Building and Running a Parallel Application
Task/Channel Model

- Design Efficient Parallel Programs (or Algorithms)
  - Mainly for distributed memory systems (e.g. Clusters)

- Break Parallel Computations into:
  - Tasks (program solving part of a problem, memory & I/O ports)
  - Channels (message queue from one Tasks output I/O port to another’s input I/O port)

- Communication Specifics
  - Tasks receive data from other Tasks via Channels
    - Receives are synchronous (task blocks until desired message is received)
  - Tasks send data to other Tasks via Channels
    - Sends are asynchronous (messages are sent and work continues)
  - Note: This is how MPI_Send & MPI_Recv api calls work
Foster’s Design Methodology

- **Partitioning**
  - Dividing the Problem into Tasks

- **Communication**
  - Determine what needs to be communicated between the Tasks over Channels

- **Agglomeration**
  - Group or Consolidate Tasks to improve efficiency or simplify the programming solution

- **Mapping**
  - Assign tasks to the Computer Processors
    - (assume distributed-memory system e.g. Cluster)
Illustration of the Four Steps

Figure 3.2 Foster’s parallel algorithm design methodology.
Step 1: Divide Computation & Data into Pieces

● Domain Decomposition – Data Centric Approach
  ■ Divide up most frequently used data
  ■ Associate the computations with the divided data

● Functional Decomposition – Computation Centric Approach
  ■ Divide up the computation
  ■ Associate the data with the divided computations

● Primitives: Resulting Pieces from either Decomposition
  ■ The goal is to have as many Primitives as possible
Figure 3.3 Three domain decompositions of a three-dimensional matrix, resulting in markedly different collections of primitive tasks.
Step 2: Determine Communication Patterns between Primitive Tasks

- **Local Communication**
  - When Tasks need data from a small number of other Tasks
  - Channel from Producing Task to Consuming Task Created

- **Global Communication**
  - When Task need data from many or all other Tasks
  - Channels for this type of communication are not created during this step
Communication Goals

- Communication is balanced among all Tasks
- Each Task Communicates with a minimal number of neighbors
- Tasks can Perform Communications concurrently
- Tasks can Perform Computations concurrently

Note: **Serial codes do not require communication. When adding communication to parallel codes, consider this an overhead because Tasks cannot perform Computations while waiting for data. If not done carefully, the cost of communication can outweigh the performance benefit of parallelism.**
Step 3: Group Tasks to Improve Efficiency or Simplify Programming

- Increase Locality
  - remove communication by agglomerating Tasks that Communicate with one another
  - Combine groups of sending & receiving task
    - Send fewer, larger messages rather than more short messages which incur more message latency.

- Maintain Scalability of the Parallel Design
  - Be careful not to agglomerate Tasks so much that moving to a machine with more processors will not be possible

- Reduce Software Engineering costs
  - Leveraging existing sequential code can reduce the expense of engineering a parallel algorithm
Figure 3.5  Agglomerating tasks can eliminate communications or at least reduce their overhead. (a) Combining tasks that are connected by a channel eliminates that communication, increasing the locality of the parallel algorithm. (b) Combining sending and receiving tasks reduces the number of message transmissions.
Step 4: Assigning Tasks to Processors

- Maximize Processor Utilization
  - Ensure computation is evenly balanced across all processors

- Minimize Interprocess Communication

Figure 3.6 The mapping process. (a) A task/channel graph. (b) Mapping of tasks to three processors. Some channels now represent intraprocessor communications, while others represent interprocessor communications.
Which Technique Should You Choose?

MPI
- Code will run on distributed- and/or shared-memory systems
- Functional or nontrivial data parallelism within a single application

OpenMP
- Code will run on shared-memory systems
- Parallel constructs are simple, e.g., independent loop iterations
- Want to parallelize a serial code using OpenMP directives to (say) gcc
- Want to create a hybrid by adding OpenMP directives to an MPI code

Task-Oriented Parallelism (Grid style)
- Parallelism is at the application-level, coarse-grained, scriptable
- Little communication or synchronization is needed
What is MPI?

