Introduction to Parallel Programming and MPI

Paul Edmon
FAS Research Computing
Harvard University

For copy of slides go to:
http://software.rc.fas.harvard.edu/training
Outline

• What is parallel computing?

• Theory

• Message Passing Interface
Parallel vs. Serial

• Serial: A logically sequential execution of steps. The result of next step depends on the previous step.

• Parallel: Steps can be contemporaneously and are not immediately interdependent or are mutually exclusive.
Microprocessor Transistor Counts 1971-2011 & Moore's Law

The curve shows transistor count doubling every two years.
High Performance Computing (HPC)

- Goal: Leverage as much computer power as possible with as much efficiency as possible to solve problems that cannot be solve by conventional means

- Sub Types
  - Algorithm and Single Chip Efficiency
  - High Throughput Computing
  - High I/O Computing
  - Tightly Coupled Parallel Computing
Scaling

- **Weak Scaling**
  - Keep the size of the problem per core the same, but keep increasing the number of cores.
  - Ideal: Amount of time to solution should not change

- **Strong Scaling**
  - Keep the total size of the problem the same but keep increasing the number of cores.
  - Ideal: Time to completion should scale linearly with the number of cores

- **Reasons for Deviation**
  - Communications Latency
  - Blocking Communications
  - Non-overlapped communications and computation.
  - Not enough computational work
Amdahl’s Law

• The maximum you can speed up any code is limited by the amount that can be effectively parallelized.

• In other words: You are limited by the mandatory serial portions of your code.
Types of Parallelization

- SIMD
- Thread
- Multinode
SIMD

• Single Instruction Multiple Data

• Vectorization
  – \( A(\cdot)=B(\cdot)+C(\cdot) \)

• Processors natively do this, compilers optimize for it.
  – SSE (Streaming SIMD Extensions): 128 bit register, \( a=a+b \)
  – AVX (Advanced Vector Extensions): 128 bit register, \( a=a+b \rightarrow 256 \text{ bit register } a=b+c \)

• Note on Optimization Flags:
  – \(-O0\): No optimization
  – \(-O1\): Safe optimization
  – \(-O2\): Mostly Safe optimization
  – \(-O3\): Aggressive optimization

• Always check your answers after your optimize to make sure that you get the same answer back. This is true for any time you recompile or build on a new system. If there are differences make sure they are minor with respect to your expected code outcome.
Thread

- Single Node, program is broken up into threads
- Libraries: OpenMP, pThreads, Cilk
- SMP: Symmetric multiprocessing
- Threads have access to the same memory pool and thus do not have to communicate
Multinode

- Program is broken up into ranks, each rank runs a part of the code
- Ranks run on multiple nodes
- Ranks do not share memory so they must communicate with each to share information
- Libraries: MPI
Is my code parallelizable?

• Does it have large loops that repeat the same commands?

• Does your code do multiple tasks that are not dependent on one another? If so is the dependency weak?

• Can any dependencies or information sharing be overlapped with computation? If not is the amount communications small?

• Do multiple tasks depend on the same data?

• Does the order of operations matter? If so how strict does it have to be?
Examples

• Computational Fluid Dynamics
• N-Body and NAMD
• Radiative Transfer and Image Processing
• Markov Chain Monte Carlo
• Embarrassingly Parallel Work
General Guidelines for Parallelization

• Is it even worth parallelizing my code?
  – Does your code take an intractably long amount of time to complete?
  – Do you run single large models or do statistics on multiple small runs?
  – Would the amount of time it take to parallelize your code be worth the gain in speed?

• Parallelizing Established Code vs. Starting from Scratch
  – Established Code: May be easier/faster to do, but may not give good performance or scaling
  – Start from Scratch: Takes longer but will give better performance, accuracy, and gives opportunity to turn a black box code into a code you understand
General Guidelines for Parallelization

• Test, test, test, etc.

• Use Nonblocking Communications as often as possible

• Overlap Communications with Computation

• Limit synchronization barriers
General Guidelines for Parallelization

• Limit Collective Communications

• Make messages small
  – Only send essential information

• Make sure messages are well packaged
  – Do one large send with data in a buffer rather than multiple sends, but don’t make the send too large.

