MPI
Message Passing Interface
Department of Computer Science
University of Kentucky

Materials are based on those from the University of Arizona
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.
Synchronous Vs. Asynchronous (cont.)
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
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

```c
#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();
}
```
The initialization routine MPI_INIT is the first MPI routine called.

MPI_INIT is called once

```c
int mpi_Init( int *argc, char **argv );
```
A minimal MPI program (c)

```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.
Improved Hello (c)

```c
#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

<table>
<thead>
<tr>
<th><strong>MPI datatype</strong></th>
<th><strong>C datatype</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SIGNED_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
</tbody>
</table>
Why defining the data types during the send of a message?

Because communications take place between heterogeneous machines. They 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.
Status is a data structure

In C:

```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);
```
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
```c
#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);
    }

    // Rest of the code...
}
```
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
  
<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Send(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Recv(0)</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Recv(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Send(0)</td>
</tr>
</tbody>
</table>

Use non-blocking operations:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISend(1)</td>
<td>ISend(0)</td>
</tr>
<tr>
<td>IRecv(1)</td>
<td>IRecv(0)</td>
</tr>
<tr>
<td>Waitall</td>
<td>Waitall</td>
</tr>
</tbody>
</table>
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:
  
  ```c
  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:
  
  ```c
  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_LAND, MPI_BAND, MPI_LOR, MPI_BOR,
MPI_LXOR, MPI_BXOR, MPI_MAXLOC,
MPI_MINLOC
Barrier Operation

MPI_Barrier (MPI_Comm comm)

- Forced synchronization, no data is transferred
- Make sure that all processes have finished certain computation. E.g., Barrier may be invoked before a distributed inner product is computed
- Frequent use of BARRIER may degrade performance of a message-passing program
MPI_Gather

- MPI_Gather collects data without the reduction operation
  Syntax:
  ```c
  MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int target, MPI_Comm comm)
  ```

- MPI_Allgather is similar to MPI-Gather, but result will be in all processes
  Syntax:
  ```c
  MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, MPI_Comm comm)
  ```
MPI_Scatter

- MPI_Scatter is a personalized one-to-all communication
  Syntax:
  ```c
  MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, MPI_Comm comm)
  ```

- MPI_Alltoall is the all-to-all personalized communication
  Syntax:
  ```c
  MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, MPI_Comm comm)
  ```
Topologies and Embedding

- Write portable code without specifying the underline interconnection network
- MPI library provides routines to find the most appropriate mapping for the given topology to minimize communication cost
  
  Syntax:
  
  ```c
  MPI_Cart_creat(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm, *comm_cart)
  ```

  - `comm_old`: original communicator, e.g., `MPI_COMM_WORLD`
  - `ndims`: dimensions of the topology, e.g., 2, 3
  - `dims`: array of size `ndims`, specify the shape of topology
  - `periods`: specify if the boundary is periodic, i.e., wraparound
  - `reorder`: whether or not reorder the processes
  - `comm_cart`: new communicator
Define Sub-communicator

- Create sub-communicator from a cartesian communicator
  
  Syntax:
  
  ```c
  MPI_Cart_sub(MPI_Comm comm_cart, int keep_dims, MPI_Comm, *comm_subcart)
  ```

  - `comm_cart`: the Cartesian communicator created
  - `keep_dims`: array to specify the partition
    - `keep_dims[0]=0`, `keep_dims[1]=1` creates a sub-group consists of processes in the same column of the calling process
  - `comm_subcart`: name of the new sub_communicator

  ```c
  MPI_Cart_sub(Comm2D, keep_dims, Column3)
  ```

- If this function is called by process 3, it will create a communicator named Column3 consists of all processes that have the same column number as that of process 3
Shifting Data

- Shifting data along a dimension of the Cartesian topology
  
  Syntax:
  
  ```c
  MPI_Cart_shift(MPI_Comm comm_cart, int dir, int s_step, int *rank_source, int *rank_dest)
  ```

- `comm_cart`: the Cartesian communicator created
- `dir`: dimension of shifting, e.g., 0, 1
- `s_step`: Size of shift step, e.g., 1, -1
- `rank_source`, `rank_dest`, source and destination processes

  ```c
  MPI_Cart_shift(Comm2D, 0, 1, inID, outID)
  ```

- If this function is called by process 3, process 3 computes the source process ID and the destination ID. The actual shift may be performed by `MPI_Send` and `MPI_Recv`
2D Cartesian Decomposition
Translation between Coordinates and Ranks

- Processes in a Cartesian topology are identified by the coordinates, usually in the form of tuples, e.g., (1,2), (0,3)
- All MPI functions for sending and receiving requires process rank
- MPI provides two functions for coordinate and rank translation

Syntax:

```
MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int *rank)
```

```
MPI_Cart_coords(MPI_Comm comm_cart, int *rank, int maxdims, int *coords)
```

- It takes the rank of the process and returns the Cartesian coordinate array
- maxdims: length of the coordinate array, e.g., 2 or 3, dimensions
Example: Cartesian Communicator (1)

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

int main(int argc, char *argv[])
{
    int myid, numprocs;
    int ProcId2D, ProcCords[2], Dims[2], Periods[2], p, q;
    MPI_Status Status;
    MPI_Comm Comm2D;

    MPI_INIT(&argc, &argv);
    MPI_COMM_SIZE(MPI_COMM_WORLD, &numprocs);
    MPI_COMM_RANK(MPI_COMM_WORLD, &myid);
    /* 2D decomposition of 16 processes */
    p = 4;
    q = 4;
```
Example: Cartesian Communicator (2)