Message Passing Interface (MPI) is a standardised interface. Using this interface, several implementations have been made. The MPI standard specifies three forms of subroutine interfaces:

1. Language independent notation;
2. Fortran notation;
3. C notation.
MPI Features

MPI implementations provide:

- Abstraction of hardware implementation
- Synchronous communication
- Asynchronous communication
- File operations
- Time measurement operations
<table>
<thead>
<tr>
<th>Implementations</th>
<th>Operating Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPICH</td>
<td>Unix / Windows NT</td>
</tr>
<tr>
<td>MPICH-T3E</td>
<td>Cray T3E</td>
</tr>
<tr>
<td>LAM</td>
<td>Unix/SGI Irix/IBM AIX</td>
</tr>
<tr>
<td>Chimp</td>
<td>SunOS/AIX/Irix/HP-UX</td>
</tr>
<tr>
<td>WinMPI</td>
<td>Windows 3.1 (no network req.)</td>
</tr>
</tbody>
</table>
What is the difference between programming using the traditional approach and the MPI approach:

1. Use of MPI library
2. Compiling
3. Running
When a program is written, compiling it should be done a little bit different from the normal situation. Although details differ for various MPI implementations, there are two frequently used approaches.
Compiling (2)

First approach

$ gcc myprogram.c -o myexecutable -lmpi

Second approach

$ mpicc myprogram.c -o myexecutable
In order to run an MPI-Enabled application we generally use the command ‘mpirun’:

```
$ mpirun -np x myexecutable <parameters>
```

Where $x$ is the number of processes to use

- `<parameters>` are the arguments to the executable, if any.
Running (2)

- The ‘mpirun’ program will take care of the creation of processes on selected processors

- By default, ‘mpirun’ will decide which processors to use, this is usually determined by a global configuration file.
MPI Programming

- Implementations of MPI support
  - Fortran
  - C
  - Both
- Here we only consider programming using the C Libraries.
- The first step in writing a program using MPI is to include the correct header:
  
  ```
  #include "mpi.h"
  ```
MPI Basics: Helloworld.c

- MPI programs must include the MPI header file
- Include file is mpi.h for C, mpif.h for Fortran
- `mpicc` to compile

```c
#include <stdio.h>
#include <mpi.h>

void main(int argc, char **argv )
{
    int myid, numprocs;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    printf("Hello from id %d\n", myid);
    MPI_Finalize();
}
```
**MPI_Init**

- Must be the first MPI function call made by every MPI process
- In C, MPI_Init also returns command-line arguments to all processes
- Note, arguments in MPI calls are generally pointer variables

```c
#include <stdio.h>
#include <mpi.h>

void main(int argc, char **argv )
{
    int i;
    int myid, numprocs;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    for (i=0; i<argc; i++)
        printf("argv[%d]=%s\n",i,argv[i]);
    printf("Hello from id %d\n", myid);
    MPI_Finalize();
}
```
Using multiple processes

- When running an MPI enabled program using multiple processes, each process will run an identical copy of the program.
- So there must be a way to know which process we are.
- This situation is comparable to that of programming using the ‘fork’ statement.
- MPI defines two subroutines that can be used.
After MPI is initialized, every process is part of a “communicator”
MPI_COMM_WORLD is the name of this default communicator
MPI_Comm_rank returns the number (rank) of the current process
For MPI_COMM_WORLD, this is a number from 0 to (numprocs-1)
It is possible to create other, user-defined communicators

```c
#include <stdio.h>
#include <mpi.h>

void main(int argc, char **argv )
{
    int i;
    int myid, numprocs;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    for (i=0; i<argc; i++)
        printf("argv[%d]=%s\n",i,argv[i]);
    printf("Hello from id %d\n", myid);
    MPI_Finalize();
}
```
MPI_Comm_size

Returns the total number of processes in the communicator

```c
#include <stdio.h>
#include <mpi.h>

void main(int argc, char **argv )
{
    int i;
    int myid, numprocs;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    for (i=0; i<argc; i++)
        printf("argv[%d]=%s\n",i,argv[i]);
    printf("Hello from id %d, %d or %d
processes\n",myid,myid+1,numprocs);
    MPI_Finalize();
}
```
#include <stdio.h>
#include <mpi.h>

void main(int argc, char **argv )
{
  int i;
  int myid, numprocs;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  for (i=0; i<argc; i++)
    printf("argv[%d]=%s\n",i,argv[i]);
  printf("Hello from id %d, %d or %d processes\n",myid,myid+1,numprocs);
  MPI_Finalize();
}

**MPI_Finalize**

- Called when all MPI calls are complete
- Pending communication should be finished before finalization
- Frees system resources used by MPI
Example ‘Hello World!’

