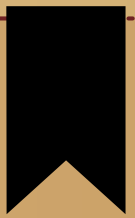


Assignment 3

Message Passing Interface



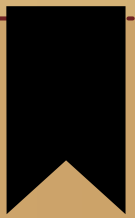
Outline



- Background
- Message Passing
- MPI
 - Group and Context
 - Communication Modes
 - Blocking/Non-blocking
 - Features
 - Programming / issues
- Tutorial



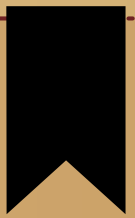
Distributed Computing Paradigms



- Communication Models:
 - Message Passing
 - Shared Memory
- Computation Models:
 - Functional Parallel
 - Data Parallel



Message Passing



- A process is a program counter and address space.
- Message passing is used for communication among processes.
- Inter-process communication:
 - Type:
Synchronous / Asynchronous
 - Movement of data from one process's address space to another's



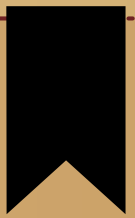
Synchronous Vs. Asynchronous

- A synchronous communication is not complete until the **message** has been received.
- An asynchronous communication completes as soon as the **message** is on the way.

What is message passing?

- Data transfer.
- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

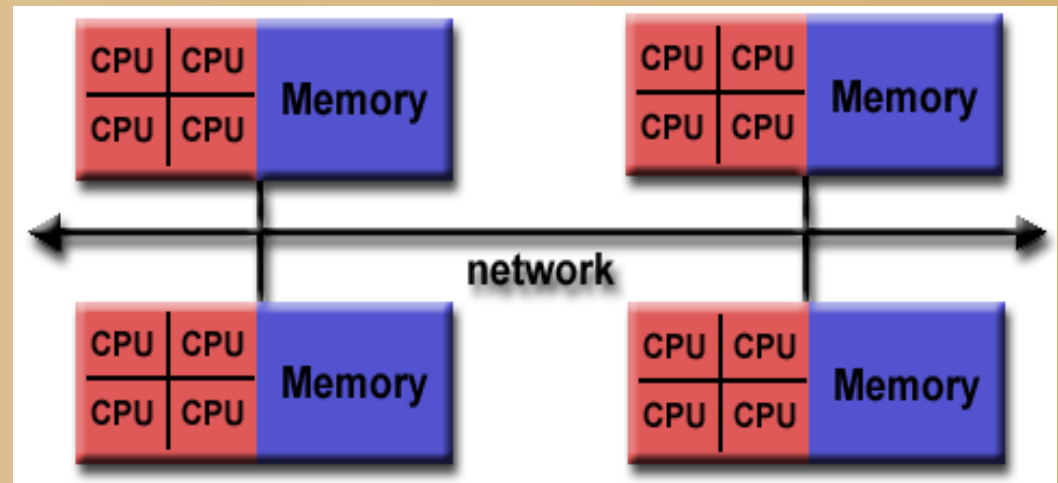
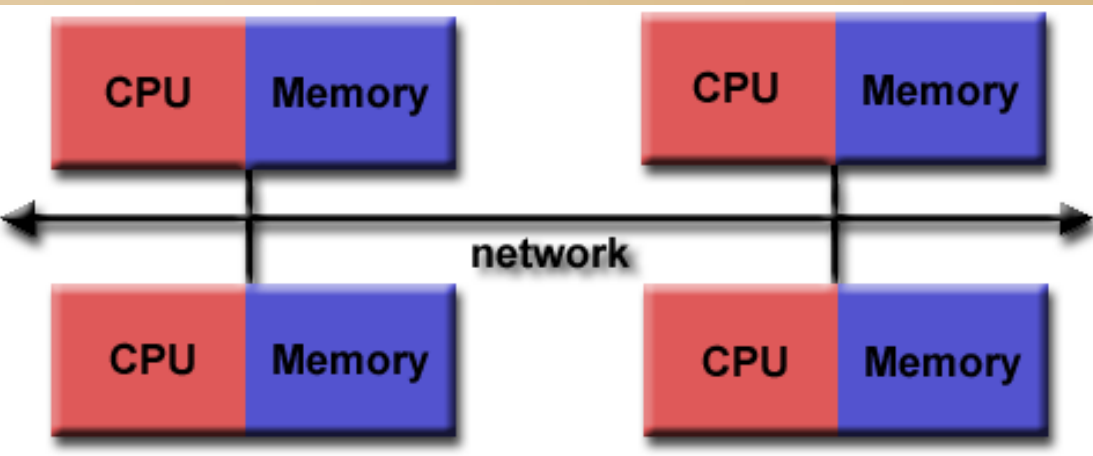
What is MPI?

