ECE 677: *Distributed Computing Systems*

Salim Hariri

High Performance Distributed Computing Laboratory

University of Arizona

Tele: (520) 621-4378

Www.ece.arizona.edu/~hpdc

Fall 2010

Distributed Systems Design Framework (Cont)

Distributed Computing Paradigms (DCP)			
Computation Models		Communication Models	
Functional Parallel	Data Parallel	Message Passing	Shared Memory
System Architecture and Services (SAS)			
Architecture Models		System Level Services	
Computer Networks and Protocols (CNP)			
Computer Networks		Communication Protocols	

Brief overview

- What: standard for a message passing library (C, C++ and Fortran) to be used for message-passing parallel computing.
- When: 92-94 MPI1; 95-97 MPI2
- Size: MPI1: 127 calls; MPI2: ~150 calls.
  - Many parallel programs can be written with 6 basic functions.
  - Functions are orthogonal.
    - Support for many different communication paradigms.
    - Support for different communication modes.
    - Options offered via different function names, rather than parameters.
- Where:
  - Parallel computers and clusters (distributed or shared memory)
  - NOWs (Network of workstations, heterogeneous systems)
- Find more: http://www.mcs.anl.gov/Projects/MPI

Companion Material

- Online examples available at http://www.mcs.anl.gov/mpi/tutorial
- ftp://ftp.mcs.anl.gov/mpi/mpiexmple.tar.gz contains source code and run scripts that allows you to evaluate your own MPI implementaton

The Message-Passing Model

- A process is (traditionally) a program counter and address space
- Processes may have multiple threads(program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.
- Interprocess communication consists of
  - Synchronization/Asynchronization
  - Movement of data from one process's address space to another's

What is message passing?

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

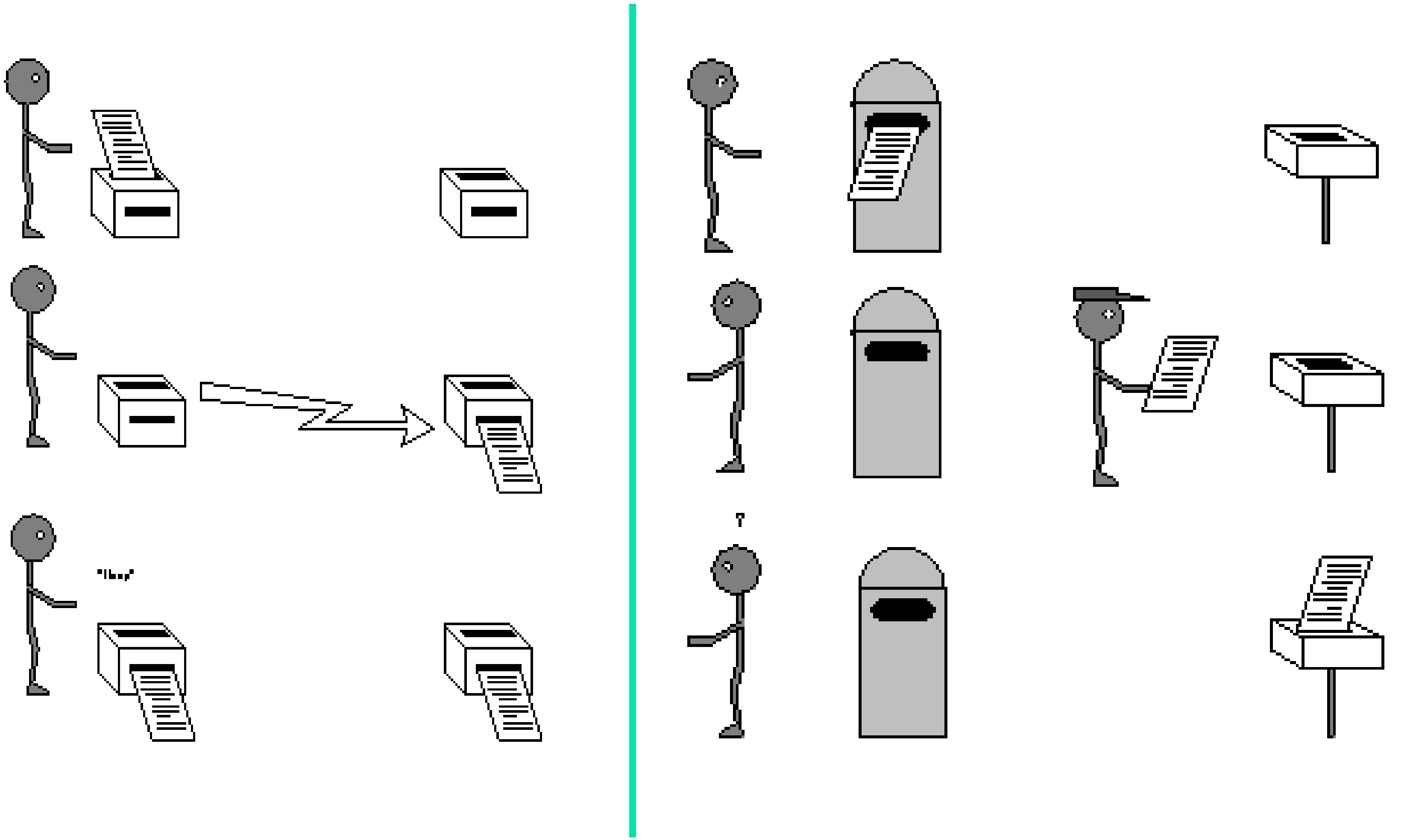
Communication Modes

- 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.

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.)



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.

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

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 !!!!!!!!!!!!!!!!!!!!!!!!!!

Is MPI large or small?

- MPI is large(125 functions)
  - MPI’s extensive functionality requires many functions.
  - Number of functions not necessarily a measure of complexity.
- MPI is small(6 functions)
  - Many parallel programs can be written with just 6 basic functions.
- MPI is just right
  - One can access flexibility when it is required.
  - One need not master all parts of MPI to use it.
  - MPI is whatever size you like

Features that are NOT part of MPI

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

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

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
partition and parallization architecture
    MPI_Finalize();
}
```

Initializing MPI

- The first MPI routine called in any MPI program must be the initialization routine MPI_INIT
- MPI_INIT is called once by every process, before any other MPI routines

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

Startup and endup

- int MPI_Init(int *argc, char ***argv)
  - The first MPI call in any MPI process
  - Establishes MPI environment
  - One and only one call to MPI_INIT per process
- int MPI_Finalize(void)
  - Exiting from MPI
  - Cleans up state of MPI
  - The last call of an MPI process

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;
}
```

Commentary

- #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 urn on each process

Notes on C

- In C:
  - mpi.h must be included by using #include mpi.h
  - 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.

Finding out about the environment

- Two important questions that arise early in a parallel program are:

-How many processes are participating in this computation?

-Which one am I?

- MPI provides functions to answer these questions:

-MPI_Comm_size reports the number of processes.

-MPI_Comm_rank reports the rank, a number between 0 and size-1, identifying the calling process.

Better Hello(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\n", rank, size);
MPI_Finalize();
return 0;
}
```

Some basic concepts

- Processes can be collected into groups.
- Each message is sent in a context, and must be received in the same context.
- A group and context together form a communicator.
- A process is identified by its rank in the group associated with a communicator.
- There is a default communicator whose group contains all initial processes, called MPI_COMM_WORLD.

To use MPI in the ECE systems

- Migrate your ECE account to ENGR domain using the online web tool at

https://account.engr.arizona.edu.

ECE DRACO Cluster

The hostname of the MPI cluster is draco.ece.arizona.edu. The individual hostnames of the cluster systems are as follows:

ursa
alcaid
perseus
altair
cetus
sirius
centauri
pegasus

.rhosts File

In order for MPI to work, users will need the following in the ".rhosts" file in their home directory:

ursa.ece.arizona.edu username
alcaid.ece.arizona.edu username
perseus.ece.arizona.edu username
altair.ece.arizona.edu username
cetus.ece.arizona.edu username
sirius.ece.arizona.edu username
centauri.ece.arizona.edu username
pegasus.ece.arizona.edu username

Compiling and running

- Head file
  - Fortran -- mpif.h
  - C -- mpi.h (*we use C in this presentation)
- Compile:
  - implementation dependent. Typically requires specification of header file directory and MPI library.
  - mpiCC –o destination-filename source-file.c
  - mpiCC filename
- Run:
  - mpirun -np <# proc> <executable>

```c
#include <stdio.h>
#include <string.h>     // this allows us to manipulate text strings
#include "mpi.h"        // this adds the MPI header files to the program

int main(int argc, char* argv[]) {
  int my_rank;        // process rank
  int p;              // number of processes
  int source;         // rank of sender
  int dest;           // rank of receiving process
  int tag = 0;        // tag for messages
  char message[100];  // storage for message
  MPI_Status status;  // stores status for MPI_Recv statements

