutines may be called, but must be declared PURE. This attribute was designed mostly to be used in conjunction with FORALL statements. Use of this attribute is essentially a promise by the programmer to the compiler that any side effect caused by the function were not intentional, and therefore not wanted. PURE functions are allowed to return a value, but not alter any other data. This construct first appeared in the Fortran 90 language.

There are other constructs in the HPF language which are intrinsically parallelized. These consist largely of array operations, which were first overloaded in the Fortran 90 language. These show a much higher level of logic. The assignment and arithmetic operators have been overloaded along with many others provided externally. Array operations do not need to be declared independent, as they are assumed to be. These operations will also be distributed according to the specified scheme.

Beyond these constructs, HPF has also added many other new high level concepts. For one, programmers have the added capabilities to create structures of data very similar to those in C. The Complex class, complete with overloaded operators is predefined in High Performance Fortran. This made the logic of the parallel Fast Fourier Transform in High Performance Fortran much more succinct than the identical C and MPI code. This Complex class can be defined with elements to be of any numeric datatype, another convenient feature.

Fortran 90 simultaneously introduced recursion into its arsenal. Any recursive function or subroutine is defined according to the following pattern:

```
RECURSIVE FUNCTION FOO(N) RESULT(ANS)
INTEGER ::ANS
```

Because of the early Fortran conventions, the RESULT clause had to be added to the language. Instead of assigning the return value to the name of the function, the programmer must define the new return variable. Assignment to that variable ends the execution of the function. This had to be done so that the name of the function would not refer to both a variable and a function. Since the type of the return value is not defined in the function header, it has to be specified as a variable declaration. If this is not done, it will obey the Fortran I-through-N rule.

Over the course of the semester, I wrote numerous programs in both of the languages described above in order to compare them. Numerical integration was the first of these programs. In each program, I integrated $4/(1 + x^2)$ from 0 to 1 splitting the region into 10,000 intervals. This algorithm is very parallelizable, because instead of one processor calculating the areas of all 10,000 regions, each of the N processors do 10,000/N of them. This method is then sped up by a factor of N, not including the time spent on message passing. In order to more closely compare the two, I have split the algorithms into three parts. These parts are the initialization, calculation, and collection of the data. Following is the code for the initialization written in C using MPI:

```
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"

#define NUMRECS 10000
#define MASTER 0

MPI_Status status;
MPI_Request request;

double f(double x) {
   double y;
     y = 4.0/ (1.0 + x*x);
     return y;
}

main(int argc, char **argv)  {
   double h, mypi, pisum, piave, pirecv,
            rec_area[NUMRECS];
   int    i, n, mytid, nproc, source,
            mtype, msgid, nbytes, mystart,
            rcode;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &mytid);
   MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

And in High Performance Fortran:

```
PROGRAM PI_EXAMPLE
      ! Compute the value of pi by numerical integration

INTEGER, PARAMETER    :: N = 10000
DOUBLE PRECISION, PARAMETER       :: H = 1.0 / N
DOUBLE PRECISION        :: MYPI
DOUBLE PRECISION, DIMENSION(N)    :: RECT_AREA

   !HPF$   PROCESSORS, DIMENSION(8)  :: PROCS
!HPF$   DISTRIBUTE (BLOCK) ONTO PROCS :: RECT_AREA

DOUBLE PRECISION FUNCTION F(X)
      DOUBLE PRECISION :: X
      F = 4 / (1.0 + X*X)
END FUNCTION F
```

Much less initialization is necessary in the Fortran version. While the variable declarations are the same, there are some important differences. The PROCESSOR and DISTRIBUTE directives in HPF serve to initialize the parallelism. For this reason, the dimension of the processors must be hardcoded into the declarations in HPF. The number must be either an integer or an integer constant. This means that if a programmer wished to run the application on only four processors, the code would have to be altered and recompiled. In the C version, the calls to the MPI_Init(), MPI_Comm_source(), and MPI_Comm_rank() extract this information from the environment. Therefore it does not have to hardcoded, but can instead be done on the command line. This is very important because it makes the application more flexible. In addition to the number of processors, the rank of each processor is returned by these function calls. This information is very important in the next section of the algorithm, the calculation.

In both programs, I have split up the work to be done in a BLOCK manner. Since this is not predefined in MPI, I have simulated it. Below is the code:

```
  mystart = NUMRECS/nproc * mytid;
h = 1.0/NUMRECS;
mypi = 0.0;

  for(i = 0; i < (NUMRECS/nproc) - 1; i++){
        rec_area[i + mystart]= h* f(h*((i + mystart)- .5));
        mypi += rec_area[i + mystart];
  }
```

In order to simulate the block distribution of the work in C, one must assign the work to each processor. This is facilitated by the use of the variable mytid. With 10,000 elements of the array to do work on, and assuming 8 processors are being used, each processor will calculate the area of 1,250 rectangles. Since the ranks of the processors range from 0 to 7, each processor calculates the area of the blocks starting with NUMRECS/nproc * mytid. The variable mystart is used to store this number in the code. Therefore processor 0 will calculate blocks 0 through 1,249; processor 1, 1,250 through 2499; and so on. Each array element is set to the area of the corresponding rectangle, with h representing the width of the rectangle, and the value returned by f() being the height. Each processor has a local copy of mypi, which holds its subtotal of the overall area. Since this value is not communicated between processors, it only holds the total for the local 1250 subregions.

The calculation section of the HPF code is much more succinct, as shown below:

```
!HPF$ INDEPENDENT
DO I = 1, N
     RECT_AREA(I) = H * F(H*(I-0.5))
END DO
```

Since RECT_AREA was declared as BLOCK distributed in the initialization stage, the programmer need only tell the compiler that this section of the code can be parallelized. The work will still be done in the exact same manner as in the C version. This section in the Fortran code does not, however, do any summation, which is partly done in the C code. That part is inseparable from the final summation, which is done in the final stage of the program.

To illustrate more the effectiveness of the Fortran method, let's pretend that it was found later that a CYCLIC distribution of the data would be more efficient. In order to change the Fortran code, only one word in the code would need to be changed, BLOCK to CYCLIC in the DISTRIBUTE directive line. Unfortunately, it is not quite so easy in C. The actual logic of the for loop must be altered. These changes are shown below:

```
  mystart = mytid;
h = 1.0/NUMRECS;
mypi = 0.0;

  for(i = 0; i < NUMRECS- nproc; i += nproc){
        rec_area[i + mystart]= h* f(h*((i + mystart)- .5));
        mypi += rec_area[i + mystart];
  }
```

The for loop must be changed to increment by the number of processors every iteration. At each loop i + mytid will be calculated by each processor. While this is not a drastic alteration, it calls for somewhat more thought than changing one word.

The final stage of this algorithm is the collection of the data, and the printing of the result. It is at this stage where the MPI message passing can become logically difficult.