• Use MPI_Iprobe to grease the wheels of nonblocking communications
General Guidelines for Parallelization

• Always post nonblocking receives before sends

• Watch out for communications deadlocks

• Be careful of your memory overhead

• Be careful of I/O
  – Avoid having all the cores write to disk at once
  – Alternately don’t have all I/O go through one rank.
General Guidelines for Parallelization

• Do as much as is possible asynchronously

• See if some one has parallelized a code similar to yours and look at what they did

• Beware of portions of the code that depend on order of operations

• Avoid gratuitous IF statements

• Do not use GOTO unless absolutely necessary
General Guidelines for Parallelization

• KISS: Keep it simple stupid.

• Print statements are your friend for debugging

• So is replicating the problem on a small number of ranks

• Think at scale
Message Passing Interface

• MPI standard: Set by MPI Forum

• Current full standard is MPI-2
  – MPI-3 is in the works which includes nonblocking collectives

• MPI allows the user to control passing data between processes through well defined subroutines

• API: C, C++, Fortran

• Libraries: C#, Java, Python, R

• MPI is “agnostic” about network architecture, all it cares is that the location that is being run on can be addressed by whatever transport method you are using
MPI Nomenclature

- Rank: The ID of a process, starts counting from 0
- Handle: The unique ID for the communication that is being done
- Buffer: An array or string, either controlled by the user or MPI, which is being transported
- Core: An individual compute element
- Node: A collection of compute elements that share the same network address, share memory, and are typically on the same main board
- Hostfile: The list of hosts you will be running on
- MPI Fabric: The communications network MPI constructs either by itself or using a daemon
- Blocking: Means the communications subroutine waits for the completion of the routine before moving on.
- Collective: All ranks talk to everyone else to solve some problem.
Available MPI Compilers on Odyssey

• OpenMPI
  – Open Source project
  – Supports MPI-2
  – Even releases are stable, odd releases are development

• MVAPICH2
  – Ohio State University project
  – MPI-2.2 support as well as some support for MPI-3

• Intel MPI
  – Version of MVAPICH2 optimized by Intel
  – Requires script to generate hostfile for SLURM

• All compile for C, C++ and Fortran
MPI Hello World (Fortran/C)

PROGRAM hello

    !### Need to include this to be able to hook into the MPI API ###
    INCLUDE 'mpif.h'

    INTEGER*4 :: numprocs, rank, ierr

    !### Initializes MPI ###
    CALL MPI_INIT(ierr)

    !### Figures out the number of processors I am asking for ###
    CALL MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

    !### Figures out which rank we are ###
    CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

    write(*,*) 'Process', rank, 'out of', numprocs

    !### Need this to shutdown MPI ###
    CALL MPI_FINALIZE(ierr)

END PROGRAM hello

#include <stdio.h>
/* Need to include this to be able to hook into the MPI API */
#include <mpi.h>

int main(int argc, char *argv[]) {
    int numprocs, rank;

    /* Initializes MPI */
    MPI_Init(&argc, &argv);

    /* Figures out the number of processors I am asking for */
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

    /* Figures out which rank we are */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    printf("Process %d out of %d\n", rank, numprocs);

    /* Need this to shutdown MPI */
    MPI_Finalize();
}
.bashrc and Example Scripts

• We need to add the relevant modules to the .bashrc so that all the nodes being used will load the correct libraries.

• Add: module load centos6/openmpi-1.6.5_intel-13.0.079

• All example scripts can be found at /n/holyscratch/computefest/mpi
Compiling and Running OpenMPI

```
[pedmon@itc011 ~]$ module load centos6/openmpi-1.6.5_intel-13.0.079
Loading module hpc/intel-compilers-13.0.079.
Loading module centos6/openmpi-1.6.5_intel-13.0.079.
[pedmon@itc011 ~]$ mpif90 hello.f90
[pedmon@itc011 ~]$ cat runscript
#!/bin/bash
#SBATCH -n 4
#SBATCH -t 10
#SBATCH --mem-per-cpu=1000
#SBATCH -p general
#SBATCH -o test.out
#SBATCH -e test.err

mpirun -np 4 ./a.out
[pedmon@itc011 ~]$ sbatch runscript
Submitted batch job 4752881
[pedmon@itc011 ~]$ cat test.out
Process 1 out of 4
Process 3 out of 4
Process 2 out of 4
Process 0 out of 4
[pedmon@itc011 ~]$
```
Running via Commandline

• OpenMPI
  – hostname slots=8
  – mpirun –np 16 --hostfile=hosts ./a.out

• MVAPICH2: Same as OpenMPI but hostfile is different
  – MVAPICH: hostname:8
MPI Collective Communications