  // starts up MPI
  MPI_Init(&argc, &argv);
  // finds out rank of each process
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  // finds out number of processes
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  if (my_rank!=0) {
    sprintf(message, "Greetings from process %d!", my_rank);
    dest = 0; // sets destination for MPI_Send to process 0
    // sends the string to process 0
    MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  } else {
    for(source = 1; source < p; source++){
      // receives greeting from each process
      MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
      printf("%s\n", message); // prints out greeting to screen
    }
  }
  MPI_Finalize(); // shuts down MPI
  return 0;
}
```

Result

- mpicc hello.c
- mpirun -np 6 a.out

Greetings from process 1!

Greetings from process 2!

Greetings from process 3!

Greetings from process 4!

Greetings from process 5!

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 souce (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 exrtra 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.

More comment on send and receive

- 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.

Review of Basic MPI routines

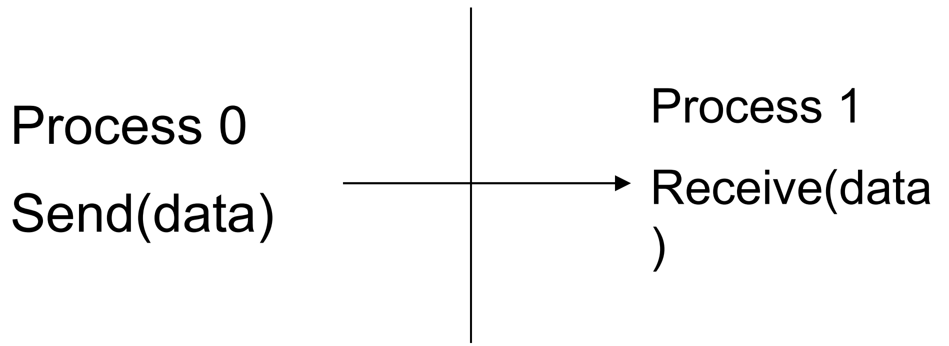
- MPI is used to create parallel programs based on message passing
- Usually the same program is run on multiple processors
- The 6 basic calls in MPI are:
  - MPI_INIT(ierr)
  - MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
  - MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
  - MPI_Send(buffer, count,MPI_INTEGER,destination, tag, MPI_COMM_WORLD, ierr)
  - MPI_Recv(buffer, count, MPI_INTEGER,source,tag, MPI_COMM_WORLD, status,ierr)
  - MPI_FINALIZE(ierr)

Communication Primitives

- Communications on distributed memory computers:
  - Point to Point
  - One to All Broadcast
  - All to All Broadcast
  - One to All Personalized
  - All to All Personalized
  - Shifts
  - Collective Computation

MPI basic send/receive

- We need to fill in the details in



- Things that need specifying:
  - How will “data” be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operation to complete?

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

MPI Data Types in C

C MPI Types	
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	-
MPI_PACKED	-

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.

Broadcast and reduce

- MPI_Bcast(buffer, count, datatype, root, comm)
  - Broadcast the message of length count in buffer from the process root to all other processes in the group. All processes must call with same arguments.
- MPI_Reduce(sbuf, rbuf, count, stype, op, root, comm)
  - Apply the reduction function op to the data of each process in the group (type stype in sbuf) and store the result in rbuf on the root process. op can be a pre-defined function, or defined by the user.

Global Communications in MPI: Broadcast

- All nodes call MPI_Bcast
- One node (root) sends a message all others receive the message
- C
  - MPI_Bcast(&buffer, count, datatype, root, communicator);
- Fortran
  - call MPI_Bcast(buffer, count, datatype, root, communicator, ierr)
- Root is node that sends the message

Global Communications in MPI: Broadcast

- broadcast.c is a parallel program to broadcast data using MPI_Bcast
  - Initialize MPI
  - Have processor 0 broadcast an integer
  - Have all processors print the data
  - Quit MPI

Global Communications in MPI: Broadcast

```
/********************************************************
This is a simple broadcast program in MPI
********************************************************/
#include <stdio.h>
#include "mpi.h"
int main(argc,argv)
int argc;
char *argv[];
{
    int i,myid, numprocs;
    int source,count;
    int buffer[4];
    MPI_Status status;
    MPI_Request request;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    source=0;
    count=4;
    if(myid == source){
      for(i=0;i<count;i++)
        buffer[i]=i;
    }
    MPI_Bcast(buffer,count,MPI_INT,source,MPI_COMM_WORLD);
    for(i=0;i<count;i++)
      printf("%d ",buffer[i]);
    printf("\n");
```

Global Communications in MPI: Reduction

- Used to combine partial results from all processors
- Result returned to root processor
- Several types of operations available. For example summation, maximum etc
- Works on single elements and arrays

Global Communications in MPI: MPI_Reduce

- C
  - **int MPI_Reduce(&sendbuf, &recvbuf, count, datatype, operation,root, communicator)**
- Fortran
  - **call MPI_Reduce(sendbuf, recvbuf, count, datatype, operation,root, communicator, ierr)**
- Parameters
  - Like MPI_Bcast, a root MPI process is specified.
  - Operation is mathematical operation

Global Communications in MPI: MPI_Reduce

MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_LAND	Logical and
MPI_LOR	Logical or
MPI_LXOR	Logical exclusive or
MPI_BAND	Bitwise and
MPI_BOR	Bitwise or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

Example: PI in C - 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) break;
```

