Introduction to MPI

SHARCNET
Brock University
March 30, 2004

Brief History of Parallelism

- different approaches to parallelism: threading, distributed...
- 15/20 years ago vendors had proprietary libraries and compiler directives
- MPI 1.0 standard created in 1994 between industry and users to have a portable, open way of writing parallel software

MPI Features

- most general and flexible approach, giving the most freedom to the application programmer
- scales the best to many processors
- portable across architectures and compilers
- unfortunately, also the most work, requiring significant rewriting of serial code and entirely new algorithms

Structure of MPI

- MPI appears as a library of functions or subroutines to the programmer, accessible from either C/C++ or Fortran 77/90
- the basic operation in MPI is to send a message containing data, from one process to another
- while there is a single API, there are multiple implementations: MPICH, LAM, vendor-supplied etc

MPI Coding1

- you need to make three subroutine calls to initialize MPI for your program:

```
MPI_INIT(ierr)

MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ierr)

MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)
```

where ierr, numprocs and myid are integers

- numprocs is the number of processors, and myid is my id number within this group of processors, 0...np-1

MPI Coding 2

- the actual message passing can be divided into two main categories:

  a) collective communications (broadcast, reduction, scatter/gather)

  b) point-to-point communications

- to finish an MPI program, one calls the subroutine MPI_FINALIZE(ierr)

Parallel “ Hello, World”

```
program main

include 'mpif.h'

integer ierr,myid,numprocs

call MPI_INIT(ierr)

call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ierr)

call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)

write(*,*) 'Hello from process',myid

call MPI_FINALIZE(ierr)

stop

end
```

MPI Coding 3

- broadcast involves one process sending the same message to all the other processes:

```
MPI_BCAST(buffer,size,type,broadcaster,communicator,ierr)
```

- the type can be one of MPI_INTEGER, MPI_REAL, MPI_COMPLEX, MPI_DOUBLE_PRECISION etc
- MPI also permits user-defined data types

MPI Coding 4

- reductions take an element from each process and perform an operation on it, like addition or minimization:

```
MPI_REDUCE(argument,result,size,type,operation,location,communicator,ierr)

MPI_ALLREDUCE(argument,result,size,type,operation,communicator,ierr)
```

- the operation can be one of MPI_SUM, MPI_PROD, MPI_MAX, MPI_MIN...
- again, MPI permits users to define their own collective operations

MPI Coding 5

- point-to-point communications are carried out using

```
MPI_SEND(buffer,size,type,destination,tag,communicator,ierr)

MPI_RECV(buffer,size,type,source,tag,communicator,status,ierr)
```

- tag is a programmer-set integer used to distinguish one message from another, and status an integer array which can be used to query the status of a message

MPI “ Ping pong”

- this Fortran 77 program simply passes an integer back and forth between the pool of processes
- uninteresting scientifically, but it does help to illustrate the point-to-point MPI functions, and can be used to get a rough estimate of the network latency

Parallel Design

- the most difficult step is usually not the writing of the parallel code, but coming up with a parallel algorithm that is efficient, scalable and produces correct output
- existence of a working serial version can be of some assistance, but often using MPI will require a complete rewrite (and rethink) of the code

Parallel Design 2

- first step lies in identifying bottlenecks in the serial code/algorithm, since it is these parts of the code that we want to improve by parallelization, for instance a loop:

```
for(i=0; i<N; ++i) {
   // Lots of computations
}
```

- we might consider trying to break this loop up:

Parallel Design 3

```
stride = N/numprocs;
l_index = myid*stride;
U_index = (1+myid)*stride;
for(i=l_index; i<u_index; ++i) {
  // Do computations
}
```

- having done this, need to consider the kind of data dependency inside the loop

Parallel Design 4

- it's highly probable that some data used in the loop will need to be shared among the different processors
- this analysis of data dependencies is what will normally tell us about the messages that need to be passed among processors
- in this example, we are breaking up the problem geometry (e.g. the number of particles or mesh points) into pieces

Parallel Design 5

- an alternative strategy for some problems is to use a master/slave model
- in this case, a master process keeps track of the computation as a whole and hands out “work units” to slave processes
- this is ideally suited to problems that can be naturally broken up into such units, like Monte Carlo algorithms

Performance

- the MPI calls in your program will always be far slower than floating point operations, indexing of arrays or other “local” activity
- so, in general try to minimize the amount of communication between processes
- when communicating, try to avoid many small messages, and opt for a few large messages
- try to balance the computational burden across all the processes

Performance 2

- scaling refers to how well your program behaves as the number of CPUs is increased
- ideally, one would have

runtime(n) = serial_runtime/n

for n CPUs

- in reality, this never happens, because of such things as communications overhead and the fact that part of the program remains serial

Performance 3

- to ensure your program scales well, you need to try to do as much “local” work as possible, per MPI call
- latency vs. bandwidth
- interconnect technologies: Ethernet, Myrinet, Quadrics/ELAN
- debugging: TotalView on idra
- use non-blocking MPI calls

Future of MPI

- very widely supported standard for parallel programming, more so than PVM
- MPI-2 standard was created in 1997, but so far implementations are lacking, especially for some difficult functionality