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

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

    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);

    printf ("Hello world! from processor (%d/%d)\n", rank+1, size);

    MPI_Finalize();

    return 0;
}
Running ‘Hello World!’

$ mpicc -o hello hello.c
$ mpirun -np 3 hello
Hello world! from processor (1/3)
Hello world! from processor (2/3)
Hello world! from processor (3/3)
$ _
int MPI_Send (void *buf,
    int count,
    MPI_Datatype datatype,
    int dest,
    int tag,
    MPI_Comm comm )

- Synchronously sends a message to \texttt{dest}.
- Data is found in \texttt{buf}, that contains \texttt{count} elements of \texttt{datatype}.
- To identify the send, a \texttt{tag} has to be specified.
- The destination \texttt{dest} is the processor rank in communicator \texttt{comm}. 
int MPI_Recv (void *buf, 
int count, 
MPI_Datatype datatype, 
int source, 
int tag, 
MPI_Comm comm, 
MPI_Status *status)

- Synchronously receives a message from source
- Buffer must be able to hold count elements of datatype
- The status field is filled with status information
- MPI_Recv and MPI_Send calls should match; 
  - Equal tag, count, datatype
**MPI_Status: Status Record**

- MPI_Recv blocks until a message is received or an error occurs
- Once MPI_Recv returns the status record can be checked
  - status->MPI_SOURCE (where the message came from)
  - status->MPI_TAG (the tag value, user-specified)
  - status->MPI_ERROR (error condition, if any)

```c
printf("Hello from id %d, %d of %d processes\n",myid,myid+1,numprocs);
for(i=1; i<numprocs; i++)
{
    MPI_Recv(sig,sizeof(sig),MPI_CHAR,i,0,MPI_COMM_WORLD,&status);
    printf("%s",sig);
    printf("Message source = %d\n",status.MPI_SOURCE);
    printf("Message tag = %d\n",status.MPI_TAG);
    printf("Message Error condition = %d\n",status.MPI_ERROR);
}
```
Datatypes

MPI_CHAR  
MPI_SHORT  
MPI_INT  
MPI_LONG  
MPI_UNSIGNED_CHAR  
MPI_UNSIGNED_SHORT  
MPI_UNSIGNED  
MPI_UNSIGNED_LONG  
MPI_FLOAT  
MPI_DOUBLE  
MPI_LONG_DOUBLE

signed char  
signed short int  
signed int  
signed long int  
unsigned char  
unsigned short int  
unsigned int  
unsigned long int  
float  
double  
long double

(http://www-jics.cs.utk.edu/MPI/MPIguide/MPIguide.html)
Example send / receive

- Write an MPI routine where master process receives squares of ID’s of each slave and prints those values.
- For example process ID 3 sends 9 as a message and master prints 9 received from 3.

```bash
$ mpirun -np 4 sendrecv
Receiving data . . .
[1] sent 1
[2] sent 4
[3] sent 9
```
#include <stdio.h>
#include "mpi.h"

int main (int argc, char *argv[]) {
    int size, rank, i, j;

    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);

    printf("Receiving data . . .\n");

    MPI_Finalize();
    return 0;
}
#include <stdio.h>
#include "mpi.h"

int main (int argc, char *argv[])
{
    MPI_Status s;
    int size, rank, i, j;

    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);

    if (rank == 0) // Master process
    {
        printf ("Receiving data . . .\n");
        for (i = 1; i < size; i++)
        {
            MPI_Recv ((void *)&j, 1, MPI_INT, i, 0xACE5, MPI_COMM_WORLD, &s);
            printf ("[%d] sent %d\n", i, j);
        }
    }
    else
    {
        j = rank * rank;
        MPI_Send ((void *)&j, 1, MPI_INT, 0, 0xACE5, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
Running send / receive