• These commands involve all-to-all, all-to-one or one-to-all communications

• All Collective Commands are blocking communications

• Collective Communications are efficient as they are written by professionals using best practices

• However, Collective Communications also do not scale well and may not work at very large scale

• Future versions of MPI will have non-blocking versions of collectives
## Collective Subroutines

<table>
<thead>
<tr>
<th>Category</th>
<th>Subroutines</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. One buffer</td>
<td><strong>MPI_BCAST</strong></td>
</tr>
<tr>
<td>2. One send buffer and one receive buffer</td>
<td><strong>MPI_GATHER, MPI_SCATTER, MPI_ALLGATHER, MPI_ALLTOALL, MPI_GATHERV, MPI_SCATTERV, MPI_ALLGATHERV, MPI_ALLTOALLV</strong></td>
</tr>
<tr>
<td>3. Reduction</td>
<td><strong>MPI_REDUCE, MPI_ALLREDUCE, MPI_SCAN, MPI_REDUCE_SCATTER</strong></td>
</tr>
<tr>
<td>4. Others</td>
<td><strong>MPI_BARRIER, MPI_OP_CREATE, MPI_OP_FREE</strong></td>
</tr>
</tbody>
</table>
MPI_BCAST

Fortran: CALL MPI_BCAST(sendbuf, count, datatype, root, comm, ierror)
C / C++: MPI_Bcast(&sendbuf, count, datatype, root, comm)

sendbuf: Buffer to be sent.
count: Number of entries in the buffer
datatype: MPI_Datatype being sent (MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION)
root: MPI Rank of host sending data out.
comm: Communicator being used, usually MPI_COMM_WORLD
ierror: Error flag for Fortran
MPI_BCAST example (Fortran/C)

```fortran
program main
  implicit none
  include 'mpif.h'
  INTEGER*4, PARAMETER :: n = 100
  INTEGER*4 :: a(n)
  INTEGER*4 :: i
  INTEGER*4 :: ierr
  INTEGER*4 :: iproc
  INTEGER*4 :: nproc
  call MPI_INIT(ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, iproc, ierr)
  if ( iproc == 0 ) then
    do i = 1, n
      a(i) = i
    end do
  end if
  call MPI_BCAST(a,n,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
  call MPI_FINALIZE(ierr)
end program main
```

```c
#include <iostream>
#include <mpi.h>
using namespace std;

int main(int argc, char** argv){
  int n = 100;
  int a[n];
  int i;
  int iproc;
  int nproc;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&iproc);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  if ( iproc == 0 ){  
    for ( i = 1; i <= n; i++ ){
      a[i-1] = i;
    }
  }
  MPI_Bcast(&a,n,MPI_INTEGER,0,MPI_COMM_WORLD);
  MPI_Finalize();
  return 0;
}
```
MPI_REDUCE

Fortran: CALL MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
C / C++: MPI_Reduce(&sendbuf, &recvbuf, count, datatype, op, root, comm)