```
if(mytid != MASTER) {
   MPI_Send(&mypi, 1, MPI_DOUBLE, MASTER,
                  mtyid, MPI_COMM_WORLD);
}
```

```
    else {
          pisum = mypi;
          for (n = 1; n < numprocs; n++) {
              MPI_Recv(&pirecv, 1, MPI_DOUBLE,
                       MPI_ANY_SOURCE, n,
                       MPI_COMM_WORLD, &status);
              pisum += pirecv;
          }
    }
    if(mytid == 0)
     printf("Mypi is approximately %10.8f\n",
              pisum);          MPI_Finalize();
}               /*End of the program*/
```

As mentioned in my overview of MPI, I designate the processor with rank 0 as the "master" process. This is done in the initialization step by creating the constant MASTER = 0. For the first time this convention comes into use. Every non-root process sends its subtotal to the master process. The master in turn collects them all and totals the subregions. The total for the area under the curve is then printed out. Notice that in the MPI_Send() and MPI_Receive() I used the tag to match up the calls. In this instance even matching tag numbers was not necessary, since all messages were processed in the exact same way. On the contrary, I could have made the source parameter n for the call on MPI_Receive(), rather than MPI_ANY_SOURCE, and it would have had the same effect on the program. The last action of this application is to print out the final value of pi, held in pisum. Since only the root has the correct value of pisum, only the master is allowed to print. This is important, because all non-root process copies of the variable will contain garbage since they have not been assigned a value.

The final piece of the Fortran program looks like this:

```
MYPI = SUM(RECT_AREA)
PRINT *,MYPI

END PROGRAM PI_EXAMPLE
```

This is infinitely more simplified than the C program. The collection and summation of the areas are done in only one line of code as opposed to seven for in the C code. The SUM() function was not defined by myself, it is predefined. It adds together the contents of each element of the array passed to it, and returns the result. Unlike the C version, mypi will have only on value, as it is a shared variable. The print statement is also executed only once, in direct contrast to the MPI print statement.

Notice that while these two programs do the exact same thing, the C code consists of twice as many lines of code as does the Fortran (36 to 18). These numbers do not count spaces, lines consisting solely of a curly bracket, or line continuations. This is a tribute to the power of the High Performance Fortran constructs. While it takes seven statements in C to collect and sum up the areas of the 10,000 rectangles, this is achieved by a call to a single predefined function in HPF. Since HPF passes messages automatically when needed, no message passing needs to be programmed. This is in direct contrast to C in which the programmer must specify all message passing. The modifiability of the Fortran code as was shown in the BLOCK, CYCLIC example is also superior to that of C. However, C does allow for better program flexibility, especially in the example of changing the number of processors.

Another example application to illustrate the differences between the two languages is Dorn's $n^{th}$- Order Horner's method. Horner's method is an algorithm used to quickly find the value of a polynomial at a given point. Dorn's Method is a parallelized version of the algorithm using the recurrence relations shown below:

$p(x) = a_0 + a_1x + a_2x^2 + \ldots\ldots + a_{15}x^{15}$ can be broken up into

$p_0(x^n) = a_0 + a_n x^n + a_{2n} x^{2n} + a_{3n} x^{3n}$

$$p_1(x^n) = a_1 + a_{n+1}\, x^n + a_{2n+1}\, x^{2n} + a_{3n+1}\, x^{3n}$$
$$p_2(x^n) = a_2 + a_{n+2}\, x^n + a_{2n+2}\, x^{2n} + a_{3n+2}\, x^{3n}$$
$$p_3(x^n) = a_3 + a_{n+3}\, x^n + a_{2n+3}\, x^{2n} + a_{3n+3}\, x^{3n}$$

Clearly then :

$$p(x) = p_0(x^n) + p_1(x^n)\, x + p_2(x^n)\, x^2 + p_3(x^n)\, x^3$$

In order to most easily test the outcome, I used a $15^{th}$ degree polynomial with all coefficients equal to one. When evaluated at two, this will be adding the first sixteen powers of two, giving $2^{16}$ -1 or 65,535. This algorithm is most effectively parallelizable when the degree of the polynomial is divisible by the number of processors. For this application I used only four processors, although it would still work with eight. Using a $15^{th}$ degree polynomial, and 4 processors, the time needed to compute the total will be cut in half. Sequentially this method would need enough time to execute 120 multiplications and 15 additions. Executing part of the algorithm in parallel means that once the first 24 multiplications and 3 additions are done by each of the processors, only 6 further multiplications and 3 additions must be executed. In terms of operations then, the parallel algorithm is better by almost a factor of N, 135 to 36. This does not mean that less operations are actually done, but rather that since so many are done simultaneously, this is how many are essentially sequential.

The initialization stage of the two programs do not have many differences from the first stage of the numerical integration programs. The C code begins:

```
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"

#define N 4
#define MASTER 0
```

```
MPI_Status status;
MPI_Request request;

double power(double base, int exp) {
   double answer = 1;
   int I;
   if(exp < 0)
      return 0;
   if(exp > 0)
      for(I = 1; I <= exp; I++)
         answer *= base;
   return answer;
}

main(int argc, char **argv)  {
  double  a[N*N], temp, myfinal = 0.0,
          final, x= 2.0;
  int     i,j,k, mytid, nproc, source,
          mtype, msgid, nbytes, rcode;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &mytid);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

The Fortran code also looks much the same:

```
PROGRAM HORNER

   INTEGER                      :: I, J
   INTEGER, PARAMETER           :: N = 4
   REAL, DIMENSION (N*N)        :: A
   REAL, DIMENSION (0:N-1, N) :: P
   REAL, DIMENSION(0:N-1)       :: POWERS
   REAL, DIMENSION (N)          :: XN, SUMS
   REAL                         :: X, ANSWER

   !HPF$ PROCESSORS, DIMENSION(N) :: PROCS
   !HPF$ DISTRIBUTE(BLOCK) ONTO PROCS :: POWERS, XN, SUMS
   !HPF$ DISTRIBUTE(BLOCK, *) ONTO PROCS :: P
```

The only new concept in these two sections of code is the final statement of the Fortran piece.