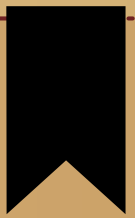


- A message-passing library specifications:
 - Extended message-passing model
 - Not a language or compiler specification
 - Not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks.
- Communication modes: *standard*, *synchronous*, *buffered*, and *ready*.
- Designed to permit the development of parallel software libraries.
- Designed to provide access to advanced parallel hardware for
 - End users
 - Library writers
 - Tool developers



Programming Model

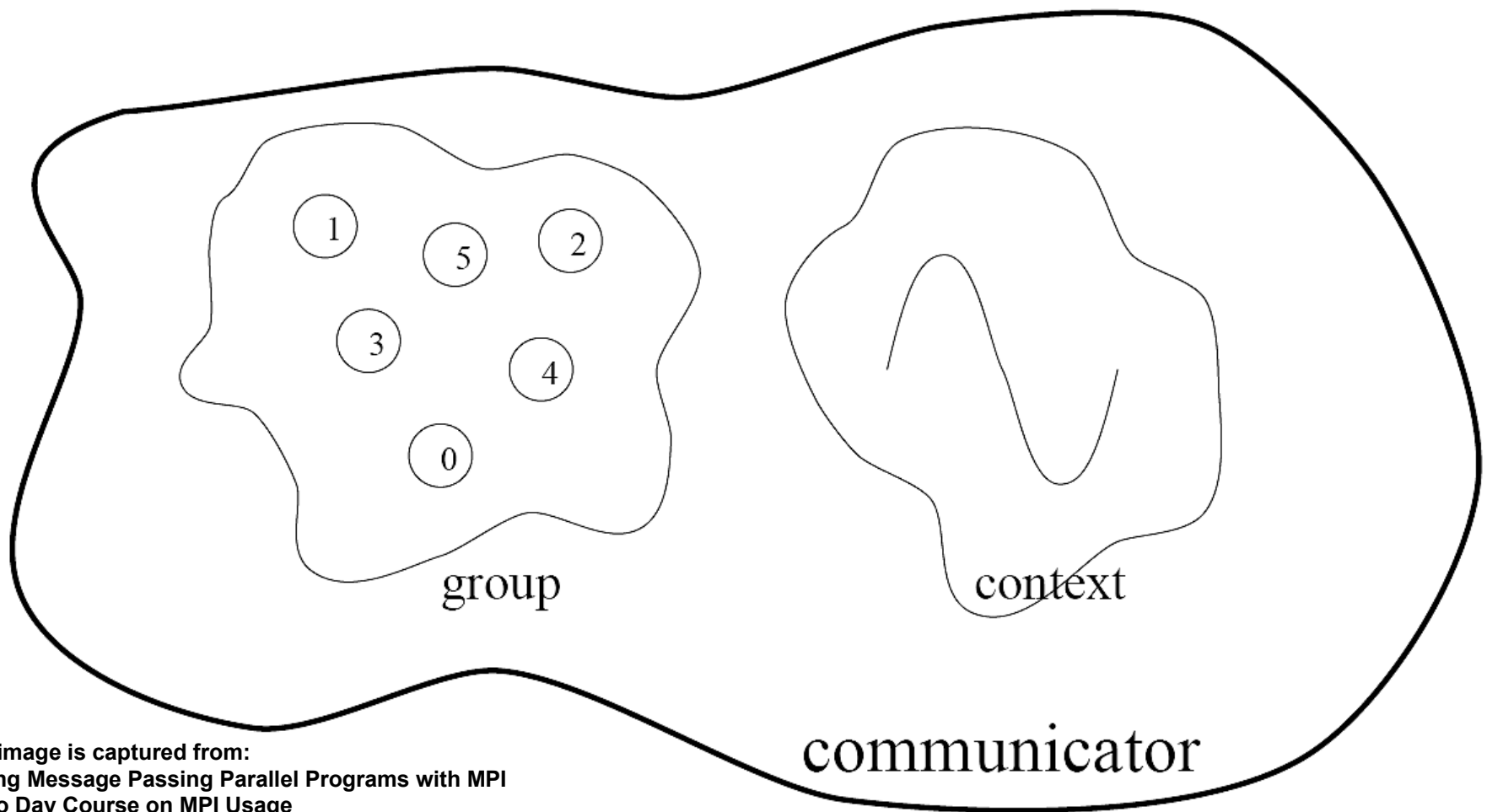




- Today, MPI runs on virtually any hardware platform:
 - Distributed Memory
 - Shared Memory
 - Hybrid
- The programming model clearly remains a distributed memory model however, regardless of the underlying physical architecture of the machine.
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.



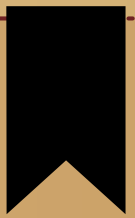
Group and Context



Group and Context (cont.)

- Are two important and indivisible concepts of MPI.
- Group: is the set of processes that communicate with one another.
- Context: it is somehow similar to the frequency in radio communications.
- Communicator: is the central object for communication in MPI. Each communicator is associated with a group and a context.

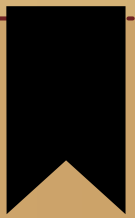
Communication Modes



- Based on the type of send:
 - Synchronous: Completes once the acknowledgement is received by the sender.
 - Buffered send: completes immediately, unless if an error occurs.
 - Standard send: completes once the message has been sent, which may or may not imply that the message has arrived at its destination.