sendbuf: The buffer of data to be sent, data to be reduced
recvbuf: The buffer of data to be received, reduced data, only available on the root processor.
op: Operation to be done (MPI_SUM, MPI_PROD, MPI_MAX, MPI_MIN, etc.)
MPI_REDUCE example (Fortran/C)

```fortran
program main
  implicit none
  include 'mpif.h'
  INTEGER*4, PARAMETER :: n = 1000
  INTEGER*4, ALLOCATABLE :: a(:)
  INTEGER*4 :: i, ista, iend, sum, ssum, ierr, iproc, nproc, nproc
  call MPI_INIT(ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, iproc, ierr)
  call para_range(1, n, nproc, iproc, ista, iend)
  ALLOCATE(a(ista:iend))
  do i = ista, iend
    a(i) = i
  end do
  sum = 0
  do i = ista, iend
    sum = sum + a(i)
  end do
  call MPI_REDUCE(sum, ssum, 1, MPI_INTEGER, MPI_SUM, 0, MPI_COMM_WORLD, ierr)
  sum = ssum
  if ( iproc == 0 ) write(*,*)'sum =', sum
  DEALLOCATE(a)
  call MPI_FINALIZE(ierr)
end program main

subroutine para_range(n1, n2, nprocs, irank, ista, iend)
  INTEGER*4 :: n1 ! Lowest value of iteration variable
  INTEGER*4 :: n2 ! Highest value of iteration variable
  INTEGER*4 :: nprocs ! # cores
  INTEGER*4 :: irank ! Iproc (rank)
  INTEGER*4 :: ista ! Start of iterations for rank iproc
  INTEGER*4 :: iend ! End of iterations for rank iproc
  INTEGER*4 :: iwork1, iwork2 ! Work space
  iwork1 = (n2 - n1 + 1) / nprocs
  iwork2 = MOD(n2 - n1 + 1, nprocs)
  ista = irank * iwork1 + n1 + iwork2
  iend = ista + iwork1 - 1
  if ( iwork2 > irank ) iend = iend + 1
  return
end subroutine para_range
```

```c
#include <iostream>
#include <mpi.h>
#include <new>

using namespace std;

int main(int argc, char** argv){
  int i, sum, ssum, iproc, nproc, ista, iend, loc_dim;
  int n = 1000;
  int *a;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&iproc);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  para_range(1,n,nproc,iproc,ista,iend);
  loc_dim = iend - ista + 1;
  a = new int[loc_dim];
  for ( i = 0; i < loc_dim; i++ ){
    a[i] = i + ista;
  }
  sum = 0;
  for ( i = 0; i < loc_dim; i++ ){
    sum = sum + a[i];
  }
  MPI_Reduce(&sum,&ssum,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD);
  sum = ssum;
  if ( iproc == 0 ){
    cout << "sum = " << sum << endl;
  }
  delete [] a;
  MPI_Finalize();
  return 0;
}

void para_range(int n1, int n2, int &nprocs, int &irank, int &ista, int &iend){
  int iwork1;
  int iwork2;
  iwork1 = (n2 - n1 + 1) / nprocs;
  iwork2 = (n2 - n1 + 1) % nprocs;
  ista = irank * iwork1 + n1 + min(irank, iwork2);
  iend = ista + iwork1 - 1;
  if ( iwork2 > irank ) iend = iend + 1;
}
```

#include <iostream>
#include <mpi.h>
#include <new>

using namespace std;

int main(int argc, char** argv){
  int i, sum, ssum, iproc, nproc, ista, iend, loc_dim;
  int n = 1000;
  int *a;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&iproc);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  para_range(1,n,nproc,iproc,ista,iend);
  loc_dim = iend - ista + 1;
  a = new int[loc_dim];
  for ( i = 0; i < loc_dim; i++ ){
    a[i] = i + ista;
  }
  sum = 0;
  for ( i = 0; i < loc_dim; i++ ){
    sum = sum + a[i];
  }
  MPI_Reduce(&sum,&ssum,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD);
  sum = ssum;
  if ( iproc == 0 ){
    cout << "sum = " << sum << endl;
  }
  delete [] a;
  MPI_Finalize();
  return 0;
}

void para_range(int n1, int n2, int &nprocs, int &irank, int &ista, int &iend){
  int iwork1;
  int iwork2;
  iwork1 = (n2 - n1 + 1) / nprocs;
  iwork2 = (n2 - n1 + 1) % nprocs;
  ista = irank * iwork1 + n1 + min(irank, iwork2);
  iend = ista + iwork1 - 1;
  if ( iwork2 > irank ) iend = iend + 1;
}
MPI_BARRIER

Fortran: CALL MPI_BARRIER(comm,ierr)
C/C++: MPI_Barrier(comm)

• Be careful this will block everything until all the ranks reach this point of the code.

• Good for debugging and synchronization.

• Avoid using barriers as much as possible.
Point-to-Point Communications

- Allows messages to be sent back and forth between two ranks
- Most scalable MPI codes use primarily point-to-point communications
- Blocking vs. non-blocking
  - Overlapping communications with computation (non-blocking)
    - Always post receives as soon as possible and do so prior to the send.
    - Sends should be posted as soon as data is ready, other work should be done prior to the send buffer being used again to allow for data to be sent
    - Be sure to ping the MPI fabric with IPROBE to keep things rolling
Blocking Communications

FORTRAN:

CALL MPI_SEND(buffer, count, datatype, destination, tag, communicator, ierr)
CALL MPI_RECV(buffer, count, datatype, source, tag, communicator, status ierr)

C/C++:

MPI_Send(&buffer, count, datatype, destination, tag, communicator)
MPI_Recv(&buffer, count, datatype, source, tag, communicator, &status)

buffer: Data to be sent / received ( e.g., array )
count: Number of data elements
datatype: Type of data, for example MPI_INT, MPI_REAL8, etc
destination: Rank of destination MPI process
source: Rank of source MPI process
tag: Message label
communicator: Set of MPI processes used for communication
status: The status object
ierr: Returned error code ( Fortran only )
Blocking Communications Example

