Tutorial on Parallel Programming

Center for Research Computing
Charles F. Vardeman II, Ph.D

November 5, 2009
Topics

- General Concepts
- Parallel Toolkits
- Introduction to MPI
- Parallel Examples for MPI
- A few advanced topics
- Conclusions
Parallelism

- Computational Problems too large to be solved in a serial fashion.
- Goal: Use a large number of processors $P$ to achieve the same result hopefully $P$ times faster. (More on that later).
- Some problems are easily decomposed into independent parallel segments. “Trivially Parallel”
Granularity of Parallelism

- Instruction Level (Processor-Compiler, Pipelining, etc)
- Thread Level
- Machine Level
Computational Architectures
(Flynn’s Taxonomy)

Single Instruction, Single Data
- SISD
- Instruction Pool
- Data Pool → PU

Multiple Instruction, Single Data
- MISD
- Instruction Pool
- Data Pool → PU → PU

Single Instruction, Multiple Data
- SIMD
- Instruction Pool
- Data Pool → PU → PU

Multiple Instruction, Multiple Data
- MIMD
- Instruction Pool
- Data Pool → PU → PU → PU
Performance

Time to solve a problem using P processors: \( T(P) \)

Speed up: \( S(P) = \frac{T(1)}{T(P)} \)

Parallel Efficiency: \( E(P) = \frac{S(P)}{P} \)

Example:

Suppose our program consists of two parts \( A \& B \).

Serial: \( A \) takes 1 minute and \( B \) takes 24 minutes.

Using 12 processors: \( A \) takes 0.1 minutes and \( B \) takes 12 minutes

\[ S(12) = \frac{1 + 12}{0.1 + 12} = 2.07 \quad E(12) = \frac{2.07}{12} = 17\% \]

- Important concept: Look to efficiently parallelize the parts that take the largest percentage of time!
Amdahl’s Law (1967)

Gives the maximum expected improvement to an overall system when only part of the system is improved.

$$S(P) \leq \frac{1}{S + \frac{1-S}{P}} \leq \frac{1}{S}$$
Parallel Software Engineering

- How much time is spent in each program segment?
- Are there ‘embarrassingly parallel’ segments?
- How much data communication will be necessary?
- To what level of granularity do you want to parallelize your code?
- When do you synchronize data among various processors?
- Load Balancing?
- Can you use a parallel library?
Parallel Libraries

linear and nonlinear solver for PDEs

Fastest Fourier Transform in the West:  http://www.fftw.org/

Cactus:  http://www.cactus.org framework for PDE solutions

SuperLU:  http://crd.lbl.gov/~xiaoye/SuperLU/ solution of large, sparse, nonsymmetric systems of linear equations


ScaLAPACK: parallel linear algebra  http://www.netlib.org/scalapack/

Pblas: http://www.netlib.org/scalapack/pblas_qref.html

Scalable Parallel Random Number Generator library:  http://sprng.cs.fsu.edu/

Suite of Nonlinear and Differential/ALgebraic equation Solvers:  http://www.llnl.gov/CASC/sundials/

Gnu Scientific Library:  http://www.gnu.org/software/gsl/
For parallel (and serial) debugging and profiling

- Tau: [http://www.cs.uoregon.edu/research/tau/home.php](http://www.cs.uoregon.edu/research/tau/home.php)
- Vampir: [http://www.vampir-ng.de/index.html](http://www.vampir-ng.de/index.html)
**Debugging and Profiling Tools**

**Prerequisite:** hardware counters API
- Performance Counter Library: [http://www.fz-juelich.de/zam/PCL/](http://www.fz-juelich.de/zam/PCL/)

**Other Tools:**
- HPCToolkit: [http://www.hipstersoft.rice.edu/hpctoolkit/](http://www.hipstersoft.rice.edu/hpctoolkit/)
- Perfsuite: [http://perfsuite.ncsa.uiuc.edu/](http://perfsuite.ncsa.uiuc.edu/)
Message Passing Interface implements the message passing model MPI-1 (1994). It's a library of functions (C) or subroutines (Fortran) to perform data communication and synchronization for programs running on parallel machines. MPI documents: [http://www.mpi-forum.org/docs/](http://www.mpi-forum.org/docs/)

**MPI main advantages are portability of the parallel code and its flexibility**

**MPI Implementations:**
- Infiniband MPI  MVAPICH ([http://mvapich.cse.ohio-state.edu/](http://mvapich.cse.ohio-state.edu/))

