A Brief Introduction to MPI

Parallel Computational Models

“A computational view is a conceptual view of what types of operations are available to the program.

It does not include the specific syntax of a particular programming language, and it is (almost) independent of the undrlying hardware that supports it.

That is any of the models that we discuss can be implemented on any modern parasllel computer, given a little help from the operating system. The effectiveness of such implementation, however, depends on the gap between the model and the machine.”

--*From USING MPI, W. Gropp, E. Lusk, and A. Skejellum. MIT Press*

Message Passing

“The message-passing model posits a set of processes that have only local memory but are alble to communicate with other processes by sending and receiving messages.

It is a defining feature of the message-passing model that the data transfer from the local memory of one process to the local memory of another requires operations to be performed by both processes.”

--*From USING MPI, W. Gropp, E. Lusk, and A. Skejellum. MIT Press*

Remote memory operations

“Halfway between the sahred memory model, where processes access memory without knowing whether they are triggering reemote communication at the hardware level, and the message-passing model, where both the local and remote processes must participate, is the remote memory operation model.

This model is typified by *put* and *get* operations.”

--*From USING MPI, W. Gropp, E. Lusk, and A. Skejellum. MIT Press*

Advantages of the message-passing model

- Universality. Matches the hardware of most of today’s parallel supercomputers, as well as workstation networks that are beginning to compete with them. Where the machine supplies extra hardware to support a share-memory model, the message passing model can take advantage of this model to speed data transfer.

- Expressivity. Message-passing has been found to be a useful and complete model in which to express parallel algorithms. It provides the control missing from the data parallel and compiler-based models. Some find its atropomorphic flavor useful in formulating parallel algorithms. It is well suited to adaptive, self scheduling algorithms and to programs that cna be made tolerant of the inbalance in process speeeds found on shared networks.

- Ease of debugging. It can be argued that debugging is easier in the message-pasing paradigm than in the sahred-memory model. This is because one of the most common causes of error is unexpected overwriting of memory. The message-passing model, by controlling memory references more explicitly than any of the other models (only one process has direct access to any memory location), makes it easier to locate erroneous memory reads and writes.

- Performance. The most compelling reason that message padding will remain a permanent part of the parallel computing environment is performance. Even on shared-memory computers, use of the message-passing model can improve operformance by providing more programmer control of data locality in the memory hierarchy.

A first MPI program

This program will compute the value of $\pi$by numerical integration.

$$\int_0^1 \frac{1}{1+x^2}dx = arc\tan(x)\Big|_0^1 = arc\tan(1) - arc\tan(0) = \frac{\pi}{4}$$

Not a very effective way of computing p, but a good academic exercise.

The strategy is to divide the interval [0,1] into n subintervals and approximate the integral by the sum of n rectangles, each associated to one of the subintervals.

Thus, each rectagle will have width 1/n, and its height will be the value of the function $1/(1+x^2)$ in the middel of each subinterval.

MPI programs usually follow the SPMD model. The overall organization of our example program is as follows:

```
Intialize
Get total number of processes (numprocs)
Get the number of THIS process (myid)
The master process reads n, the number of subintervals
The master process broadcasts n
h=1./n
localpi=0
do i=myid+1,n,numprocs
   x = h*i-h/2
   localpi = localpi + (1/(1+x**2))*h
end do
Perform a sum of all localpi from all processes
The master prints the result of the sum
```

Initialize

```
      call MPI_INIT(ierr)
```

This call is required in every MPI program and must be the first MPI call. It establishes the MPI environment.

Only one invocation of MPI_INIT can occur in each program execution.

Its only argumetn is an error code. Every Fortran MPI routine returns an error code in its last argument, which is either MPI_SUCCESS (a constant defined in the mpif.h file) or an implementation-defined error code. We will not test the error codes in this example.

Get process number

```
      call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)
```

In MPI, processes can be divided into groups. If a group contains n processes, then its processes are identified within the group by ranks, which are integers from 0 to n-1.

All MPI communications are associated with a communicator that describes the context [to be defined later] and an associated group.

In this program we use the default communicator (MPI_COMM_WORLD [defined in the file mpif.h]) that defines one group containing the set of all processes.

Get total number of processes

```
call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ierr)
```

The master process reads n

```
if (myid .eq. 0) then
   print *, ‘enter the number of intervals: (0 quits)’
   read (*,*) n
end if
```

The master process broadcasts n

```
      call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD, ierr)
```

This call results in every process (in the groups associated with the communicator given in the fifth argument) ending up with a copy of n.

The data to be communicated is decribed by the address (n), the data type (MPI_INTEGER), and the number of items (1). The proces with the original copy is specified by the fourth argument (0).

Thus, after the call to MPI_BCAST, all processes have n and their own identifiers (i.e. their myid) which is enough information for each one to compute its contribution, mypi.

Sum all localpi variables

```
call MPI_REDUCE(localpi,pi,1,MPI_DOUBLE_PRECISION,
                MPI_SUM,0,MPI_COMM_WORLD, ierr)
```

The first two arguments identify the source and result addresses, repectively. The data being collected consists of 1 (third argument) item of type MPI_DOUBLE_PRECISION (fourth argument). The operation is addition (MPI_SUM, the next argument), and the result of the operatin is to bel palced in pi on the process with rank 0 (fifth argument).

The first two arguments of MPI_Reduce mut not overlap (i.e. must be different variables or sections of an array).

A self-scheduling example: matrix-vector multiplication

This example illustrates explicit point-to point communication, and at the same time illustrates one of the most common parallel algorithms prototypes: the self scheduling, or master worker algorithm.

This example does not illustrate the best way to parallelize this particular numerical computation, but rather it illustrates MPI send and receive operations in the context of a fundamental type of parallel algorithm.

In this example, the master (process 0) executes different code from the slaves (processes 1 to numprocs-1).

The objective is to compute **c** = **Ab**

```
master:
broadcast vector (b) to all slaves
send  first numprocs-1 rows of A to slaves (one to each)
numsent = numprocs-1
do number of rows times
   receive an element of c (from process p)
   if numsent < number of rows then
      send another row of A to p
   else
      send finish messsage to p
   end if
   numsent = numsent + 1
end do

slave:
receive b
top: receive row or finish message
if finish message then
   finish
else
   compute dot product of row time b
   go to top
end if
```

master: broadcast vector b:

```
      call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master,
                     MPI_COMM_WORLD, ierr)
```

Notice that cols in the number of elements of b

slave: receive vector b:

```
      call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master,
                     MPI_COMM_WORLD, ierr)
```

master: send first numprocs-1 row of A:

```
      do 40 i=1, numprocs-1
         do 30 j=1,cols
            buffer(j)=a(i,j)
 30   continue
      call MPI_SEND(buffer,cols, MPI_DOUBLE_PRECISION,i,
                         i, MPI_COMM_WORLD, ierr)
 40   continue
```