 - Ready send: completes immediately, if the receiver is ready for the message it will get it, otherwise the message is dropped silently.



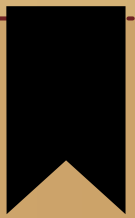
Blocking vs. Non-Blocking



- Blocking, means the program will not continue until the communication is completed.
- Non-Blocking, means the program will continue, without waiting for the communication to be completed.



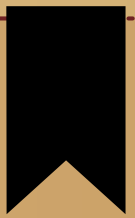
Features of MPI



- General
 - Communications combine context and group for message security.
 - Thread safety can't be assumed for MPI programs.



Features that are NOT part of MPI



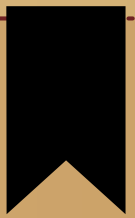
- Process Management
- Remote memory transfer
- Threads
- Virtual shared memory



Why to use MPI?

- MPI provides a powerful, efficient, and portable way to express parallel programs.
- MPI was explicitly designed to enable libraries which may eliminate the need for many users to learn (much of) MPI.
- Portable
- Good way to learn about subtle issues in parallel computing

How big is the MPI library?



- Huge (125 Functions).
- Basic (6 Functions).





Basic Commands

Standard with blocking



Skeleton MPI Program

```
#include <mpi.h>

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

    /* main part of the program */

    /*
       Use MPI function call depend on your data
       partitioning and the parallelization
       architecture
    */

    MPI_Finalize();
}
```

Initializing MPI

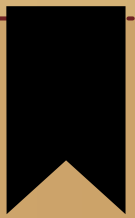
- The initialization routine MPI_INIT is the first MPI routine called.
- MPI_INIT is called once

```
int mpi_Init( int *argc, char **argv );
```

A minimal MPI program(c)