$ mpicc -o sendrecv sendrecv.c
$ mpirun -np 4 sendrecv
Receiving data . . .
[1] sent 1
[2] sent 4
[3] sent 9
$ _
Measuring running time

- double MPI_Wtime (void); // wallclock timer
- MPI_Wtick returns the resolution of MPI_Wtime in seconds
- Value will be some small fraction of a second

```c
double timeStart, timeEnd;
...
timeStart = MPI_Wtime();
    // Code to measure time for goes here.
timeEnd = MPI_Wtime();
...
printf ("Running time = %f seconds\n", timeEnd – timeStart);
```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
for (i=0; i<argc; i++) printf("argv[%d]=%s\n",i,argv[i]);
if (myid == 0)
{
  printf("Hello from id %d, %d of %d processes\n",myid,myid+1,numprocs);
  for(i=1; i<numprocs; i++)
  {
    MPI_Recv(sig,sizeof(sig),MPI_CHAR,i,0,MPI_COMM_WORLD,&status);
    printf("%s",sig);
  }
start = MPI_Wtime();
for (i=0; i<100; i++)
{
  a[i] = i;
  b[i] = i * 10;
  c[i] = i + 7;
  a[i] = b[i] * c[i];
}
end = MPI_Wtime();
printf("Our timers precision is %.20f seconds\n",MPI_Wtick());
printf("This silly loop took %.5f seconds\n",end-start);
int MPI_Bcast ( void *buffer,  
    int count,  
    MPI_Datatype datatype,  
    int root,  
    MPI_Comm comm)

● Synchronously broadcasts a message from root, to all processors in communicator \textit{comm}
● Buffer is used as source in root processor, as destination in others.
int MPI_Barrier (MPI_Comm comm)

- Blocks until all processes defined in \textit{comm} have reached this routine
- A mechanism to force synchronization amongst all processes
- Useful when you are timing performance
  - Assume all processes are performing the same calculation
  - You need to ensure they all start at the same time

\begin{verbatim}
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
result = run_big_computation();
MPI_Barrier(MPI_COMM_WORLD);
end = MPI_Wtime();
printf("This big computation took %.5f seconds\n",end-start);
\end{verbatim}
Example broadcast / barrier

- Useful when you want to ensure that all processes have completed an operation before any of them begin a new one.

```c
int main (int argc, char *argv[])
{
    int rank, i;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);

    if (rank == 0) i = 27;
    MPI_Bcast ((void *)&i, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf ("[%d] i = %d\n", rank, i);

    // Wait for every process to reach this code
    MPI_Barrier (MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
```
Running broadcast / barrier

$ mpicc -o broadcast broadcast.c
$ mpirun -np 3 broadcast
Running broadcast / barrier

```bash
$ mpicc -o broadcast broadcast.c
$ mpirun -np 3 broadcast
[0] i = 27
[1] i = 27
[2] i = 27
$ _
```
void my_bcast(void* data, int count, MPI_Datatype datatype, int root, MPI_Comm communicator) {
    int world_rank;
    MPI_Comm_rank(communicator, &world_rank);
    int world_size;
    MPI_Comm_size(communicator, &world_size);
}
void my_bcast(void* data, int count, MPI_Datatype datatype, int root,  
    MPI_Comm communicator) { 

    int world_rank;  
    MPI_Comm_rank(communicator, &world_rank);  
    int world_size;  
    MPI_Comm_size(communicator, &world_size);  

    if (world_rank == root) {  
        // If we are the root process, send our data to everyone  
        int i;  
        for (i = 0; i < world_size; i++) {  
            if (i != world_rank) {  
                MPI_Send(data, count, datatype, i, 0, communicator);  
            }  
        }  
    } else {  
        // If we are a receiver process, receive the data from the root  
        MPI_Recv(data, count, datatype, root, 0, communicator,  
            MPI_STATUS_IGNORE);  
    }  
}