The DISTRIBUTE directive has a two dimensional argument. In this case it is BLOCK, *. A two

dimensional array being distributed must have the same number of arguments in the directive. These arguments, however can be blank, or *. The two dimensional distributions are shown below with a four by four array and four processors, and p1 meaning processor 1:

DISTRIBUTE A(BLOCK, BLOCK)

| | | | |
|---|---|---|---|
| p1 | p1 | p2 | p2 |
| p1 | p1 | p2 | p2 |
| p3 | p3 | p4 | p4 |
| p3 | p3 | p4 | p4 |

DISTRIBUTE A(*, BLOCK)

| | | | |
|---|---|---|---|
| p1 | p2 | p3 | p4 |
| p1 | p2 | p3 | p4 |
| p1 | p2 | p3 | p4 |
| p1 | p2 | p3 | p4 |

CYCLIC can also be used in a multidimensional distribution. DISTRIBUTE(*, CYCLIC) with the above parameters it is exactly the same configuration as the DISTRIBUTE(*, BLOCK), which is shown above, but this is a rare case. A much larger two dimensional array would be needed to illustrate the differences. Below is the scheme presented by DISTRIBUTE( CYCLIC, CYCLIC):

| | | | |
|---|---|---|---|
| p1 | p2 | p1 | p2 |
| p3 | p4 | p3 | p4 |
| p1 | p2 | p1 | p2 |
| p3 | p4 | p3 | p4 |

Knowing how the P array is distributed, the computations can now be examined.

Stage two consists of the calculation of the powers of x, in this case 2.0, and the summation of the $p_i$ 's. In C the code consists of the following:

```
for( i = 0; i < N*N; i++)
        a[i] = 1;

for(j= 0; j < N*N/ nproc; j++)
    myfinal += a[(N*j)+mytid] *power(x, N*j);

myfinal *= power(x, mytid);
```

In the C version of the code, instead of using an array POWER as in the Fortran code, I merely created a power function. In the above code, each processor finds the sum of $p_{mytid}$ from the recurrence relations shown above. Myfinal is then multiplied by $x^{mytid}$ before returned. While it would have been easy to implement the power() function in the Fortran code, I did not in order to illustrate the use of the overloaded array operators. The code is as follows:

```
X = 2.0

POWERS(0) = 1

!HPF$ INDEPENDENT
DO I = 1, N*N
   A(I) = 1
END DO

DO  I = 1, N
   POWERS(I) = POWERS(I-1) * X
END DO

XN(:) = 1

DO I = 2, N
   XN(I) = XN(I-1)* POWERS(N)
END DO

!HPF$ INDEPENDENT
DO I = 0, N-1
   P(I, :) = A(1:N*(N-1)+1:N) * XN(1:N)
END DO

!HPF$ INDENPENDENT
DO I = 1, N                    !GET SUMS OF P[I]'S
   SUMS(I) = SUM( P(I-1,:))
END DO
```

The first two DO loops are merely for initializing the A and POWER arrays. This also applies to the next DO loop. Notice that the loop for initializing the POWER array is not designated as INDEPENDENT. This is the first instance of a dependent loop. Since each element of the array is x

times the previous element they are dependent, and therefore must be done in sequence. To clarify the meaning of some of the arrays, XN(I) holds $x^{i*n}$, POWER(I) holds $x^{i}$, and SUMS(I) represents the sum of each $p_i$. Perhaps the most confusing piece in this logic is the array assignment. This statement ' P(I, :) = A(1:N*(N-1)+1:N) * XN(1:N),' means that each element of the one-dimensional array P(I, ) gets the product of the corresponding elements of the A() and XN() arrays. Any arrays used in statements such as these must have the same shape. This is checked by the compiler. The colon in the subscript of the array makes this a multiple assignment. The numbers on either side of the colon represent the upper and lower bounds of the selected section of the array to be used. If a second colon appears the final parameter is a step variable. So the subscript of the A() array above means 1, N+1, 2N+1, etc.. This type of statement is implicitly parallelized, even without the use of an INDEPENDENT directive.

Finally, the collection stage of Dorn's method consists merely of adding together all of these intermediate sums and printing the result. Once again, the Fortran code is infinitely more succinct than the C code. The Fortran,

```
SUMS = SUMS(:) * POWERS(:)
ANSWER = SUM(SUMS)

PRINT *, ANSWER

END PROGRAM HORNER
```

and the C,

```
if(mytid != MASTER) {
     MPI_Send(&myfinal, 1, MPI_DOUBLE, MASTER, mytid, MPI_COMM_WORLD);
   }
else{
   final = myfinal;
   for(k = 1; k < nproc; k++) {
        MPI_Recv(&temp, 1, MPI_DOUBLE,
```

```
                MPI_ANY_SOURCE, k, MPI_COMM_WORLD,
                &status);
      final += temp;
   }
   }
   if(mytid == MASTER) {
      printf("The answer is %10.8f\n", final);
    }
   MPI_Finalize();
   }
```

Once again, these statements send all of the intermediate totals back to the master process which sums them up, and prints out the total. The Fortran code includes another implicitly parallelized array operation, the results of which are simultaneously being summed.

A more complicated example which I worked on is the Fast Fourier Transform. The Fast Fourier Transform is an improvement to the Discrete Fourier Transform using a complicated set of recurrence relations. The Discrete Fourier Transform is based on the matrix equation $c = Fy$. In this equation, y is the input, a vector with N components. F represents an N x N matrix in which all entries are complex numbers. The k,j component of F is computed by $e^{2\pi jki/N}$. In this equation i represents the imaginary root, $\sqrt{-1}$. These values can be more easily represented as $\omega_{kj}$ or $\cos(2\pi jk/N) + i \sin(2\pi jk/N)$. The real and imaginary components of the complex number $\omega_{jk}$ are the x and y coordinates of the point which falls jk/N of the way around the unit circle traversing counterclockwise. The inverse of this function is almost the same, except that the k,j components of F are found from $e^{-2\pi jki/N}$, or $\cos(-2\pi jk/N) + i \sin(-2\pi jk/N)$. In 1965 J.W. Cooley and J.W. Tukey found a complicated set of recurrence relations which, when solved cut the number of calculations needed from $N^2$ to N. This is commonly referred to as the Fast Fourier Transform.

In order to implement this program[1], the use of a complex type is needed. In HPF the Complex class is predefined as described previously, but in C and MPI it needed to be constructed. The code for these constructions was included earlier.