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    printf("Hello, world!\n");
    MPI_Finalize();
    Return 0;
}
```

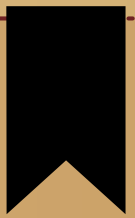
A minimal MPI program(c) (cont.)



- `#include "mpi.h"` provides basic MPI definitions and types.
- `MPI_Init` starts MPI
- `MPI_Finalize` exits MPI
- Note that all non-MPI routines are local; thus `printf` run on each process
- Note: MPI functions return error codes or `MPI_SUCCESS`



Error handling



- By default, an error causes all processes to abort.
- The user can have his/her own error handling routines.
- Some custom error handlers are available for downloading from the net.



Hello with rank and size


```
>include <mpi.h#  
#include <stdio.h>  
int main(int argc, char *argv[])  
{  
    int rank, size;  
    MPI_Init(&argc, &argv);  
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
    MPI_Comm_size(MPI_COMM_WORLD, &size);  
    printf("I am %d of %d\n", rank, size);  
    MPI_Finalize();  
    return 0;  
}
```


Some concepts

- The default communicator is the
`MPI_COMM_WORLD`
- A process is identified by its rank in the group associated with a communicator.

Data Types



- The data message which is sent or received is described by a triple (address, count, datatype).
 - The following data types are supported by MPI:
 - Predefined data types that are corresponding to data types from the programming language.
 - Arrays.
 - Sub blocks of a matrix
 - User defined data structure.
 - A set of predefined data types
- 

Basic MPI types

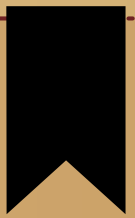
MPI datatype

MPI_CHAR
MPI_SIGNED_CHAR
MPI_UNSIGNED_CHAR
MPI_SHORT
MPI_UNSIGNED_SHORT
MPI_INT
MPI_UNSIGNED
MPI_LONG
MPI_UNSIGNED_LONG
MPI_FLOAT
MPI_DOUBLE
MPI_LONG_DOUBLE

C datatype

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

Why defining the data types during the send of a message?



Because communications take place between heterogeneous machines. Which may have different data representation and length in the memory.



MPI blocking send

```
MPI_SEND(void *start, int  
count, MPI_DATATYPE datatype, int dest,  
int tag, MPI_COMM comm)
```

- The message buffer is described by (**start**, **count**, **datatype**).
- **dest** is the rank of the target process in the defined communicator.
- **tag** is the message identification number.

MPI blocking receive

```
MPI_RECV (void *start, int count,  
MPI_DATATYPE datatype, int source, int tag,  
MPI_COMM comm, MPI_STATUS *status)
```

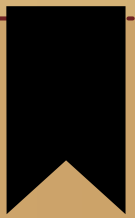
- **Source** is the rank of the sender in the communicator.
- The receiver can specify a wildcard value for source (MPI_ANY_SOURCE) and/or a wildcard value for tag (MPI_ANY_TAG), indicating that any source and/or tag are acceptable
- **Status** is used for extra information about the received message if a wildcard receive mode is used.
- If the count of the message received is less than or equal to that described by the MPI receive command, then the message is successfully received. Else it is considered as a buffer overflow error.

MPI_STATUS

- Status is a data structure
- In C:

```
int recvd_tag, recvd_from, recvd_count;  
MPI_Status status;  
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG,  
        ..., &status)  
recvd_tag = status.MPI_TAG;  
recvd_from = status.MPI_SOURCE;  
MPI_Get_count(&status, datatype,  
             &recvd_count);
```

More info



- A receive operation may accept messages from an arbitrary sender, but a send operation must specify a unique receiver.
- Source equals destination is allowed, that is, a process can send a message to itself.



Why MPI is simple?

- Many parallel programs can be written using just these six functions, only two of which are non-trivial;
 - MPI_INIT
 - MPI_FINALIZE
 - MPI_COMM_SIZE
 - MPI_COMM_RANK
 - MPI_SEND
 - MPI_RECV

Simple full example

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

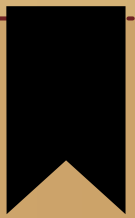
int main(int argc, char *argv[])
{
    const int tag = 42;          /* Message tag */
    int id, ntasks, source_id, dest_id, err, i;
    MPI_Status status;
    int msg[2]; /* Message array */

    err = MPI_Init(&argc, &argv); /* Initialize MPI */
    if (err != MPI_SUCCESS) {
        printf("MPI initialization failed!\n");
        exit(1);
    }
    err = MPI_Comm_size(MPI_COMM_WORLD, &ntasks); /* Get nr of tasks */
    err = MPI_Comm_rank(MPI_COMM_WORLD, &id); /* Get id of this process */
    if (ntasks < 2) {
        printf("You have to use at least 2 processors to run this program\n");
        MPI_Finalize(); /* Quit if there is only one processor */
        exit(0);
    }
}
```