Fortran:
if ( iproc == 0 ) then
    call MPI_SEND(sendbuf, icount, MPI_REAL8, 1, itag, MPI_COMM_WORLD, ierr)
else if ( iproc == 1 ) then
    call MPI_RECV(recvbuf, icount, MPI_REAL8, 0, itag, MPI_COMM_WORLD, istatus, ierr)
end if

C/C++:
if ( iproc == 0 ) {
    MPI_Send(sendbuf, icount, MPI_REAL8, 1, itag, MPI_COMM_WORLD);
}
else if ( iproc == 1 ) {
    MPI_Recv(recvbuf, icount, MPI_REAL8, 0, itag, MPI_COMM_WORLD, istatus);
}
Non-Blocking Communications

FORTRAN:
   CALL MPI_ISEND(buffer, count, datatype, destination, tag, communicator, request, ierr)
   CALL MPI_IRECV(buffer, count, datatype, source, tag, communicator, request, ierr)

C/C++:
   MPI_Isend(&buffer, count, datatype, destination, tag, communicator, &request)
   MPI_Irecv(&buffer, count, datatype, source, tag, communicator, &request)

buffer: Data to be sent / received (e.g., array)
count: Number of data elements
datatype: Type of data, for example MPI_INT, MPI_REAL8, etc
destination: Rank of destination MPI process
source: Rank of source MPI process
tag: Message label
communicator: Set of MPI processes used for communication
request: The handle assigned by MPI for the message.
ierr: Returned error code (Fortran only)
MPI_WAIT

Fortran: CALL MPI_WAIT(request,status,ierr)
C/C++: MPI_Wait(&request, &status)

- This command will make the rank wait until the message associated with the request handle is completed

- Place this immediately prior to when you need the buffer again, not immediately after the ISEND or IRECV, else you defeat the purpose of using non-blocking communications

- MPI_WAITALL will take an array of requests
Non-Blocking Communications Example

Fortran:
if ( iproc == 0 ) then
    call MPI_ISEND(sendbuf, icount, MPI_REAL8, 1, itag, MPI_COMM_WORLD, ireq, ierr)
else if ( iproc == 1 ) then
    call MPI_IRECV(recvbuf, icount, MPI_REAL8, 0, itag, MPI_COMM_WORLD, ireq, ierr)
end if
call MPI_WAIT(ireq, istatus, ierr)

C/C++:
if ( iproc == 0 ) {
    MPI_Isend(sendbuf, icount, MPI_REAL8, 1, itag, MPI_COMM_WORLD, ireq);
}
else if ( iproc == 1 ) {
    MPI_Irecv(recvbuf, icount, MPI_REAL8, 0, itag, MPI_COMM_WORLD, ireq);
}
MPI_Wait(ireq, istatus);
MPI_IPROBE

Fortran: CALL MPI_IPROBE(source,tag,comm,flag,status,ierr)
C/C++: MPI_Iprobe(source,tag,comm,&flag,&status)

source: Source rank or MPI_ANY_SOURCE
tag: Tag of communications or MPI_ANY_TAG
flag: Logical flag regarding communications

• Probes communications to see if things are running
  Essential greases the wheels

• Place periodically in code
Non-Trivial, Trivial Example: 2-D Jacobi Relaxation

- Heat Transfer Boundary Value Problem
- Iterate until solution is to steady state
  \[ f_{n+1}(i,j) = 0.25 \times (f_n(i+1,j) + f_n(i-1,j) + f_n(i,j+1) + f_n(i,j-2)) \]
- Domain Decomposition
Serial Jacobi Relaxation Workflow

• Start Program

• Initialize Boundaries and Grid

• Run Relaxation Scheme

• Check to see if solution is converged

• If not iterate, if so finish and output result (optional, print intermediate states)
Parallel Jacobi Relaxation Workflow

- Start Program

- Initialize external and internal Boundaries for each MPI domain as well as Grid for each MPI brick

- Run Relaxation Scheme on the MPI domain for each brick.

- Send boundary data for next run of relaxation scheme

- Calculate convergence per brick and then pool result.

- If relaxation still needs to be run then receive data from boundary communication and run iteration again. If not then reassemble all the data into a unified grid and print result (optional do this for intermediate states as well).
Questions, Comments, Concerns?

Contact rchelp@fas.harvard.edu for help.