One of the problems facing a C and MPI implementation of this program is the lack of explicitly shared memory. During each iteration of the main loop in the program, which is shown later, each processor updates the element of the c[] array corresponding to its mytid. Since all of these arrays are local, each processor only has the updated copy of the element it computed. In order to correctly do its work in the next iteration, however, each processor needs all of the updated elements. Shared memory must therefore be mimicked using message passing in this program. Unfortunately the speed of the program will be slowed significantly because of the need for this additional communication. Since the recurrence relation is $\log_2 N$ steps deep, the main loop iterates this many times, and so the data must to be 'shared' $\log_2 N$ times. Therefore, without the support for shared memory in MPI, this program will take much longer than merely the time for the N calculations. The code for the sharing function in MPI is shown below:

```
void share(struct Complex y[], int mytid, int k, int b, MPI_Datatype m_complex) {

int i,j;
struct Complex temp;

if(mytid != MASTER)
    MPI_Send(&y[k], 1, m_complex, MASTER, (N*k) + b, MPI_COMM_WORLD);

for(i = 1; i < N; i++)
    if(mytid == MASTER)      {
        MPI_Recv(&temp, 1, m_complex, MPI_ANY_SOURCE, (N*i) + b, MPI_COMM_WORLD,
            &status);
        y[i] = temp;
}
```

[1] Since the code for the Fast Fourier Transform is lengthy and hard to follow in both languages, I have only included some important pieces of it. The full code for both versions of this program is included in the appendix.

```
MPI_Bcast(y, N, m_complex, MASTER, MPI_COMM_WORLD);
}
```

The above code uses the master process to collect and redistribute all of the updated data. In this instance, the tag parameters are used to ensure the correct message is received. Because the master process has to collect all data first, it must execute N-1 blocking receives. The master process is thus not synchronized with the rest of the processors, and in fact falls well behind. For this reason, a message sent in the first iteration may be received by the master process in the second round unless it has a unique tag value. For this reason, b has been figured into the tag value. Furthermore, unless the source of the message is also known, the value sent cannot be placed in the correct position in the array. The tag value then has become k*N + b, which will be unique for all iteration and source combinations. Once all pieces are collected, the broadcast function quickly redistributes the array to all nodes.

The main loop in the C using MPI is as follows:

```
while( b >= 1)  {
   k = mytid;
   ...
   /*assignments */
   ...
   c[k].re = y[sub].re + sign*((y[sub+b].re*w[exp].re) -( y[sub].im* y[sub+b].im));
   c[k].im = y[sub].im + sign*( (y[sub+b].re * w[exp].im) +( y[sub].im* y[sub+b].re));

   y[k] = c[k];
   share( y, mytid, k, b, m_complex);
   b /= 2;
   }
```

The HPF main loop is as follows:

```
DO WHILE( B >= 1)
     B2 = N/ 2*B
```

```
        DO MYTID = 0, N-1
            K = MYTID
            ...
            !assignments
            ...
            C(K) = Y(SUB) + SIGN*(Y(SUB + B) * W(EXP))
        END DO

        Y(:) = C(:)
        B = B/2
    END DO
```

The HPF version of this program needs a second embedded DO loop. In the C code, however, the work is completely distributed even without a second loop, but the effect is the same. The assignment to c[k] where k = mytid spreads this work out over all of the processors. The distribution of the data in MPI in this example is done much differently than in the to previous examples.

Overall, each of these languages has certain strengths and weaknesses. Using C with MPI allows the programmer more flexibility than High Performance Fortran. Since distribution schemes are predefined in HPF, there only a small number of them. The power of MPI comes from being able to specifically program the work to be done by each processor. The fact that each processor will execute the same program, but with different local information provides the programmer with significant flexibility. Virtually any distribution scheme could be implemented using MPI. There is a tradeoff in providing this much flexibility, however. Since the distribution is programmed rather than declared in MPI, it becomes more difficult to modify, which is a clear advantage for HPF.

Providing automatic data communication also brings tradeoffs. The logic of the HPF programs are greatly simplified by not having to include message passing. On the other hand

specifying communications allows the programmer to picture the efficiency of the scheme better, since the number of messages actually sent can be seen in the code. For example in the Fast Fourier Transform, 2N messages are being passed during each of $\log_2 N$ iterations. In the HPF version of the FFT, close inspection must be made to determine the amount of intercommunication, which is this case in none.

These are only two of the several parallel languages currently available. Although this is a somewhat incomplete comparison of the two languages, the important techniques have been covered. My research is ongoing, particularly with regard to the distribution of the complex arrays in the Fast Fourier Transform. I expect to learn more of the important distinctions of these two languages as I continue my research.

# REFERENCES


1. Numerical Analysis, Richard L. Burden and J. Douglas Faires, 5th Edition, PWS-KENT Publishing Company, 1993.

2. Practical Parallel Processing, Alan Chalmers and Jonathan Tidmus, International Thompson Computer Press, New York, 1996.

3. MPI The Complete Reference, Snir, Otto, Huss-Lederman, Walker and Dongarra, MIT Press, Cambridge, MA, 1996.

4. Using MPI: Portable Parallel Programming with the Message Passing Interface, Gropp, Lusk, and Skjellum, MIT Press, Cambridge, MA, 1994.

5. The Design and Analysis of Parallel Algorithms, Justin R. Smith, Oxford University Press, New York, 1993.

6. Fundamentals of Sequential and Parallel Algorithms, Kenneth A. Berman and Jerome L. Paul, PWS Publishing Company, Boton, MA, 1997.

7. Analysis and Design of Parallel Algorithms, S. Lakshmivarahan and Sudarshan K. Dhall, McGraw-Hill Publishing Company, New York, 1990.