Simple full example (Cont.)

```
if (id == 0) { /* Process 0 (the receiver) does this */
    for (i=1; i<ntasks; i++) {
        err = MPI_Recv(msg, 2, MPI_INT, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, \
                        &status); /* Receive a message */
        source_id = status.MPI_SOURCE; /* Get id of sender */
        printf("Received message %d %d from process %d\n", msg[0], msg[1], \
               source_id);
    }
}
else { /* Processes 1 to N-1 (the senders) do this */
    msg[0] = id; /* Put own identifier in the message */
    msg[1] = ntasks; /* and total number of processes */
    dest_id = 0; /* Destination address */
    err = MPI_Send(msg, 2, MPI_INT, dest_id, tag, MPI_COMM_WORLD);
}

err = MPI_Finalize(); /* Terminate MPI */
if (id==0) printf("Ready\n");
exit(0);
return 0;
}
```



Standard with Non-blocking



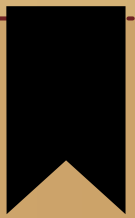
Non-Blocking Send and Receive

```
MPI_ISEND(buf, count, datatype, dest, tag,  
          comm, request)
```

```
MPI_IRecv(buf, count, datatype, dest, tag,  
          comm, request)
```

- request is a request handle which can be used to query the status of the communication or wait for its completion.

Non-Blocking Send and Receive (Cont.)



- A non-blocking send call indicates that the system may start copying data out of the send buffer. The sender must not access any part of the send buffer after a non-blocking send operation is posted, until the complete-send returns.
- A non-blocking receive indicates that the system may start writing data into the receive buffer. The receiver must not access any part of the receive buffer after a non-blocking receive operation is posted, until the complete-receive returns.



Non-Blocking Send and Receive (Cont.)

`MPI_WAIT (request, status)`

`MPI_TEST (request, flag, status)`

- The `MPI_WAIT` will block your program until the non-blocking send/receive with the desired request is done.
- The `MPI_TEST` is simply queried to see if the communication has completed and the result of the query (`TRUE` or `FALSE`) is returned immediately in flag.

Deadlocks in **blocking** operations

- What happens with

Process 0

Send(1)

Recv(1)

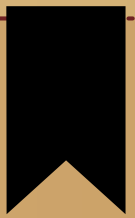
Process 1

Send(0)

Recv(0)

- Send a large message from process 0 to process 1
 - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space(through a receive)
- This is called “unsafe” because it depends on the availability of system buffers.

Some solutions to the “unsafe” problem



- Order the operations more carefully

Process 0

Send(1)

Recv(1)

Process 1

Recv(0)

Send(0)

Use non-blocking operations:

Process 0

ISend(1)

IRecv(1)

Waitall

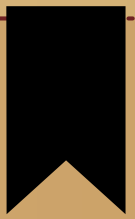
Process 1

ISend(0)

IRecv(0)

Waitall





Collective Operations



Introduction to collective operations in MPI

- Collective operations are called by all processes in a communicator
- MPI_Bcast distributes data from one process(the root) to all others in a communicator.