```c
Dims[0] = p;
Dims[1] = q;
Periods[0] = 1;
Periods[1] = 1;

MPI_Cart_creat(MPI_COMM_WORLD, 2, Dims, Periods, 1, &Comm2D);

MPI_Comm_rank(Comm2D, &ProcID2D);

MPI_Cart_coords(Comm2D, ProcId2D, 2, ProcCords);

printf("Coord(1)=%d Coord(2)=%d \tProcess=%d \tOldid=%d\n", ProcCords[0], ProcCords[1], ProdId2D, myid);

MPI_Finalize();

return 0;
```

Example: Cartesian Communicator Results

\begin{align*}
\text{Coord}(1) &= 0 \quad \text{Coord}(2) &= 0 \quad \text{Process} = 0 \quad \text{Oldid} = 0 \\
\text{Coord}(1) &= 0 \quad \text{Coord}(2) &= 1 \quad \text{Process} = 1 \quad \text{Oldid} = 1 \\
\text{Coord}(1) &= 0 \quad \text{Coord}(2) &= 2 \quad \text{Process} = 2 \quad \text{Oldid} = 2 \\
\text{Coord}(1) &= 0 \quad \text{Coord}(2) &= 3 \quad \text{Process} = 3 \quad \text{Oldid} = 3 \\
\text{Coord}(1) &= 1 \quad \text{Coord}(2) &= 0 \quad \text{Process} = 4 \quad \text{Oldid} = 4 \\
\text{Coord}(1) &= 1 \quad \text{Coord}(2) &= 1 \quad \text{Process} = 5 \quad \text{Oldid} = 5 \\
\text{Coord}(1) &= 1 \quad \text{Coord}(2) &= 2 \quad \text{Process} = 6 \quad \text{Oldid} = 6 \\
\text{Coord}(1) &= 1 \quad \text{Coord}(2) &= 3 \quad \text{Process} = 7 \quad \text{Oldid} = 7 \\
\text{Coord}(1) &= 2 \quad \text{Coord}(2) &= 0 \quad \text{Process} = 8 \quad \text{Oldid} = 8 \\
\text{Coord}(1) &= 2 \quad \text{Coord}(2) &= 1 \quad \text{Process} = 9 \quad \text{Oldid} = 9 \\
\text{Coord}(1) &= 2 \quad \text{Coord}(2) &= 2 \quad \text{Process} = 10 \quad \text{Oldid} = 10 \\
\text{Coord}(1) &= 2 \quad \text{Coord}(2) &= 3 \quad \text{Process} = 11 \quad \text{Oldid} = 11 \\
\text{Coord}(1) &= 3 \quad \text{Coord}(2) &= 0 \quad \text{Process} = 12 \quad \text{Oldid} = 12 \\
\text{Coord}(1) &= 3 \quad \text{Coord}(2) &= 1 \quad \text{Process} = 13 \quad \text{Oldid} = 13 \\
\text{Coord}(1) &= 3 \quad \text{Coord}(2) &= 2 \quad \text{Process} = 14 \quad \text{Oldid} = 14 \\
\text{Coord}(1) &= 3 \quad \text{Coord}(2) &= 3 \quad \text{Process} = 15 \quad \text{Oldid} = 15
\end{align*}
Example: Compute $\pi$ (0)

$$\pi = \int_{0}^{1} \frac{4}{1 + x^2} \, dx$$
Example: Compute PI (1)

```c
#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)
        }  
    }  
```
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
Programming with MPI
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`.

MPI on ECE Solaris Machines (1)

- Log in to *draco.ece.arizona.edu*
- From outside the UofA first log in to *shell.ece.arizona.edu*
- Create a Text file and name it. For example ML, and have the following lines:

```
150.135.221.71
150.135.221.72
150.135.221.73
150.135.221.74
150.135.221.75
150.135.221.76
150.135.221.77
150.135.221.78
```
#include "mpi.h"
#include <math.h>
#include <stdio.h>

int main(argc, argv) 
    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) break;
        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();
}
MPI on ECE Solaris Machines (3)

- How to compile:
  
  $mpicc$ ex2.c -o ex2 -lm

- How to run:
  
  $mpirun$ -np 4 -machinefile ml ex2
Where to get MPI library?

- MPICH (WINDOWS / UNICES)

- Open MPI (UNICES)
  - http://www.open-mpi.org/
Step By Step Installation of MPICH on windows XP(1)

Process Manager setup

The smpd process manager will be installed on this system. It requires administrator privileges to install so if you are not in the administrator's group you should cancel the installation now. Smpd will be installed as a service used to launch MPI processes. Authorized access to the smpd service is regulated by a secret word entered here. The same passphrase must be used on all systems.

Passphrase:

behappy
Step By Step Installation of MPICH on windows XP(2)
Step By Step Installation of MPICH on windows XP(3)
Step By Step Installation of MPICH on windows XP(4)
Step By Step Installation of MPICH on windows XP(5)
Step By Step Installation of MPICH on windows XP(6)

// mpi-test.cpp : Defines the entry point for the console application.

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

int _tmain(int argc, _TCHAR* 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
", rank, size);
    MPI_Finalize();
    return 0;
}
Step By Step Installation of MPICH on windows XP(7)
Step By Step Installation of MPICH on windows XP(8)
Step By Step Installation of MPICH on windows XP(9)
Step By Step Installation of MPICH on windows XP(10)
Step By Step Installation of MPICH on windows XP(11)
Step By Step Installation of MPICH on windows XP(12)

- Copy executable file to the *bin* directory

- Execute using:
  
  `mpiexec.exe -localonly <# of procs> exe_file_name.exe`