8. The High Performance Fortran Handbook, Koelbel, Loveman, Schreiber, Steele and Zosel, MIT Press, Cambridge, MA, 1994.

9. Frontiers in Applied Mathematics: Computational Frameworks for the Fast Fourier Transform, Charles Van Loan, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1992.

```
Script command is started on Wed Apr 23 13:36:49 EDT 1997.spe01% cat -n picyc.c
     1
     2  #include <stdlib.h>
     3  #include <stdio.h>
     4  #include "mpi.h"
     5
     6  #define NUMRECS 10000
     7  #define MASTER 0
     8
     9  MPI_Status status;
    10  MPI_Request request;
    11
    12  double f(double x) {
    13    double y;
    14    y = 4.0/ (1.0 + x*x);
    15    return y;
    16  }
    17
    18
    19  main(int argc, char **argv)
    20  {
    21  double  h,
    22          mypi,
    23          pisum,
    24          piave,
    25          pirecv,
    26          rec_area[NUMRECS];
    27  int     i,n,
    28          mytid,
    29          nproc,
    30          source,
    31          mystart;
    32
    33  MPI_Init(&argc, &argv);
    34  MPI_Comm_rank(MPI_COMM_WORLD, &mytid);
    35  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    36
    37  mystart = mytid;
    38  h = 1.0/NUMRECS;
    39  mypi = 0.0;
    40
    41  for(i = 0; i < NUMRECS - nproc + 1; i+= nproc){
    42      rec_area[i + mystart] = h* f(h*((i + mystart) + .5));
    43      mypi += rec_area[i + mystart];
    44  }
    45
    46  if(mytid != MASTER) {
    47     MPI_Send(&mypi, 1, MPI_DOUBLE, MASTER, mytid, MPI_COMM_WORLD);
    48       }
    49
    50  else{
    51     pisum = mypi;
    52     for(n = 1; n < nproc; n++) {
    53        MPI_Recv(&pirecv, 1, MPI_DOUBLE, MPI_ANY_SOURCE, n, MPI_COMM_WORLD
 , &status);
    54        pisum += pirecv;
    55        }
    56  }
    57  if(mytid == MASTER)
    58     printf("Mypi is approximately %19.17lf\n", pisum);
```



```
    59  MPI_Finalize();
    60  }               /*End of the program.*/
spe01% mpcc -o picyc picyc.c
spe01% picyc
Mypi is approximately 3.14159265442312430
spe01% exi
Script command is complete on Wed Apr 23 13:37:30 EDT 1997.
```

```
Script command is started on Wed Apr  9 15:00:01 EDT 1997.spe01% cat -n mypi.f
     1        PROGRAM PI_EXAMPLE
     2          ! Compute the value of pi by numerical integration
     3
     4          INTEGER, PARAMETER    :: N = 10000
     5          DOUBLE PRECISION, PARAMETER      :: H = 1.0 / N
     6          DOUBLE PRECISION        :: MYPI
     7          DOUBLE PRECISION, DIMENSION(N)    :: RECT_AREA
     8
     9          !HPF$    PROCESSORS, DIMENSION(8)   :: PROCS
    10          !HPF$    DISTRIBUTE (BLOCK) ONTO PROCS :: RECT_AREA
    11
    12          !HPF$ INDEPENDENT
    13          DO I = 1, N
    14               RECT_AREA(I) = H * F(H*(I-0.5))
    15          END DO
    16
    17          MYPI = SUM(RECT_AREA)
    18
    19          PRINT *,MYPI
    20
    21          CONTAINS
    22
    23          DOUBLE PRECISION FUNCTION F(X)
    24            DOUBLE PRECISION :: X
    25            F = 4 / (1.0 + X*X)
    26          END FUNCTION F
    27
    28        END PROGRAM PI_EXAMPLE
    29
spe01% xlhpf90 -L/usr/lpp/ssp/css/libip/ -o mypif mypi.f
** pi_example   === End of Compilation 1 ===
1501-510  Compilation successful for file mypi.f.
spe01% mypif
 3.14159265442312741
spe01% exit
Script command is complete on Wed Apr  9 15:00:51 EDT 1997.
```

```
Script command is started on Wed Apr  9 14:53:56 EDT 1997.spe01% cat -n horner.c
     1
     2  #include <stdlib.h>
     3  #include <stdio.h>
     4  #include "mpi.h"
     5
     6  #define N 4
     7  #define MASTER 0
     8
     9  MPI_Status status;
    10  MPI_Request request;
    11  double power(double base, int exp) {
    12     double answer = 1.0;
    13     int l;
    14     if(exp < 0)
    15        return 0;
    16     if(exp > 0)
    17        for(l = 1; l <= exp; l++)
    18           answer *= base;
    19     return answer;
    20  }
    21
    22
    23  main(int argc, char **argv)
    24  {
    25  double  a[N*N],
    26          temp,
    27          myfinal = 0.0,
    28          final,
    29          x = 2.0;
    30  int     i,j,k,
    31          mytid,
    32          nproc,
    33          source,
    34          mtype,
    35          msgid,
    36          nbytes,
    37          mystart,
    38          rcode;
    39
    40  MPI_Init(&argc, &argv);
    41  MPI_Comm_rank(MPI_COMM_WORLD, &mytid);
    42  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    43
    44  for( i = 0; i < N*N; i++)
    45      a[i] = 1;
    46
    47  for(j= 0; j < N*N/nproc; j++)
    48      myfinal += a[(N*j) + mytid] * power(x, N*j);
    49
    50  myfinal *= power(x, mytid);
    51
    52  if(mytid != MASTER) {
    53      MPI_Send(&myfinal, 1, MPI_DOUBLE, MASTER, mytid,MPI_COMM_WORLD);
    54      }
    55
    56  else{
    57     final = myfinal;
    58     for(k = 1; k < nproc; k++) {
```



```
    59        MPI_Recv(&temp, 1, MPI_DOUBLE, MPI_ANY_SOURCE, k, MPI_COMM_WORLD
&status);
    60        final += temp;
    61        }
    62  }
    63
    64  if(mytid == MASTER)
    65     printf("The answer is approximately %10.8f\n", final);
    66
    67  MPI_Finalize();
    68  }
spe01% mpcc -o horner horner.c
spe01% horner -procs 4
The answer is approximately 65535.00000000
spe01% exit
Script command is complete on Wed Apr  9 14:54:29 EDT 1997.
```

```
Script command is started on Wed Apr  9 14:58:34 EDT 1997.spe01% cat -n picyc.f
     1        PROGRAM PI_EXAMPLE
     2          ! Compute the value of pi by numerical integration
     3
     4          INTEGER, PARAMETER    :: N = 10000
     5          DOUBLE PRECISION, PARAMETER      :: H = 1.0 / N