**Other solutions:**
- Linda [http://www.lindaspaces.com/about/index.html](http://www.lindaspaces.com/about/index.html)
- High Performance Fortran, Fortress, Unified parallel C, Cilk, ...
- Do not parallelize!
Introduction to MPI

MPI terminology:

- process – one instance of the parallel program
  - usually, 1 process = 1 processing core
- communicator – subset of the active processes
  - MPI_COMM_WORLD – default communicator that contains all of the active processes
  - each process in a communicator has a unique id which are numbered 0, 1, ..., size – 1
  - if a process belongs to multiple communicators, its id can be different
Introduction to MPI

- **topology** – (logical) arrangement of the processes inside of a communicator
  - e.g. 9 processes arranged in a 3 x 3 grid
- **communication mode** – way that data is communicated between the processes
  - e.g. blocking & non-blocking, buffered & non-buffered, broadcast & point-to-point
- **collective/reduction/scatter operation** – single MPI function that can collect/reduce/scatter data using all processes in a communicator
Basic MPI

General structure of MPI program:

• MPI initialization: MPI_INIT (ierr) / int MPI_Init(int *argc, char ***argv);

• MPI_COMM_WORLD and communicators
  
  MPI_COMM_RANK(comm, rank)
  MPI_COMM_SIZE(comm, size)

• MPI Communication:
  
  MPI_SEND(address, count, datatype, destination, tag, comm)
  MPI_RECV(address, maxcount, datatype, source, tag, comm, status)

• MPI cleanup: MPI_FINALIZE(ierr)
Point-to-point communications

It involves 2 processes

Most basic commands:
• MPI_Send
• MPI_Recv

Message = data + envelope

Envelope (extra information to the data that the application needs to transmit):
• Rank of the receiver to distinguish messages received from different processes
• Rank of the sender
• Tag: user-specified integer to distinguish messages received from a single process
• Communicator (predefined MPI_COMM_WORLD=all processes running)
Communication envelope returned from MPI_RECV as status

Query of status argument (if wild card as been used in arguments in the receive call)

• Source process (rank): STATUS(MPI_SOURCE) (MPI_ANY_SOURCE argument)

• Message tag: STATUS(MPI_TAG) (MPI_ANY_TAG argument)

• Datatype number: MPI_GET_COUNT (status,datatype,count) (count argument)

Rules of pt2pt communications:

• Order is preserved: process 0 sends two message to 1. Process 1 posts 2 receive calls. The messages are guaranteed to be received in the same order they were sent

• A matching send and receive can not remain indefinitely open: process 0 posts a send, process 1 posts a matching receive. One of them will eventually complete.

• Datatype matching rules
```c
#include <mpi.h>

int main(int argc, char **argv)
{
    /*
     ... serial code ...
    */
    // initialize MPI
    MPI_Init(&argc, &argv);
    /*
     ... MPI code ...
    */
    // finalize MPI
    MPI_Finalize();
    /*
     ... serial code ...
    */
    return 0;
}
```
#include <mpi.h>

int main(int argc, char **argv)
{
    // initialize MPI
    MPI_Init(&argc, &argv);

    printf("Hello world!\n");

    MPI_Finalize();

    return 0;
}
Building a MPI application

```
$ module avail

3.2.6

$ module load ompi/1.3.2-intel

$ which mpicc

$ /afs/crc.nd.edu/x86_64_linux/openmpi/1.3.2/intel/bin/mpicc

$`

Running a MPI application
MPI Send-Receive

4 communications modes:
• Standard send: either synchronous or buffered MPI_SEND
• Synchronous send: only completes when the receive has complete (the receiving process sends back an acknowledgment) MPI_SSEND
• Buffered send: always completes (unless error), irrespective of whether the receive has completed MPI_BSEND
• Ready send: always completes (unless error), irrespective of whether the receive has completed MPI_RSEND
Deadlock Solution

```fortran
program example3_a

implicit none

!-- Include the mpi header file
include 'mpif.h'
integer ierr,myid,numpocs
integer irc,tag
integer status(MPI_STATUS_SIZE)
real a,b

!-- Initialize MPI
call MPI_INIT(ierr)

!-- Who am I? --- get my rank=myid
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