Example: PI in C - 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;
}
```

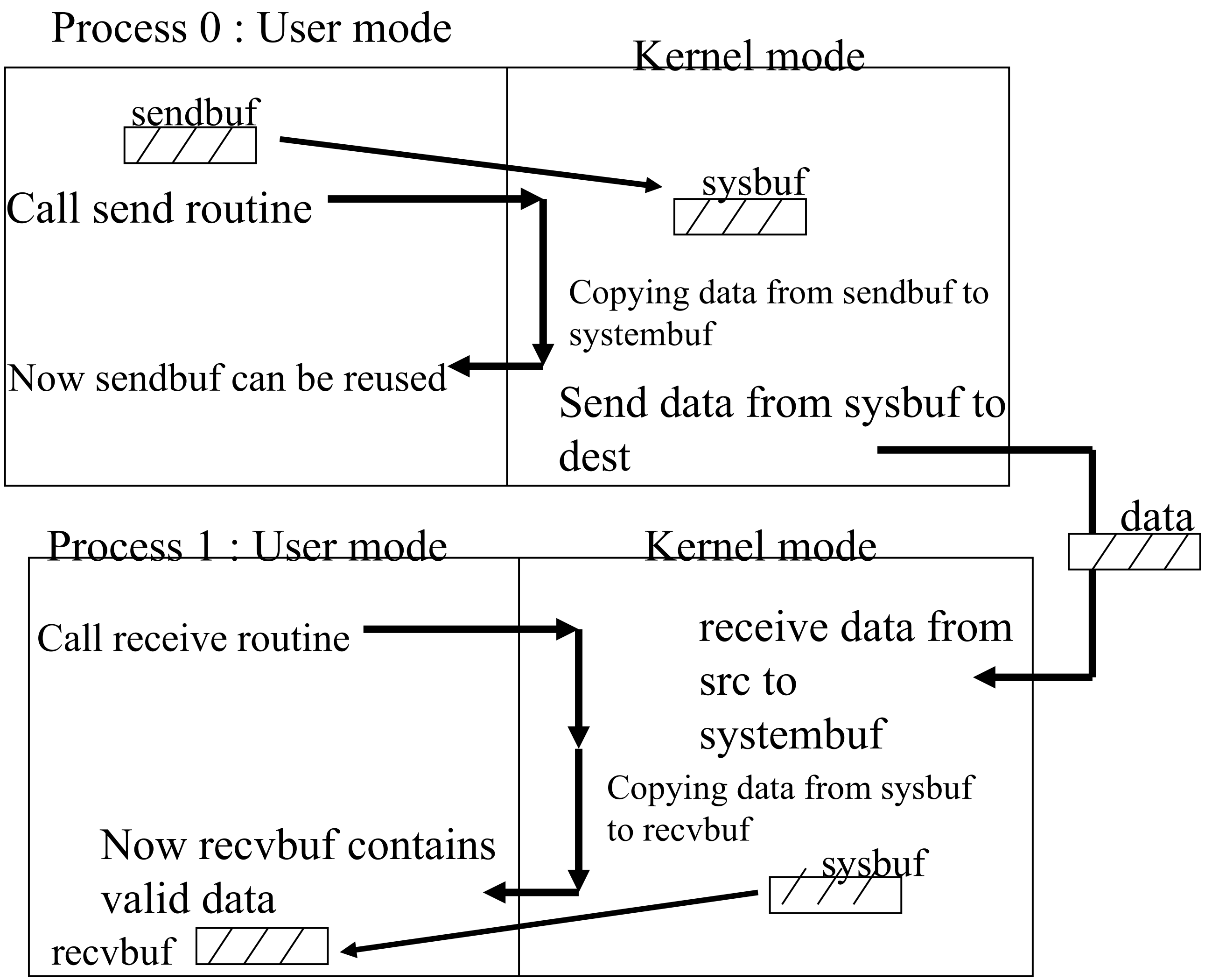
Point to Point Communications in MPI

- Basic operations of Point to Point (PtoP) communication in MPI
- Several steps are involved in the PtoP communication
- Sending process
  - data is copied to the user buffer by the user
  - User calls one of the MPI send routines
  - System copies the data from the user buffer to the system buffer
  - System sends the data from the system buffer to the destination processor

Point to Point Communications in MPI

- Receiving process
  - User calls one of the MPI receive subroutines
  - System receives the data from the source process, and copies it to the system buffer
  - System copies the data from the system buffer to the user buffer
  - User uses the data in the user buffer

Point to Point Communications in MPI



- More information of point to point communication are in the Appendixes

MPI tags

- Message are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message.
- Message can be screened at the receiving end by specifying a specific tag, or not screened by specifying MPI_ANY_TAG as the tag in a receive.
- Some non-MPI message-passing systems have called tags”message types”. MPI calls them tags to avoid confusion with datatype.

MPI_Barrier

- Blocks the caller until all members in the communicator have called it.
- Used as a synchronization tool.
- C
  - `MPI_Barrier(comm )`
- Fortran
  - `Call MPI_BARRIER(COMM, IERROR)`
- Parameter
  - Comm: communicator (often MPI_COMM_WORLD)

Overview of Some Advanced MPI Routines

- Can split MPI communicators (MPI_Comm_split)
- Probe incoming messages (MPI_Probe)
- Asynchronous communication (MPI_Isend, MPI_Irecv, MPI_Wait, MPI_Test etc)
- Scatter different data to different processors (MPI_Scatter), Gather (MPI_Gather)
- MPI_AllReduce, MPI_Alltoall
- Derived data types (MPI_TYPE_STRUCT etc)
- MPI I/O

Group routines

- **MPI_Group_size** returns number of processes in group
- **MPI_Group_rank** returns rank of calling process in group
- **MPI_Group_compare** compares group members and group order
- **MPI_Group_translate_ranks** translates ranks of processes in one group to those in another group
- **MPI_Comm_group** returns the group associated with a communicator
- **MPI_Group_union** creates a group by combining two groups
- **MPI_Group_intersection** creates a group from the intersection of two groups

Group routines ...

- **MPI_Group_difference** creates a group from the difference between two groups
- **MPI_Group_incl** creates a group from listed members of an existing group
- **MPI_Group_excl** creates a group excluding listed members of an existing group
- **MPI_Group_range_incl** creates a group according to first rank, stride, last rank
- **MPI_Group_range_excl** creates a group by deleting according to first rank, stride, last rank
- **MPI_Group_free** marks a group for deallocation

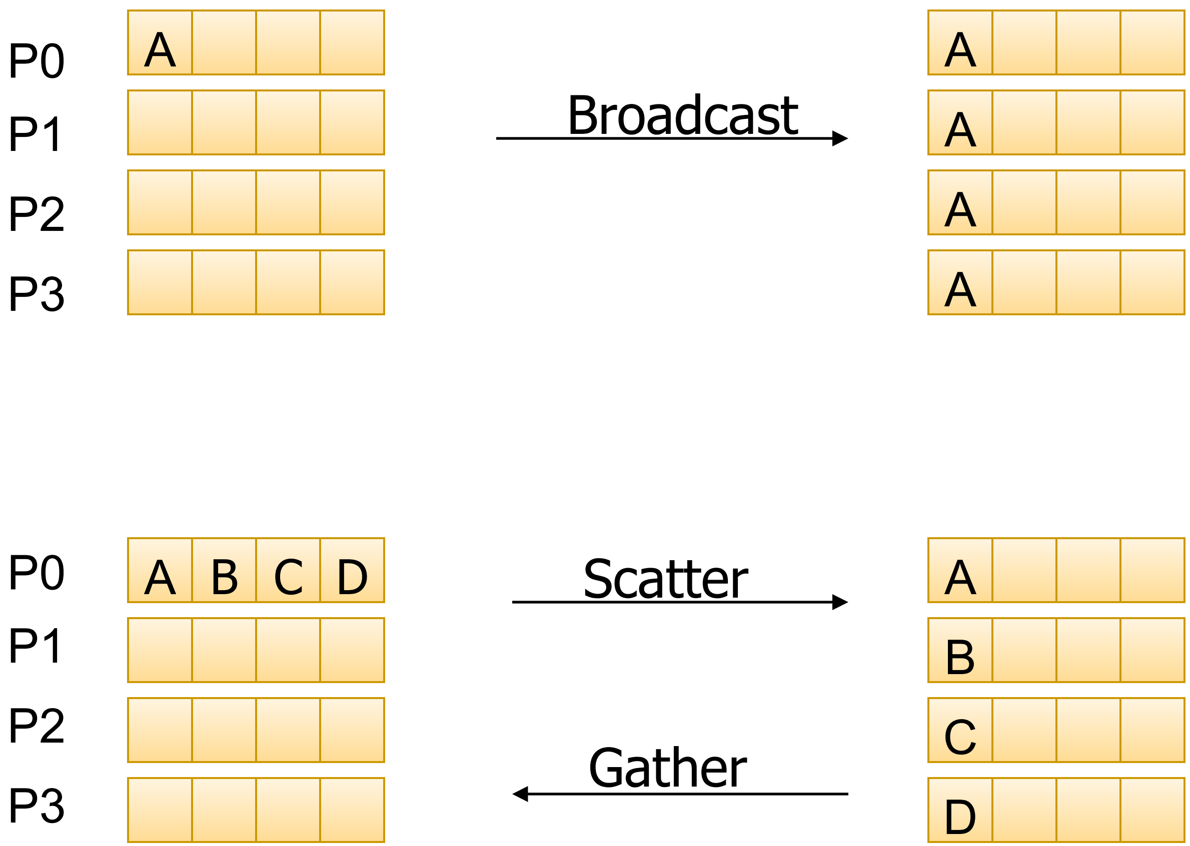
Communicator routines

- **MPI_Comm_size** returns number of processes in communicator's group
- **MPI_Comm_rank** returns rank of calling process in communicator's group
- **MPI_Comm_compare** compares two communicators
- **MPI_Comm_dup** duplicates a communicator
- **MPI_Comm_create** creates a new communicator for a group
- **MPI_Comm_split** splits a communicator into multiple, non-overlapping communicators
- **MPI_Comm_free** marks a communicator for deallocation

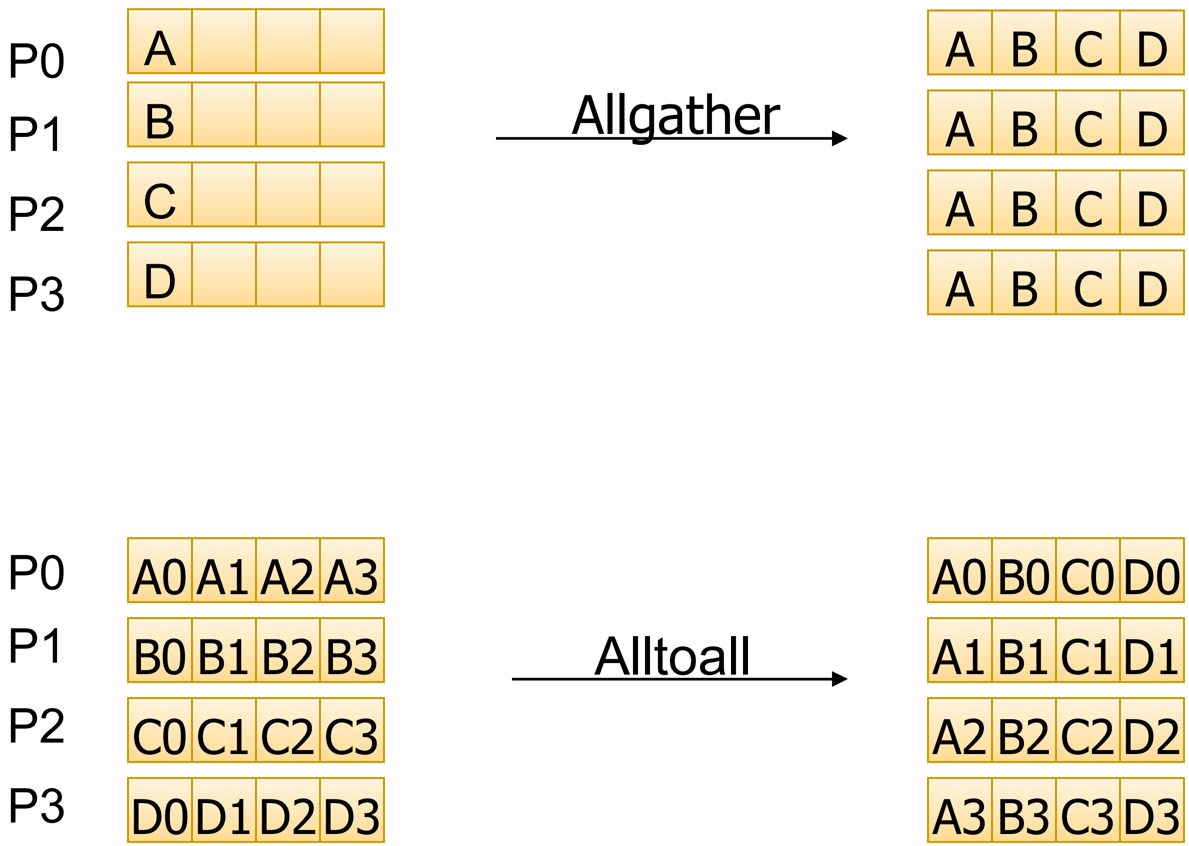
Collective communication

- MPI_Allgather All processes gather messages
- MPI_Allreduce Reduce to all processes
- MPI_Alltoall All processes gather distinct messages
- MPI_Bcast Broadcast a message
- MPI_Gather Gather a message to root
- MPI_Reduce Global reduce operation
- MPI_ReduceScatter Reduce and scatter results
- MPI_Scatter Scatter a message from root
- MPI_Scan Global prefix reduction

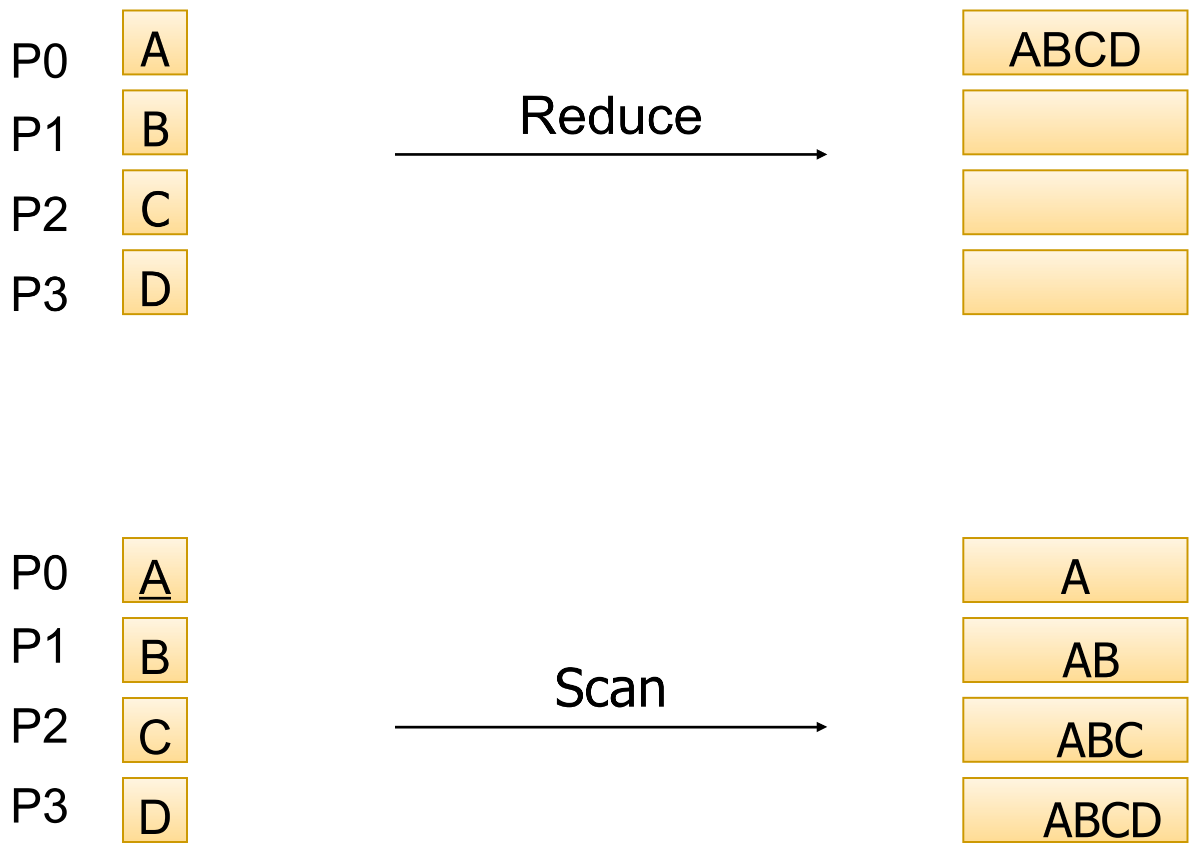
Collective Data Movement



More Collective Data Movement



Collective Computation



Timing

- MPI Wtime() returns the wall-clock time.