     6          DOUBLE PRECISION         :: MYPI
     7          DOUBLE PRECISION, DIMENSION(N)    :: RECT_AREA
     8
     9          !HPF$   PROCESSORS, DIMENSION(8)  :: PROCS
    10          !HPF$   DISTRIBUTE (CYCLIC) ONTO PROCS :: RECT_AREA
    11
    12          !HPF$ INDEPENDENT
    13          DO I = 1, N
    14               RECT_AREA(I) = H * F(H*(I-0.5))
    15          END DO
    16
    17          MYPI = SUM(RECT_AREA)
    18
    19          PRINT *,MYPI
    20
    21          CONTAINS
    22
    23          DOUBLE PRECISION FUNCTION F(X)
    24            DOUBLE PRECISION :: X
    25            F = 4 / (1.0 + X*X)
    26          END FUNCTION F
    27
    28        END PROGRAM PI_EXAMPLE
    29
spe01% xlhpf90 -L/usr/lpp/ssp/css/libip/ -o picycf picyc.f
** pi_example   === End of Compilation 1 ===
1501-510  Compilation successful for file picyc.f.
spe01% setenv MP_PROCS 8
spe01% picycf
 3.14159265442312430
spe01% exit
spe01%
Script command is complete on Wed Apr  9 14:59:50 EDT 1997.
```

```
Script command is started on Wed Apr 23 13:32:36 EDT 1997.spe01% cat -n mypi.c
     1
     2  #include <stdlib.h>
     3  #include <stdio.h>
     4  #include "mpi.h"
     5
     6  #define NUMRECS 10000
     7  #define MASTER 0
     8
     9  MPI_Status status;
    10  MPI_Request request;
    11
    12  double f(double x) {
    13    double y;
    14    y = 4.0/ (1.0 + x*x);
    15    return y;
    16  }
    17
    18
    19  main(int argc, char **argv)
    20  {
    21  double  h,
    22          mypi,
    23          pisum,
    24          piave,
    25          pirecv,
    26          rec_area[NUMRECS];
    27  int     i,n,
    28          mytid,
    29          nproc,
    30          source,
    31          mystart;
    32
    33  MPI_Init(&argc, &argv);
    34  MPI_Comm_rank(MPI_COMM_WORLD, &mytid);
    35  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    36
    37  mystart = NUMRECS/nproc * mytid;
    38
    39  h = 1.0/NUMRECS;
    40  mypi = 0.0;
    41
    42  for(i = 0; i < (NUMRECS/nproc); i++){
    43      rec_area[i + mystart] = h* f(h*((i + mystart) + .5));
    44      mypi += rec_area[i + mystart];
    45  }
    46
    47  if(mytid != MASTER) {
    48     MPI_Send(&mypi, 1, MPI_DOUBLE, MASTER, mytid, MPI_COMM_WORLD);
    49       }
    50
    51  else{
    52     pisum = mypi;
    53     for(n = 1; n < nproc; n++) {
    54        MPI_Recv(&pirecv, 1, MPI_DOUBLE, MPI_ANY_SOURCE, n, MPI_COMM_WORLD
, &status);
    55        pisum += pirecv;
    56        }
    57  }
    58  if(mytid == MASTER)
```



```
    59     printf("Mypi is approximately %19.17lf\n", pisum);
    60  MPI_Finalize();
    61  }               /*End of the program.*/
spe01% mpcc -o mypi mypi.c
spe01% mypi
Mypi is approximately 3.14159265442312741
spe01% exit
Script command is complete on Wed Apr 23 13:33:04 EDT 1997.
```

```
Script command is started on Wed Apr  9 14:57:21 EDT 1997.spe01% cat -n horner.f
     1  PROGRAM HORNER
     2
     3        INTEGER                     :: I, J
     4        INTEGER, PARAMETER          :: N = 4
     5        REAL, DIMENSION (N*N)        :: A
     6        REAL, DIMENSION (0:N-1, N)    :: P
     7        REAL, DIMENSION(0:N)         :: POWERS
     8        REAL, DIMENSION (N)         :: XN, SUMS
     9        REAL                        :: X
    10
    11        !HPF$ PROCESSORS, DIMENSION(N) :: PROCS
    12        !HPF$ DISTRIBUTE(BLOCK) ONTO PROCS :: POWERS, XN, SUMS
    13        !HPF$ DISTRIBUTE(BLOCK, *) ONTO PROCS :: P
    14
    15        X = 2.0
    16
    17        POWERS(0) = 1
    18
    19        !HPF$ INDEPENDENT
    20        DO I = 1, N*N
    21                A(I) = 1
    22        END DO
    23
    24        DO  I = 1, N
    25                POWERS(I) = POWERS(I-1) * X
    26        END DO
    27
    28        XN(:) = 1
    29
    30        DO I = 2, N
    31                XN(I) = XN(I-1)* POWERS(N)
    32        END DO
    33
    34        !HPF$ INDEPENDENT
    35        DO I = 0, N-1
    36                P(I, :) = A(I +1:N*(N-1)+I +1:N) * XN(1:N)
    37        END DO
    38
    39        !HPF$ INDEPENDENT
    40        DO I = 1, N                !GET SUMS OF P[I]'S
    41                SUMS(I) = SUM( P(I-1,:))
    42        END DO
    43
    44        SUMS(:) = SUMS(:) * POWERS(0:N-1)
    45        DO I = 1, N
    46                ANSWER = SUM(SUMS)
    47        END DO
    48
    49        PRINT *, ANSWER
    50
    51    END PROGRAM HORNER
spe01% xlhpf90 -L/usr/lpp/ssp/css/libip/ -o hornerf horner.f
** horner    === End of Compilation 1 ===
1501-510  Compilation successful for file horner.f.
spe01% setenv MP_PROCS 4
spe01% hornerf
 65535.00000
spe01% exit
```



```
Script command is complete on Wed Apr  9 14:58:17 EDT 1997.
```

```
Script command is started on Thu Apr 10 15:17:14 EDT 1997.spe01% cat -n fft.c
     1  #include <stdlib.h>
     2  #include <stdio.h>
     3  #include "mpi.h"
     4  #include <math.h>
     5
     6  #define N 4
     7  #define MASTER 0
     8
     9  const double PI = 3.1415926;
    10
    11  MPI_Status status;
    12  MPI_Request request;
    13
    14  struct Complex{
    15     double im;
    16     double re;
    17     };
    18
    19  int log2(int n)
    20  {
    21      int log = 0;
    22      if(n < 1)
    23          return -1;
    24      if(n == 1)
    25          return  0;
    26      while( n  > 1)  {
    27          n = n/2;
    28          log += 1;
    29          if( n == 1)
    30              return log;
    31       }
    32      return log;
    33  }
    34
    35  void initomegas(struct Complex w[], struct Complex winv[])
    36  {
    37     int i = 0, j;
    38     for(i; i < N; i++){
    39        w[i].re = cos(i*(-2.0/N) *PI);