!-- How many processes in the global group?
call MPI_COMM_SIZE(MPI_COMM_WORLD, numpocs, ierr)
print *, "Process ", myid, " of ", numpocs, " is alive"

if (myid.eq.0) then
  a=0.
b=1.
else
  a=1.
b=0.
end if

!-- Exchange messages
tag=1
if (myid.eq.0) then
call mpi_send(a,1,mpi_real,1,tag(MPI_COMM_WORLD),ierr)
call mpi_recv(b,1,mpi_real,1,tag(MPI_COMM_WORLD),status,ierr)
elseif (myid.eq.1) then
call mpi_recv(a,1,mpi_real,0,tag(MPI_COMM_WORLD),status,ierr)
call mpi_send(b,1,mpi_real,0,tag(MPI_COMM_WORLD),ierr)
end if

!-- Finalize MPI
call MPI_FINALIZE(irc)

stop
end
```

Wednesday, November 11, 2009
Even Better Solution
Non blocking Communication

- The call to send/receive may return before the operation completes
- It can prevent deadlock and slowness (latency hiding)

Standard send: MPI_ISEND(message, count, datatype, dest, tag, comm, request)
After initiating call, it continues with computations that do not alter the send buffer
Synchronous send: MPI_ISSEND
Buffered send: MPI_BSEND
Ready send: MPI_RSEND
Receive: MPI_Irecv After posting the call, the receiving process does other computations until it needs the received data.
Test for Completion
(checkpointing)

Blocking Communication:
Wait type: it blocks until the communication has completed
MPI_WAIT(request,status)
Blocking=Non-blocking+Wait
Test type:returns TRUE or FALSE depending on the communication status
MPI_TEST(request,flag,status)

Non-Blocking
Test for completion of all of the communications
MPI_WAITALL(count,array_requests,array_of_statuses) / MPI_TESTALL
(count,array_requests,flag,array_of_statuses)
Test for completion of any of the communications
MPI_WAITANY(count,array_requests,index,status) / MPI_TESTANY
(count,array_requests,index,flag,status)
Test for some of the communications
MPI_WAITSOME(count,array_requests,outcount,array_indices,array_statuses)
MPI_TESTSOME(count,array_requests,outcount,array_indices,flag,array_statuses)
Collective Communication

It involves every processes in a specified communicator

- A message is an array of one particular datatype
- Datatypes must match between send and receive
- Completion means buffer can be used or be-reused
- It cannot be picked up by a pt2pt receive
- It may or may not synchronize the processes involved
- No tag
- The sent message must fill the receive buffer

MPI collective communication routines:

- Barrier synchronization across all processes MPI_BARRIER(comm,ierror)
- Broadcast from one process to all other processes
- Global reduction operations (sum, min, max, user-defined operations)
- Gather data from all processes to one process
- Scatter data from one process to all processes
- Other operations where all processes receive the same result from a gather, scatter or reduction
Collective Communication

MPI_BCAST(buffer,count,datatype,root,comm)

MPI_REDUCE(sendbuf,recvbuf,count,datatype,op,root,comm)

MPI_GATHER(sendbuffer,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm)

MPI_ALLGATHER(sendbuffer,sendcount,sendtype,recvbuf,recvcount,recvtype,comm)

MPI_SCATTER(sendbuffer,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm)

MPI_REDUCE(sendbuf,recvbuf,count,datatype,op,root,comm)
Reduce

- All processes in a communicator provide data
- The result of the operation performed is returned to root

Reduce:
$ mpif90 example5_c.f -o example5_c
$ mpirun -np 4 -machinefile mf example5_c
P: 0 prod result is 24
Scatter

Divides a big array into smaller parts equal to number of processes sending each a piece in rank order.

Scatter:

$ mpif90 example5_a.f -o example5_a
$ mpirun -np 4 -machinefile mf
example5_a

P: 0 mine is 24.00000
P: 3 mine is 27.00000
P: 2 mine is 26.00000
P: 1 mine is 25.00000
Gather

- Receives data stored in small arrays from all processes and combines into receive array in rank order.

**Gather:**

```bash
$ mpif90 example5_b.f -o example5_b
$ mpirun -np 4 -machinefile mf example5_b
```

<table>
<thead>
<tr>
<th>PE:</th>
<th>0</th>
<th>param(1)</th>
<th>23.00000</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE:</td>
<td>0</td>
<td>param(2)</td>
<td>24.00000</td>
</tr>
<tr>
<td>PE:</td>
<td>0</td>
<td>param(3)</td>
<td>25.00000</td>
</tr>
<tr>
<td>PE:</td>
<td>0</td>
<td>param(4)</td>
<td>26.00000</td>
</tr>
</tbody>
</table>
Questions
Acknowledgments

- JC Ducom
- Jonathan Hauenstein
- Wikipedia (for nice graphic of computational architecture)