```
double start, finish, time;
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
...
...
MPI_Barrier(MPI_COMM_WORLD);
finish = MPI_Wtime();
time = finish - start;
```

MPI Trace Output

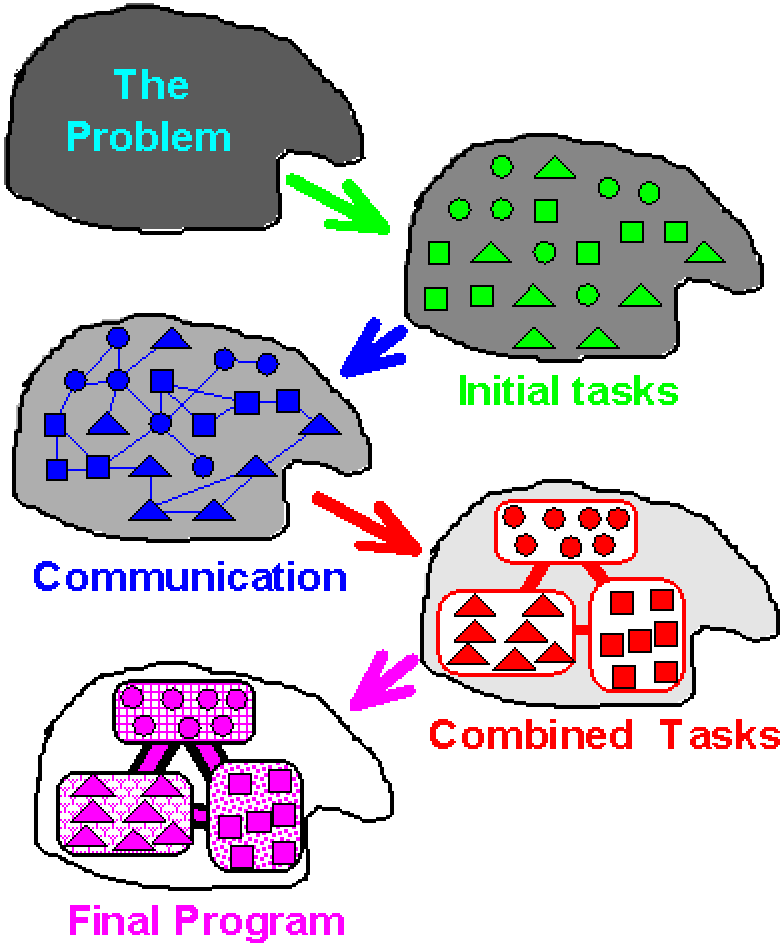
```
-----------------------------------------------------------------
MPI Routine                    #calls     avg. bytes      time(sec)
-----------------------------------------------------------------
MPI_Comm_size                       1            0.0          0.000
MPI_Comm_rank                       1            0.0          0.000
MPI_Send                          500         1024.0          0.001
MPI_Recv                          500         1024.0          0.008
MPI_Barrier                       500            0.0          0.013
-----------------------------------------------------------------
total communication time = 0.022 seconds.
total elapsed time       = 3.510 seconds.
user cpu time            = 3.500 seconds.
system time              = 0.010 seconds.
maximum memory size      = 15856 KBytes.

-----------------------------------------------------------------
Message size distributions:

MPI_Send                  #calls    avg. bytes      time(sec)
                             500        1024.0          0.001

MPI_Recv                  #calls    avg. bytes      time(sec)
                             500        1024.0          0.008



-----------------------------------------------------------------
Call Graph Section:

communication time = 0.022 sec, parent = poisson
   MPI Routine                    #calls        time(sec)
```

MPI-2

- MPI-2 new topics:
  - process creation and management, including client/server routines
  - one-sided communications (put/get, active messages)
  - extended collective operations
  - external interfaces
  - I/O

Designing MPI programs

- Partitioning
  - Before tackling MPI
- Communication
  - Many point to collective operations
- Agglomeration
  - Needed to produce MPI processes
- Mapping
  - Handled by MPI



MPI

- Pros:
  - Very portable
  - Requires no special compiler
  - Requires no special hardware but can make use of high performance hardware
  - Very flexible -- can handle just about any model of parallelism
  - No shared data! (You don't have to worry about processes "treading on each other's data" by mistake.)
  - Can download free libraries for your Linux PC!
  - Forces you to do things the "right way" in terms of decomposing your problem.

- Cons:
  - All-or-nothing parallelism (difficult to incrementally parallelize existing serial codes)
  - No shared data! Requires distributed data structures
  - Could be thought of assembler for parallel computing -- you generally have to write more code
  - Partitioning operations on distributed arrays can be messy.

MPI v.s. OpenMP

- Message passing v.s. shared data
- Processes v.s. Threads
- MPI has no work sharing structure.

OpenMP

- Pros:
  - Incremental parallelism -- can parallelize existing serial codes one bit at a time
  - Quite simple set of directives
  - Shared data!
  - Partitioning operations on arrays is very simple.
- Cons:
  - Requires proprietary compilers
  - Requires shared memory multiprocessors
  - Shared data!
  - Having to think about what data is shared and what data is private
  - Cannot handle models like master/slave work allocation (yet)
  - Generally not as scalable (more synchronization points)
  - Not well-suited for non-trivial data structures like linked lists, trees etc

- Homework #1 (programming) will be posted.
- Due: September 14 before the class

Appendix

Unidirectional Communication

- Blocking send and blocking receive
  - ```
          if (myrank == 0) then
            call MPI_Send(...)
        elseif (myrank == 1) then
            call MPI_Recv(....)
        endif
    ```
- Non-blocking send and blocking receive
  - ```
          if (myrank == 0) then
            call MPI_ISend(...)
            call MPI_Wait(...)
        else if (myrank == 1) then
            call MPI_Recv(....)
        endif
    ```

Unidirectional Communication

- Blocking send and non-blocking recv

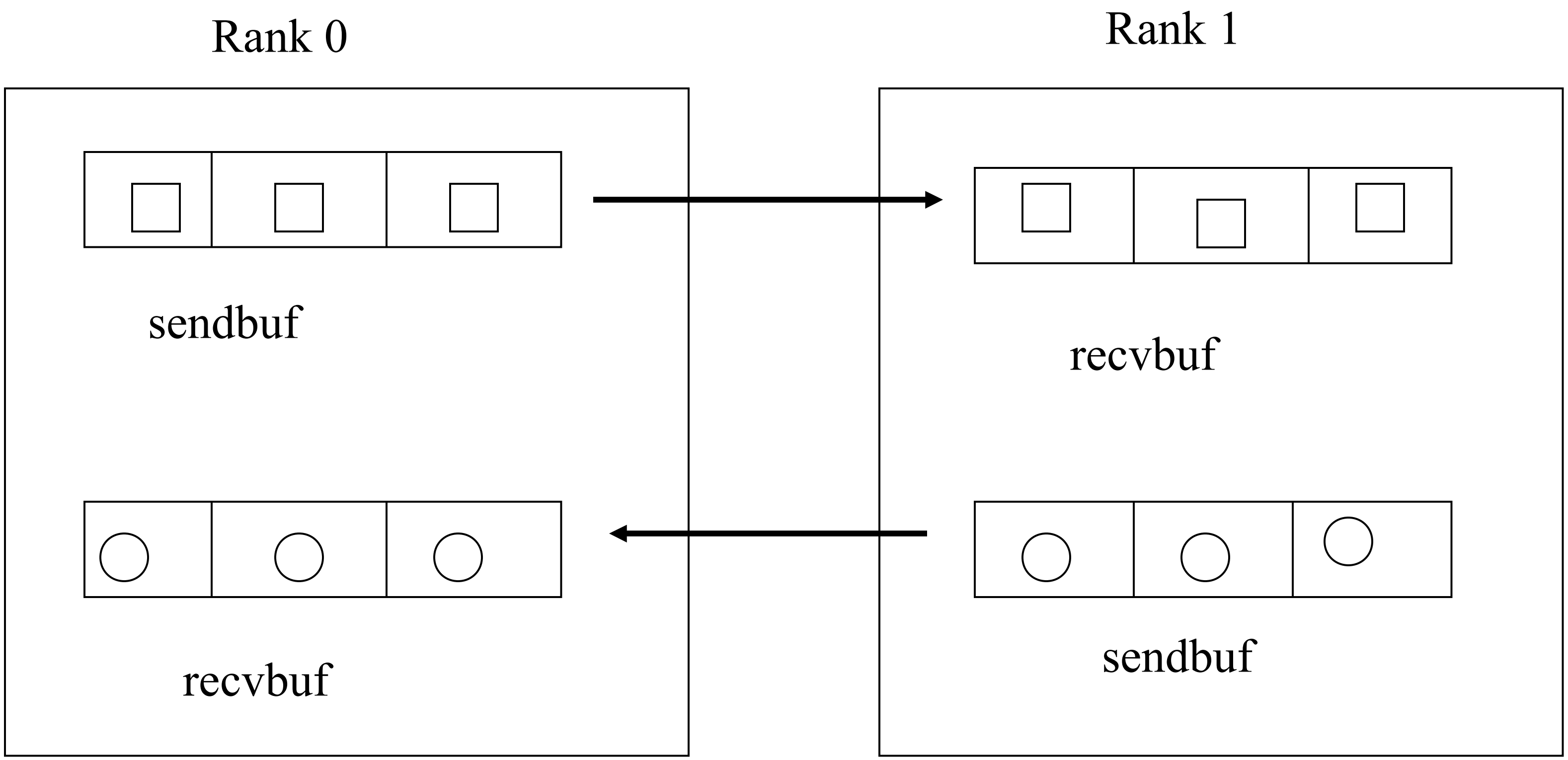
```
if (myrank == 0 ) then
      call MPI_Send(…..)

elseif (myrank == 1) then
      call MPI_Irecv (…)
      call MPI_Wait(…)

endif
```

- Non-blocking send and non-blocking recv

```
            if (myrank == 0 ) then
            call MPI_Isend (…)
            call MPI_Wait (…)
      elseif (myrank == 1) then
            call MPI_Irecv (….)
            call MPI_Wait(..)
      endif
```

Bidirectional Communication

- Need to be careful about deadlock when two processes exchange data with each other
- Deadlock can occur due to incorrect order of send and recv or due to limited size of the system buffer



Bidirectional Communication

- Case 1 : both processes call send first, then recv