    40        w[i].im = sin(i*(-2.0/N) *PI);
    41        winv[i].re = cos(i*(2.0/N) *PI);
    42        winv[i].im = sin(i*(2.0/N) *PI);
    43       }
    44  }
    45
    46  int power(int base, int n) {
    47     int x = 1, i;
    48     for (i = 1; i <= n; i++)
    49        x *=base;
    50     return x;
    51  }
    52
    53  int binaryreverse(int p, int n) {
    54      int m, r;
    55      if (n == 0)
    56         r = 0;
    57      else  {
    58         m = power(2, p-1);
    59         if(n < m)
```



```
    60             r = (2* binaryreverse( p-1, n));
    61         else
    62             r = (1+ binaryreverse( p, n-m));
    63      }
    64      return r;
    65  }
    66
    67  void share(struct Complex y[], int mytid, int k, int b, MPI_Datatype m_c
omplex) {
    68  int i,j;
    69  struct Complex temp;
    70  if(mytid != MASTER)
    71     MPI_Send(&y[k], 1, m_complex, MASTER, (N*k) + b, MPI_COMM_WORLD);
    72
    73  for(i = 1; i < N; i++)
    74       if(mytid == MASTER)       {
    75          MPI_Recv(&temp, 1, m_complex, MPI_ANY_SOURCE, (N*i) + b, MPI_CO
MM_WORLD, &status);
    76           y[i] = temp;
    77         }
    78
    79  MPI_Bcast(y, N, m_complex, MASTER, MPI_COMM_WORLD);
    80  }
    81
    82  void fft(struct Complex y[], struct Complex c[], struct Complex w[], int
 mytid, int nproc, MPI_Datatype m_complex)
    83  {
    84  int b = N/2,b2,
    85      i,
    86      k,j,
    87      binrev[N],
    88      exp,
    89      sign,
    90      sub;
    91  double temp;
    92  struct Complex comp;
    93
    94  for(i = 0; i < N/2; i++){
    95     binrev[i] = binaryreverse( log2(N), i);
    96     if(i != binrev[i]){
    97         temp = y[i].re;
    98         y[i].re = y[binrev[i]].re;
    99         y[binrev[i]].re = temp;
   100         }
   101  }
   102
   103  while(b >= 1) {
   104       b2 = N/ (2*b);
   105       k = mytid;
   106       sub = ( mytid % (2*b));
   107       if(( mytid % (2*b)) >= b)
   108          sub -= b;
   109       if(mytid >=  (N/ b2))
   110          sub += mytid - (mytid % (2*b));
   111       sign = (( mytid % (2*b)) < b) *2 -1;
   112       if(mytid < (N/b2))
   113          exp = 0;
   114       else
   115          exp = binaryreverse(b2/2,mytid/ (2*b)) * (N/b2);
   116
```

```
   117         c[k].re = y[sub].re + sign *((y[sub + b].re * w[exp].re) - (y[sub
+ b].im * w[exp].im));
   118         c[k].im = y[sub].im + sign*((y[sub + b].re* w[exp].im) +  (y[sub+
b].im* w[exp].re));
   119
   120         y[k].re = c[k].re;
   121         y[k].im = c[k].im;
   122         share(y, mytid, k, b, m_complex);
   123
   124         b /= 2;
   125   }
   126     for(i = 0; i < N/2; i++) {
   127         c[2*i].re =  y[2*i].re;
   128         c[2*i +1].re =  y[2*i +1].re;
   129         }
   130   }
   131
   132
   133   main(int argc, char **argv)
   134   {
   135   double  temp,
   136           starttime,
   137           endtime;
   138
   139   struct Complex w[N],
   140           winv[N],
   141           y[N],
   142           yinv[N],
   143           c[N];
   144
   145   int     i,j,k,b=1,
   146           binrev[N],
   147           mytid,
   148           nproc, numpr,
   149           source,
   150           mtype,
   151           msgid,
   152           nbytes,
   153           mystart,
   154           rcode,
   155           size;
   156
   157   MPI_Datatype m_complex;
   158   MPI_Datatype type[2] = {MPI_DOUBLE, MPI_DOUBLE};
   159   int blocklength[2] = {1,1};
   160   MPI_Aint disp[2] = {0, sizeof(double)};
   161
   162   MPI_Init(&argc, &argv);
   163   MPI_Comm_rank(MPI_COMM_WORLD, &mytid);
   164   MPI_Comm_size(MPI_COMM_WORLD, &nproc);
   165
   166   MPI_Type_struct(2, blocklength, disp, type, &m_complex);
   167   MPI_Type_commit(&m_complex);
   168
   169   if(mytid == MASTER)
   170      for(i = 0; i < N; i++) {
   171           scanf("%lf", &y[i].re);
   172           y[i].im = 0.0;
   173           c[i].im = 0.0;
   174           yinv[i].im = 0.0;
```



```
   175         }
   176   MPI_Bcast(&y, N, m_complex, MASTER, MPI_COMM_WORLD);
   177
   178   starttime= MPI_Wtime();
   179
   180   initomegas(w, winv);
   181
   182   fft(y, c, w, mytid, nproc, m_complex);
   183   if(mytid == MASTER) {
   184       for(j = 0; j < N; j++)
   185           printf("%5.3lf *x^%d +", c[j].re, j);
   186           c[j].im = 0.0;
   187            }
   188   MPI_Bcast(y, N, m_complex, MASTER, MPI_COMM_WORLD);
   189
   190   fft(c, yinv, winv, mytid, nproc, m_complex);
   191
   192   endtime = MPI_Wtime();
   193
   194   if(mytid == MASTER) {
   195       printf("\nThis is the second equation: \n");
   196       for(j = 0; j < N; j++)
   197           printf("%5.3lf *x^%d +", yinv[j].re/N, j);
   198       printf("\nThis took %5.3f seconds\n", endtime - starttime);
   199            }
   200
   201   MPI_Finalize();
   202   }
spe01% mpcc -o fft.c fftc -lm
cc: 1501-218 file fft.c contains an incorrect file suffix
spe01% mpcc -o fftc fft.c -lm
spe01% fftc -procs 4
1
2
3
4
10.000 *x^0 +-4.000 *x^1 +-0.000 *x^2 +-2.000 *x^3 +
This is the second equation:
1.000 *x^0 +2.000 *x^1 +3.000 *x^2 +4.000 *x^3 +
This took 0.002 seconds
spe01% !!
fftc -procs 4
99.99
53.10
40.22
965.33
1158.640 *x^0 +-852.460 *x^1 +972.000 *x^2 +-878.220 *x^3 +
This is the second equation:
99.990 *x^0 +53.100 *x^1 +40.220 *x^2 +965.330 *x^3 +
This took 0.005 seconds
spe01% exit
spe01%
Script command is complete on Thu Apr 10 15:18:41 EDT 1997.
```

```
Script command is started on Thu Apr 10 15:20:41 EDT 1997.spe01% cat -n fft.f
     1
     2       PROGRAM FFT
     3
     4       DOUBLE PRECISION, PARAMETER  :: PI = 3.1415926535