Syntax:

```
MPI_Bcast(void *message, int count, MPI_Datatype  
datatype, int root, MPI_Comm comm)
```

- MPI_Reduce combines data from all processes in communicator or and returns it to one process

Syntax:

```
MPI_Reduce(void *message, void *recvbuf, int count,  
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm  
comm)
```

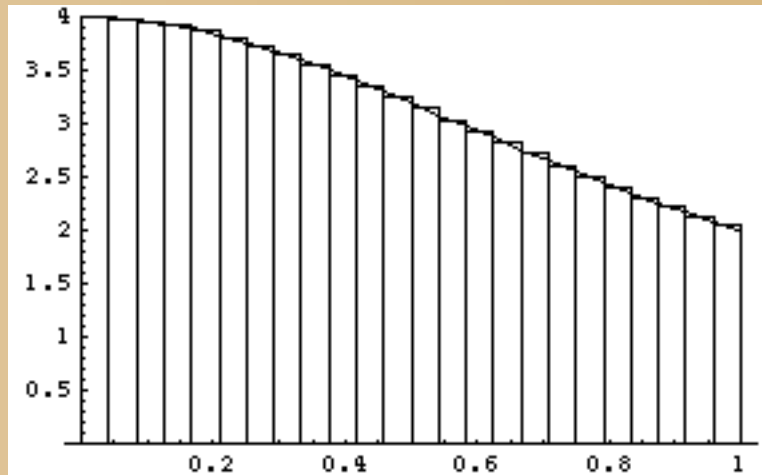
- In many numerical algorithm, send/receive can be replaced by Bcast/Reduce, improving both simplicity and efficiency.

Collective Operations

MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD,
MPI_BAND, MPI_BOR, MPI_LOR, MPI_LAND,
MPI_BXOR, MPI_LXOR, MPI_MAXLOC,
MPI_MINLOC

Example: Compute PI (0)

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

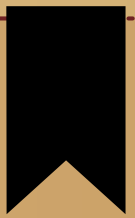


Example: Compute PI (1)

```
#include "mpi.h"
#include <math.h>

int main(int argc, char *argv[])
{
    int done = 0, n, myid, numprocs, I, rc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;
    MPI_INIT(&argc, &argv);
    MPI_COMM_SIZE(MPI_COMM_WORLD, &numprocs);
    MPI_COMM_RANK(MPI_COMM_WORLD, &myid);
    while (!done)
    {
        if (myid == 0)
        {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d", &n);
        }
        MPI_BCAST(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0)
        {
            done = 1;
        }
    }
}
```

Example: Compute PI (2)



```
h = 1.0 / (double)n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs)
{
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x * x);
}
mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

if (myid == 0) printf("pi is approximately %.16f, Error
is %.16f\n", pi, fabs(pi - PI25DT));

MPI_Finalize();
return 0;
}
```

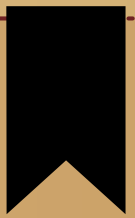
When to use MPI

- Portability and Performance
- Irregular data structure
- Building tools for others
- Need to manage memory on a per processor basis

Compile and run the code

- Compile using:
`mpicc -o pi pi.c`
Or
`mpic++ -o pi pi.cpp`
- `mpirun -np # of procs -machinefile XXX pi`
- `-machinefile` tells MPI to run the program on the machines of XXX.

Where to get MPI library?

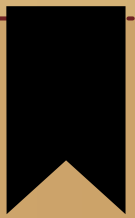


- MPICH (WINDOWS / UNICES)
 - <http://www-unix.mcs.anl.gov/mpi/mpich/>
- Open MPI (UNICES)
 - <http://www.open-mpi.org/>



MPI Sources

- The Standard itself:
 - at <http://www.mpi-forum.org>
 - All MPI official releases, in both postscript and HTML
- Books:
 - *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Skjellum, MIT Press, 1994.
 - *MPI: The Complete Reference*, by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1996.
 - *Designing and Building Parallel Programs*, by Ian Foster, Addison-Wesley, 1995.
 - *Parallel Programming with MPI*, by Peter Pacheco, Morgan-Kaufmann, 1997.
 - *MPI: The Complete Reference Vol 1 and 2*, MIT Press, 1998(Fall).
- Other information on Web:



Thank You