```
if (myrank == 0 ) then
    call MPI_Send(....)
    call MPI_Recv (...)
elseif (myrank == 1) then
    call MPI_Send(....)
    call MPI_Recv(....)
endif
```

- No deadlock as long as system buffer is larger than send buffer
- Deadlock if system buffer is smaller than send buf
- If you replace MPI_Send with MPI_Isend and MPI_Wait, it is still the same
- Moral : there may be error in coding that only shows up for larger problem size

Bidirectional Communication

- Case 2 : both processes call recv first, then send

```
if (myrank == 0 ) then
    call MPI_Recv(….)
    call MPI_Send (…)
elseif (myrank == 1) then
    call MPI_Recv(….)
    call MPI_Send(….)
endif
```

- The above will always lead to deadlock (even if you replace MPI_Send with MPI_Isend and MPI_Wait)

Bidirectional Communication

- The following code can be safely executed

```
if (myrank == 0 ) then
    call MPI_Irecv(....)
    call MPI_Send (...)
    call MPI_Wait(...)
elseif (myrank == 1) then
    call MPI_Irecv(....)
    call MPI_Send(....)
    call MPI_Wait(....)
endif
```

Bidirectional Communication

- Case 3 : one process call send and recv in this order, and the other calls in the opposite order

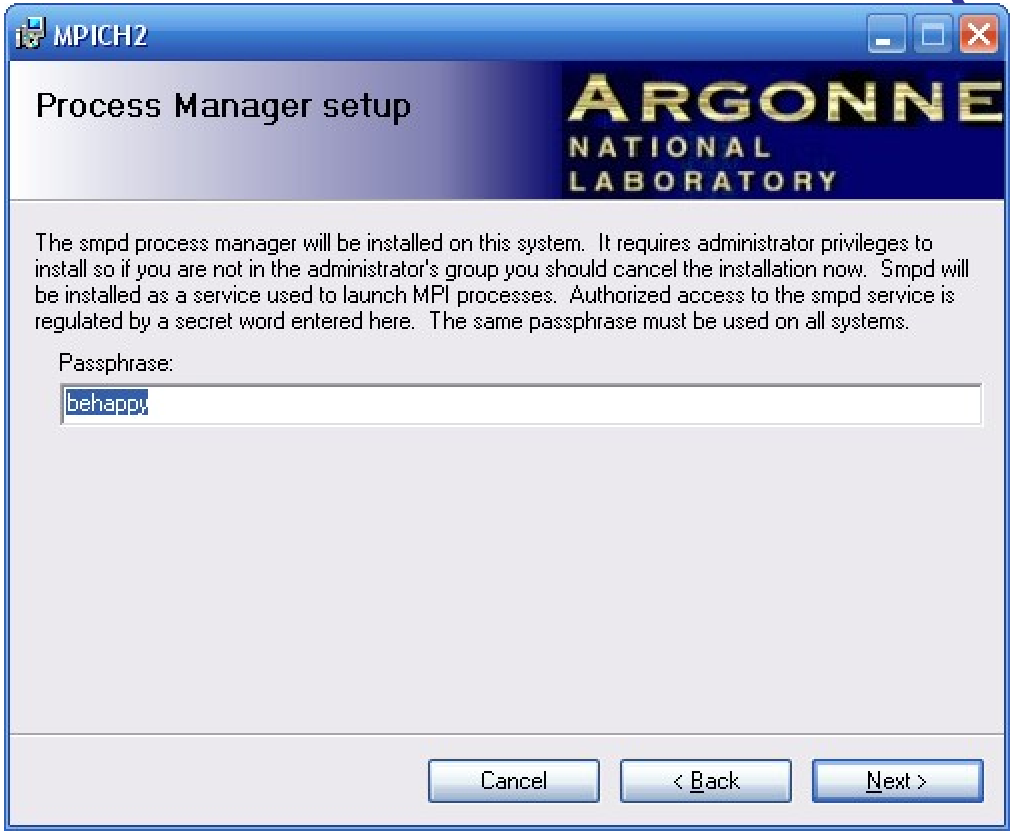
```
if (myrank == 0 ) then
     call MPI_Send(....)
     call MPI_Recv(...)
elseif (myrank == 1) then
     call MPI_Recv(....)
     call MPI_Send(....)
endif
```

- The above is always safe
- You can replace both send and recv on both processor with Isend and Irecv

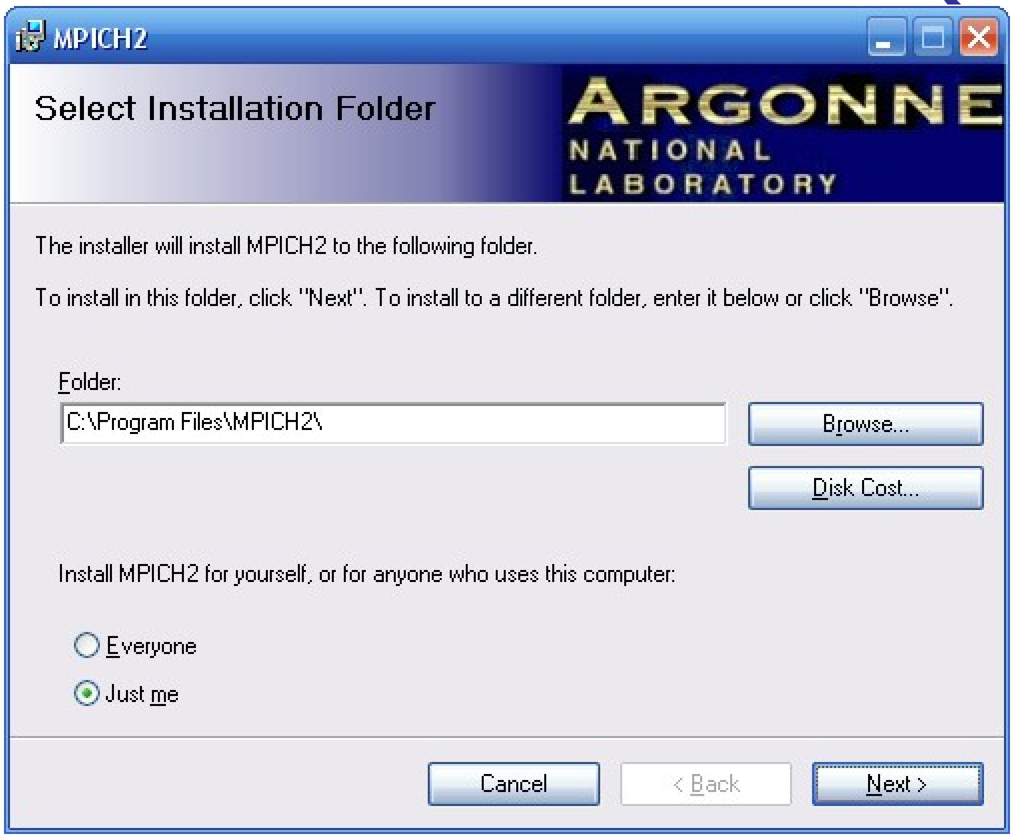
Where to get MPI library?

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

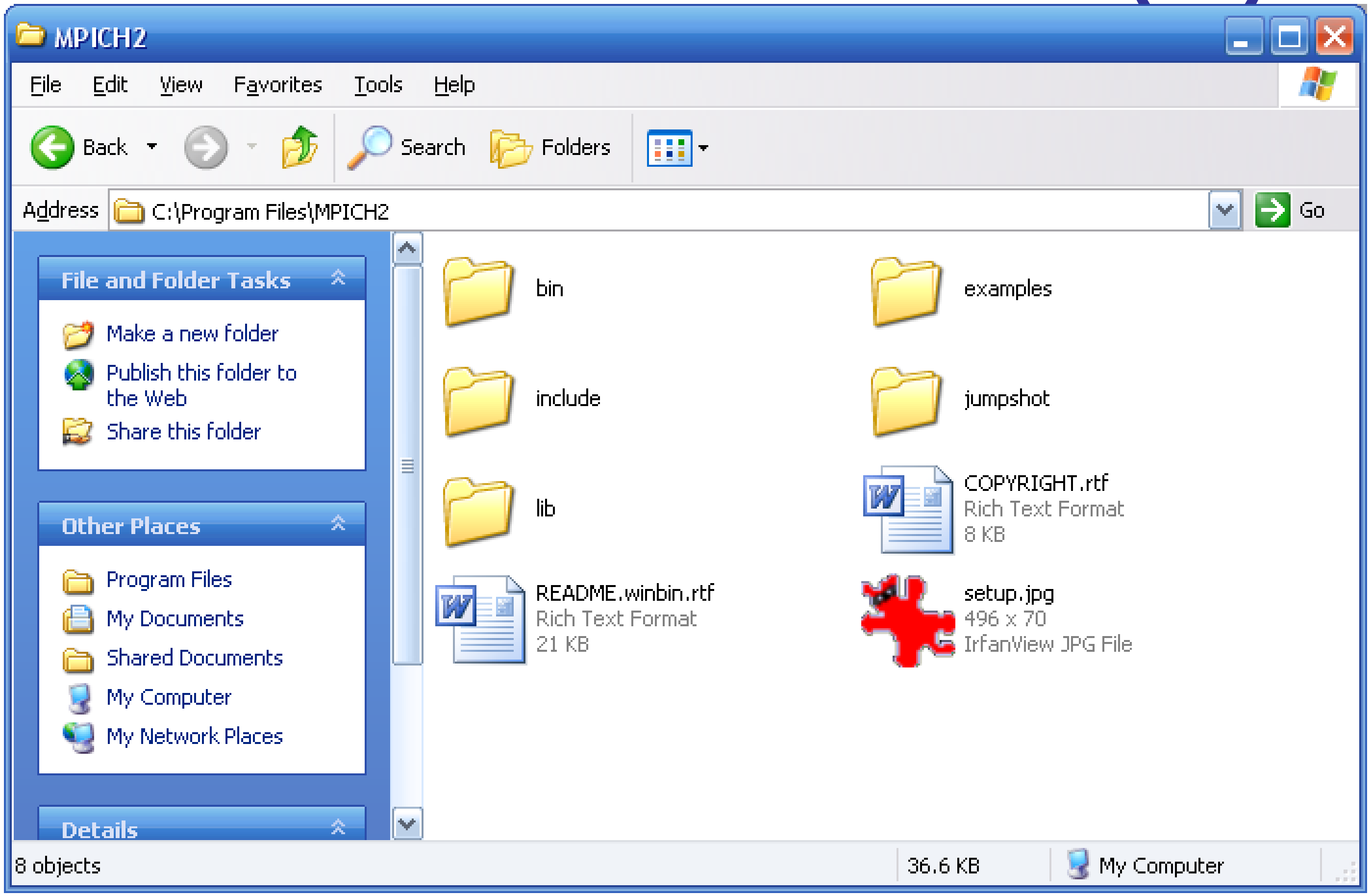
Step By Step Installation of MPICH on windows XP(1)



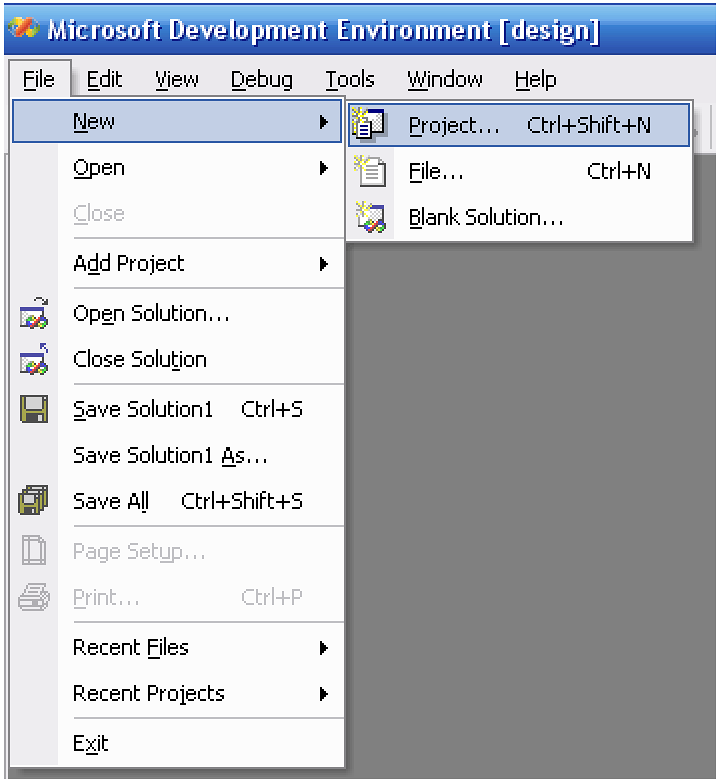
Step By Step Installation of MPICH on windows XP(2)



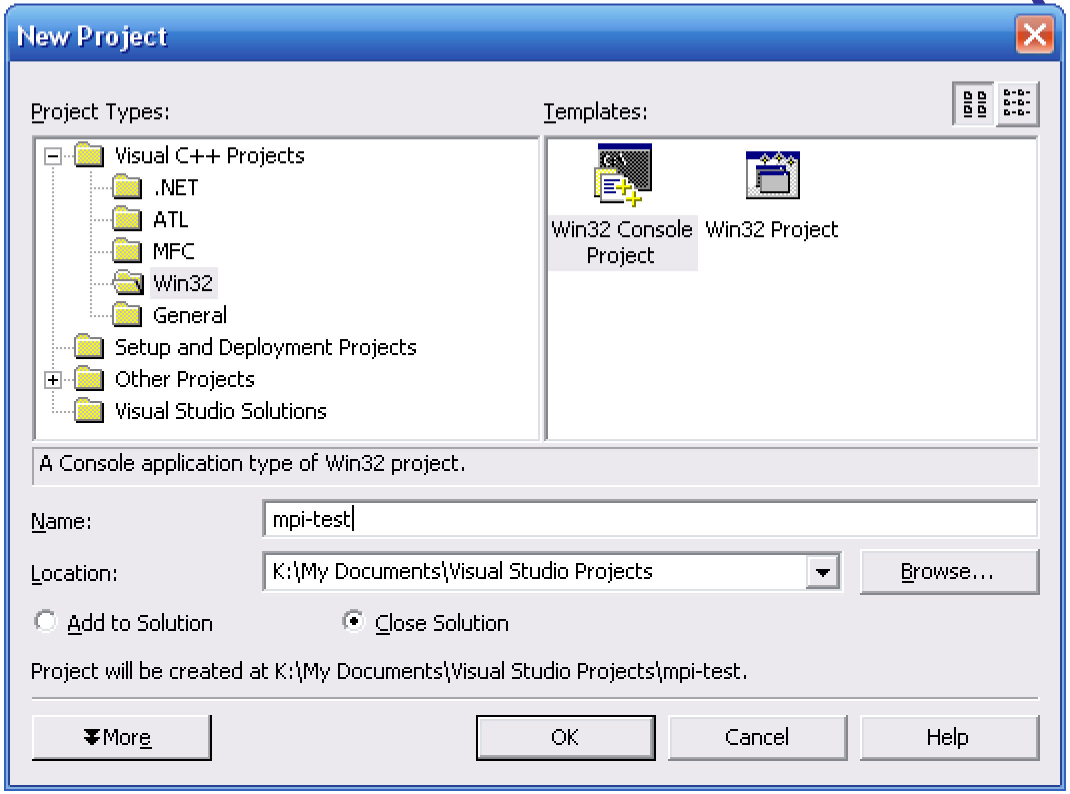
Step By Step Installation of MPICH on windows XP(3)



Step By Step Installation of MPICH on windows XP(4)



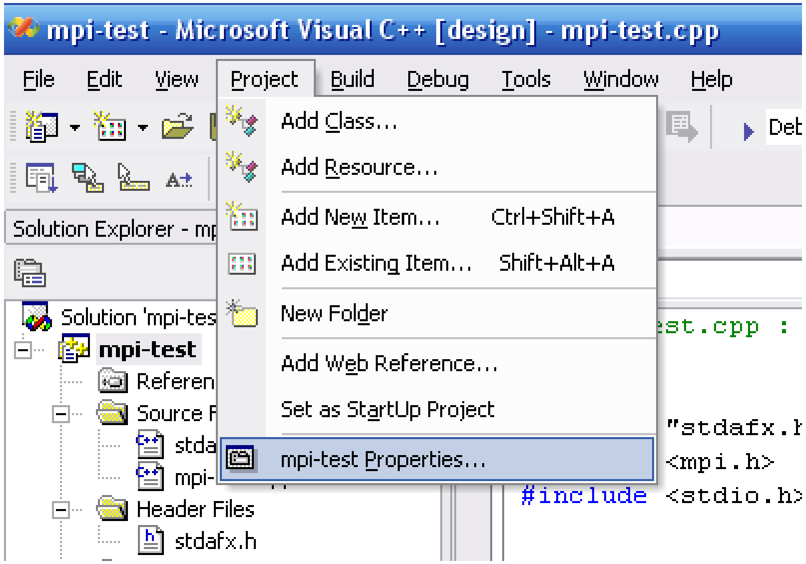
Step By Step Installation of MPICH on windows XP(5)



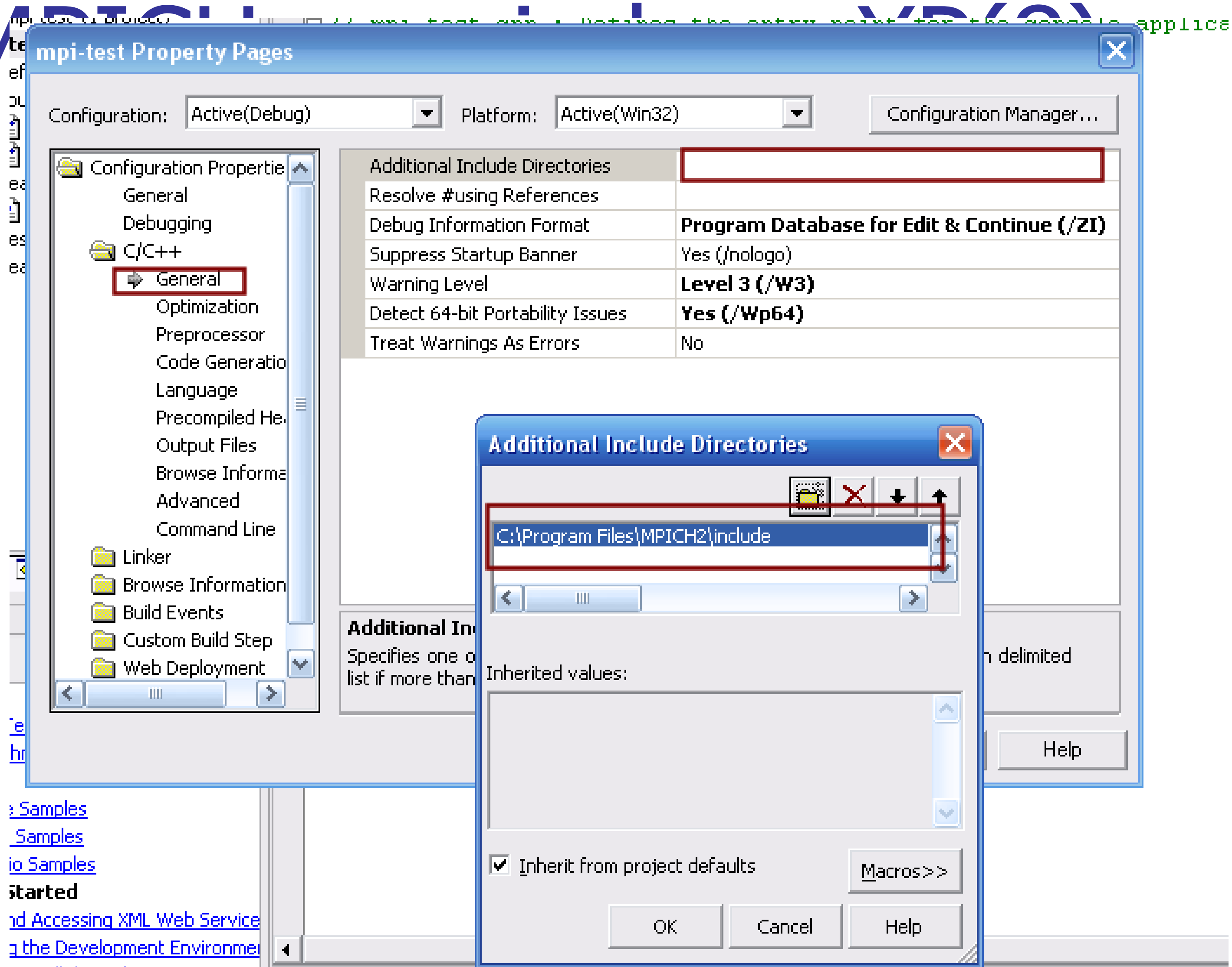
Step By Step Installation of MPICH on windows XP(6)