     5       INTEGER, PARAMETER           :: N = 4
     6       DOUBLE COMPLEX, DIMENSION(0:N-1)  :: Y, W, WINV
     7       DOUBLE PRECISION, DIMENSION(0:N-1) :: IN
     8       DOUBLE COMPLEX               :: TEMP
     9
    10       !HPF$  PROCESSORS, DIMENSION(N)    :: PROCS
    11       !HPF$  DISTRIBUTE (CYCLIC) ONTO PROCS :: Y
    12
    13      DO I = 0, N-1
    14         READ *, IN(I)
    15         Y(I) = (IN(I), 0)
    16      END DO
    17
    18      CALL INITOMEGAS(W, WINV)
    19
    20      CALL FASTFOURIER( Y, W)
    21
    22      DO I = 0, N-1
    23          PRINT *, Y(I)
    24      END DO
    25
    26      CALL FASTFOURIER( Y, WINV)
    27
    28      DO I = 0, N-1
    29          PRINT *, Y(I)/N
    30      END DO
    31
    32
    33        CONTAINS
    34
    35
    36      INTEGER FUNCTION LOG2(L)
    37      LOG = 0
    38      IF(L .LT. 1)  THEN
    39         LOG2 = 0
    40      END IF
    41      IF(L == 1) THEN
    42         LOG2 = 0
    43      END IF
    44      DO J = 1,L/2
    45         L = L/2
    46         LOG = LOG + 1
    47         IF( L == 1)  THEN
    48            LOG2 = LOG
    49         END IF
    50      ENDDO
    51      LOG2 = LOG
    52      END FUNCTION LOG2
    53
    54
    55      SUBROUTINE INITOMEGAS(W, WINV)
    56      DOUBLE COMPLEX, DIMENSION(0:N-1), INTENT(INOUT)    :: W, WINV
    57
    58      !HPF$ INDEPENDENT
    59      DO I = 0, N -1
```



```
    60         W(I) = (COS(I*(-2.0/N)*PI), SIN(I*(-2.0/N)*PI))
    61         WINV(I) = (COS(I*(2.0/N)*PI), SIN(I*(2.0/N)*PI))
    62      ENDDO
    63      END SUBROUTINE INITOMEGAS
    64
    65
    66      INTEGER FUNCTION POWER(L, J)
    67      INTEGER, INTENT(IN)    :: J,L
    68      INTEGER :: X = 1
    69      IF(J == 0)   THEN
    70         POWER = 1
    71      END IF
    72      IF(J .LT. 0)  THEN
    73         POWER = 0
    74      END IF
    75      DO I = 1, J
    76         X = X * L
    77      ENDDO
    78      POWER = X
    79      END FUNCTION POWER
    80
    81
    82      RECURSIVE FUNCTION REVERSE( P, L)   RESULT(ANS)
    83      INTEGER :: P, L
    84      INTEGER   :: ANS
    85      INTEGER :: M
    86      IF(L == 0) THEN
    87         ANS = 0
    88      ELSE
    89         M = POWER(2, P-1)
    90         IF(L .LT. M) THEN
    91             ANS = 2* REVERSE(P-1, L)
    92         ELSE
    93             ANS = 1 + REVERSE(P, L-M)
    94         ENDIF
    95      ENDIF
    96      RETURN
    97      END FUNCTION REVERSE
    98
    99
   100      SUBROUTINE FASTFOURIER(Y, W)
   101      DOUBLE COMPLEX, DIMENSION(0:N-1), INTENT(INOUT) :: Y
   102      DOUBLE COMPLEX, DIMENSION(0:N-1), INTENT(IN) :: W
   103      INTEGER                    :: I, J, B, B2
   104      INTEGER                 :: SUB, EXP, MYTID, K, SIGN
   105      DOUBLE COMPLEX          :: TEMP
   106      DOUBLE COMPLEX, DIMENSION(0:N-1)   :: C
   107      INTEGER, DIMENSION(3,0:7)      :: BINREV
   108
   109      BINREV(1,0) = 0
   110      BINREV(1,1) = 1
   111      BINREV(2,0) = 0
   112      BINREV(2,1) = 2
   113      BINREV(2,2) = 1
   114      BINREV(2,3) = 3
   115      BINREV(3,0) = 0
   116      BINREV(3,1) = 4
   117      BINREV(3,2) = 2
   118      BINREV(3,3) = 6
   119      BINREV(3,4) = 1
```

```
   120      BINREV(3,5) = 5
   121      BINREV(3,6) = 3
   122      BINREV(3,7) = 7
   123      DO I = 0, (N/2) -1
   124         IF( .NOT. (I == BINREV(LOG2(N), I)))  THEN
   125             TEMP = Y(I)
   126             Y(I) = Y(BINREV(LOG2(N), I))
   127             Y(BINREV(LOG2(N),I)) = TEMP
   128         END IF
   129      ENDDO
   130
   131      B = N/2
   132      DO WHILE( B >= 1)
   133         B2 = N/ (2*B)
   134
   135         DO MYTID = 0, N-1
   136            K = MYTID
   137            L = MOD(MYTID,   (2*B))              !32
   138            SUB = L
   139            IF (SUB .GE. B) THEN
   140                SUB = SUB -B
   141            END IF
   142            IF(MYTID .GE. (N/B2)) THEN
   143                SUB = SUB + MYTID - L
   144            END IF
   145            IF (L .LT. B) THEN
   146               SIGN = 1
   147            ELSE
   148               SIGN = -1
   149            END IF
   150            IF(MYTID .LT. (N/B2)) THEN
   151               EXP = 0
   152            ELSE
   153               EXP = (N/ B2) * BINREV(B2/2, MYTID/(2*B))
   154            END IF
   155            C(K) = Y(SUB) + SIGN*(Y(SUB +B)*W(EXP))
   156         END DO
   157
   158         Y(:) = C(:)
   159         B = B/2
   160      END DO
   161      END SUBROUTINE FASTFOURIER
   162
   163      END PROGRAM FFT
spe01% xlhpf90 -L/usr/lpp/ssp/css/libip/ -o fftf fft.f
** fft   === End of Compilation 1 ===
1501-510  Compilation successful for file fft.f.
spe01% fft -procs 4
fft: Command not found.
spe01% fftf -procs 4
1
2
3
4
 (10.0000000000000000,0.00000000000000000E+00)
 (-4.0000000000000000,0.00000000000000000E+00)
 (-0.37747582837255324E-14,-0.87422780003724745 4E-07)
 (-1.9999999999999623,0.87422780003724745 4E-07)
 (1.0000000000000000,0.00000000000000000E+00)
 (2.0000000000000000,0.00000000000000000E+00)
```



```
 (3.0000000000000000,-0.43619100791813562 1E-24)
 (4.0000000000000000,0.43619100791813562 1E-24)
spe01% !!
fftf -procs 4
99.99
53.10
36.22
965.33
 (1154.6400000000010,0.00000000000000000E+00)
 (-848.46000000000036,0.00000000000000000E+00)
 (975.999999999996476,-0.81225379129260696 3E-04)
 (-882.219999999996503,0.81225379129260696 3E-04)
 (99.9900000000000091,0.00000000000000000E+00)
 (53.1000000000000227,0.00000000000000000E+00)
 (36.2199999999999704,-0.33279123183482851 6E-20)
 (965.330000000000155,0.33279123183482851 6E-20)
spe01% exit
Script command is complete on Thu Apr 10 15:22:42 EDT 1997.
```