```
// 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\n", rank, size);
	MPI_Finalize();
	return 0;
}
```

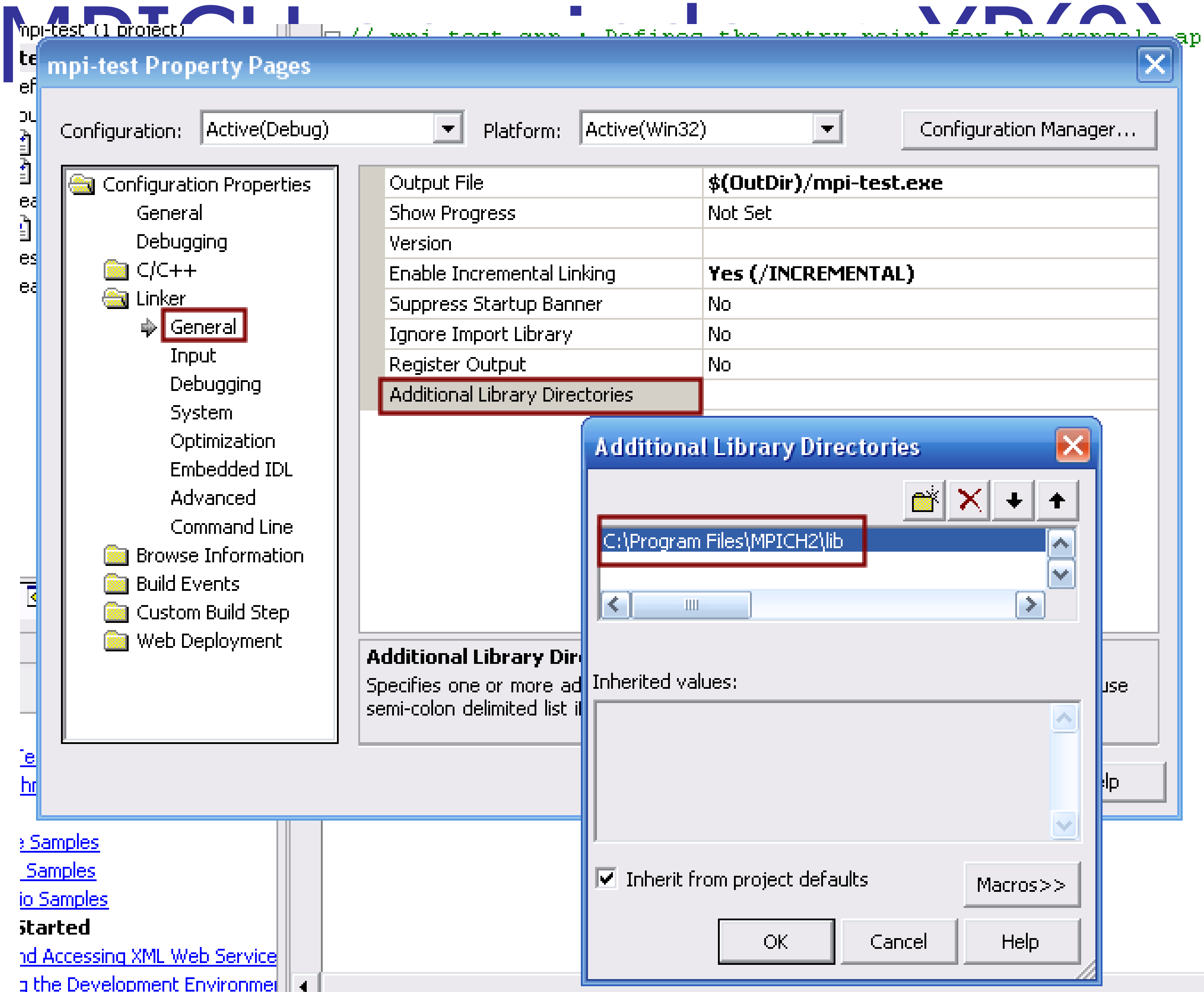
Step By Step Installation of MPICH on windows XP(7)



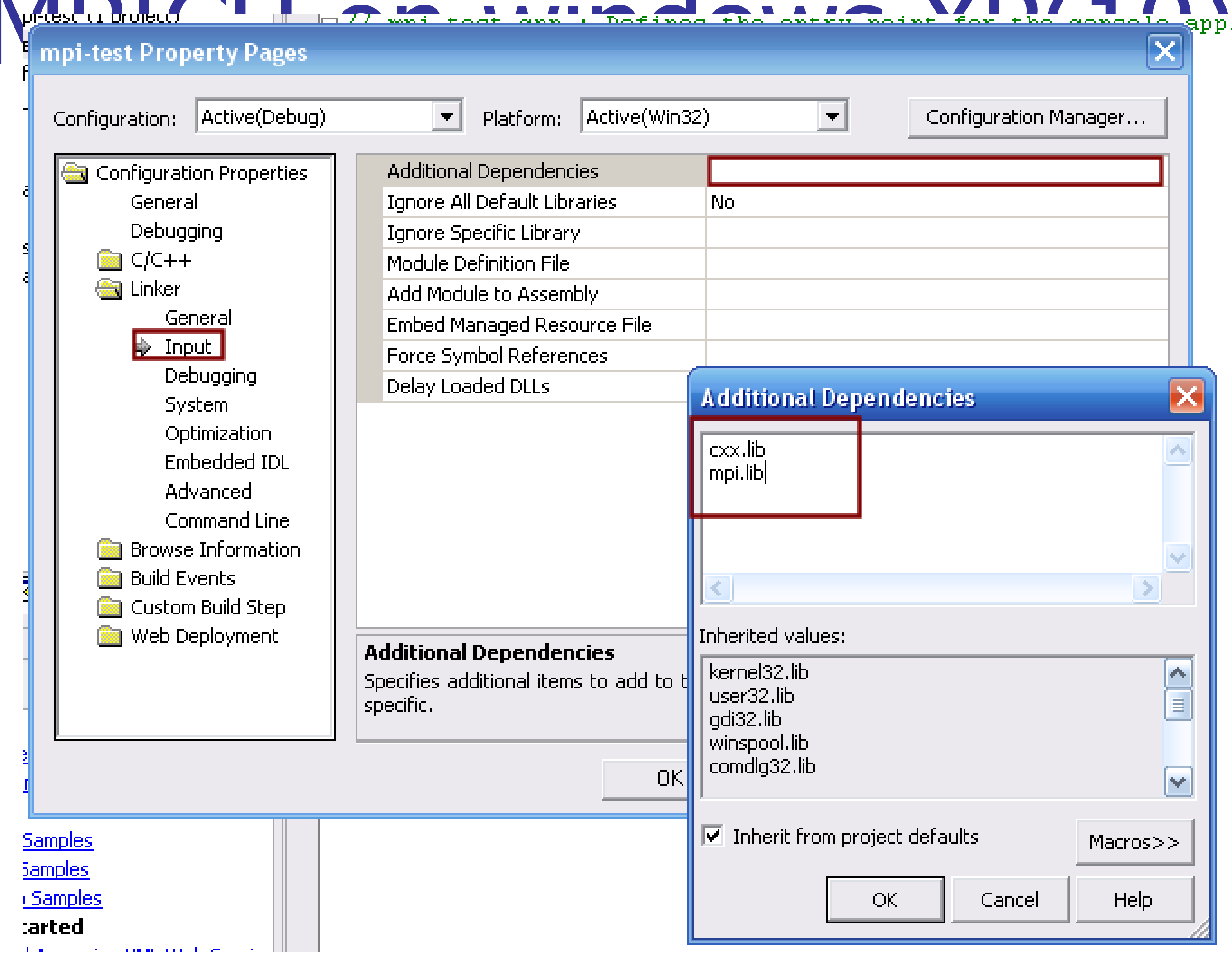
Step By Step Installation of



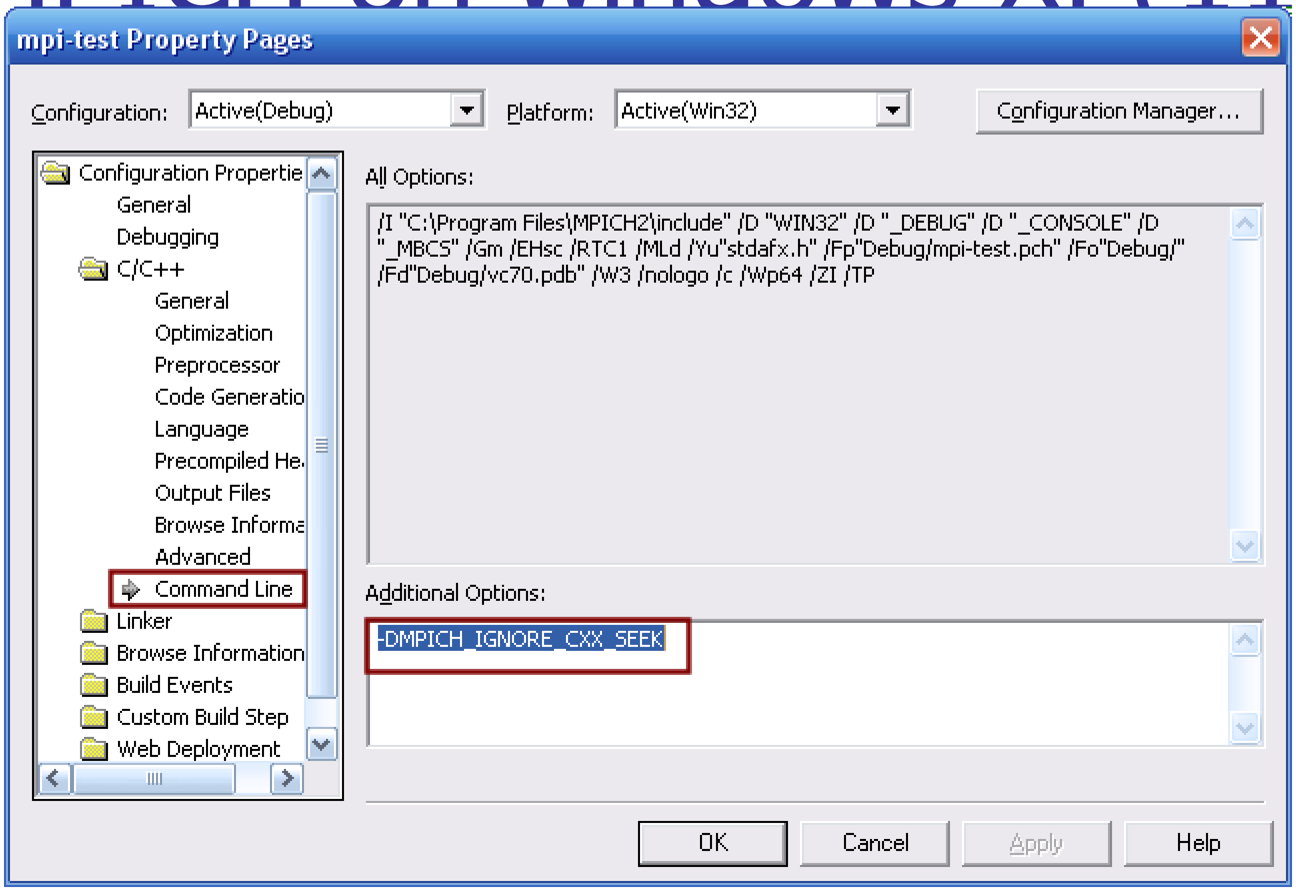
Step By Step Installation of



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
```

Old Compile MSVS 6

Program with MPI and play with it

- MPICH-1.2.4 for windows 2000 has installed in ECE226.
- On every machine, please refer to c:\Program Files\MPICH\www\nt to find the HTML help page on how to run and program in the environment of Visual C++ 6.0
- Examples have been installed under c:\Program Files\MPICH\SDK\Examples

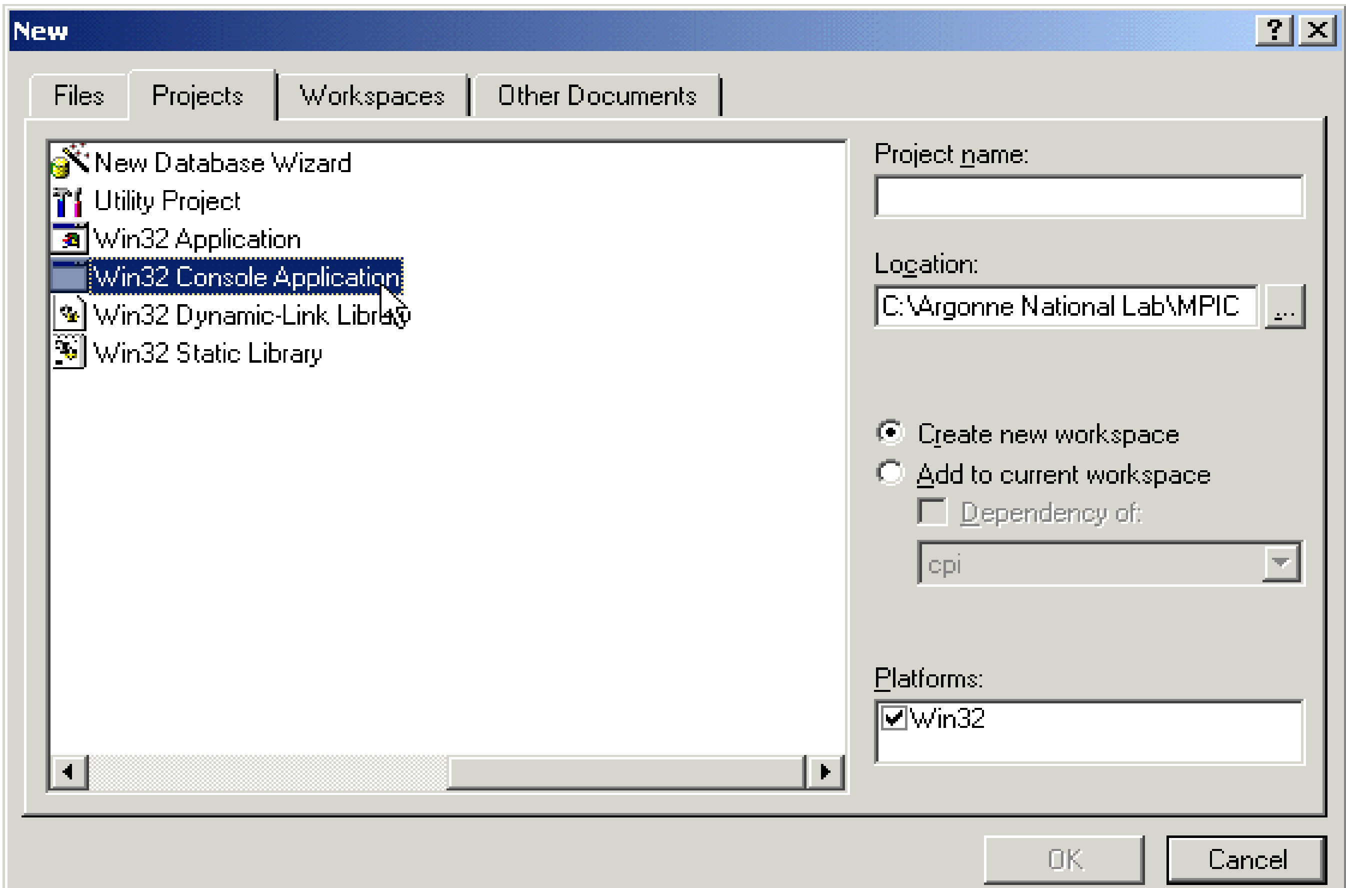
How to run the example

1. Open the MSDEV workspace file found in MPICH\SDK\Examples\nt\examples.dsw
2. Build the Debug target of the cpi project
3. Copy MPICH\SDK\Examples\nt\Debug\cpi.exe to a shared directory. (use copy/paste to \\pearl\files\mpi directory)Open a command prompt and change to the directory where you placed cpi.exe
4. Execute mpirun.exe –np 4 cpi
5. In order to set path in DOS, in this case, use command: set PATH=%PATH%;c:\Program Files\MPICH\mpd\bin

Create your own project

1. Open MS Developer Studio - Visual C++
2. Create a new project with whatever name you want in whatever directory you want. The easiest one is a Win32 console application with no files in it.

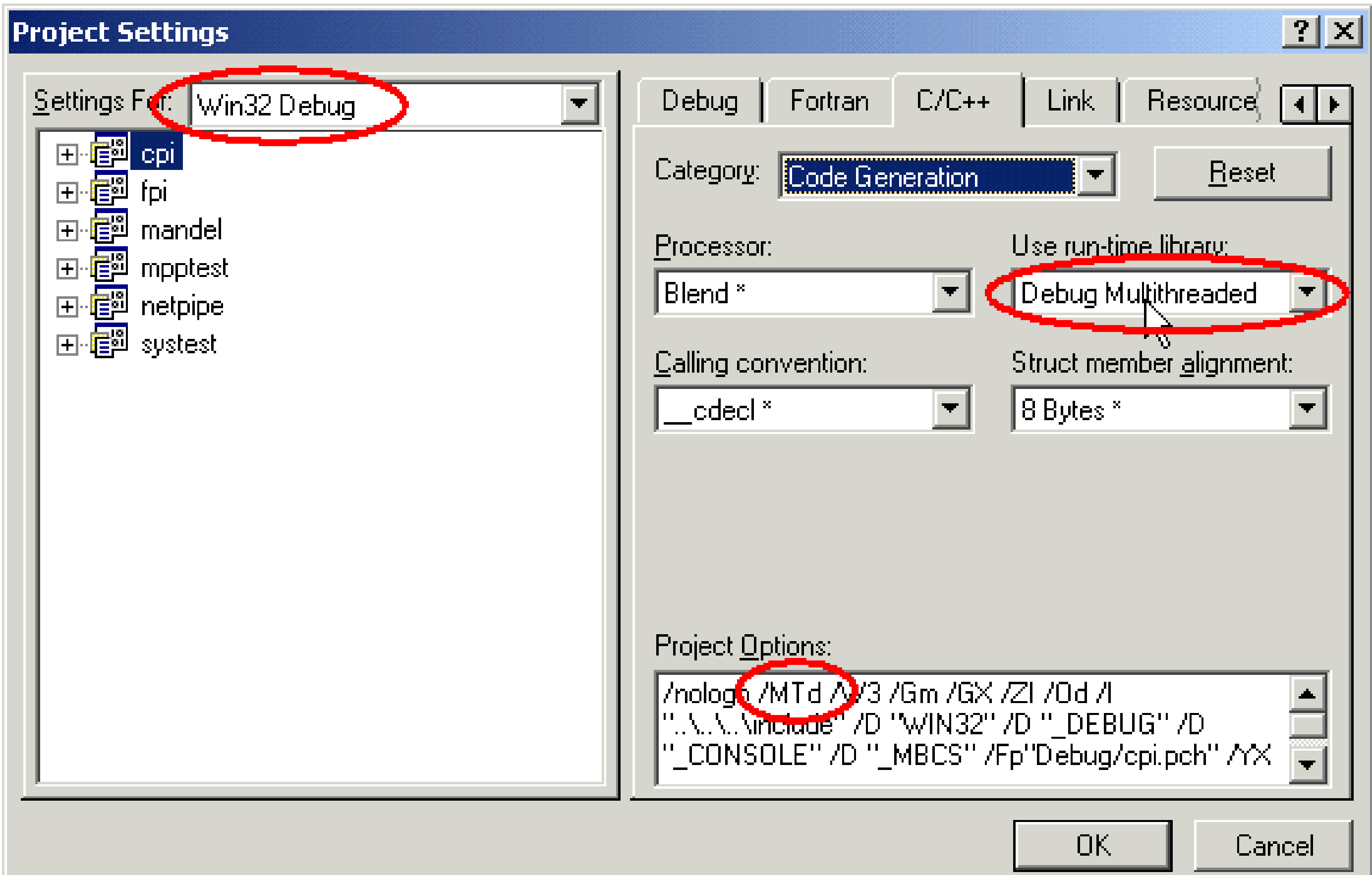
Create your own project



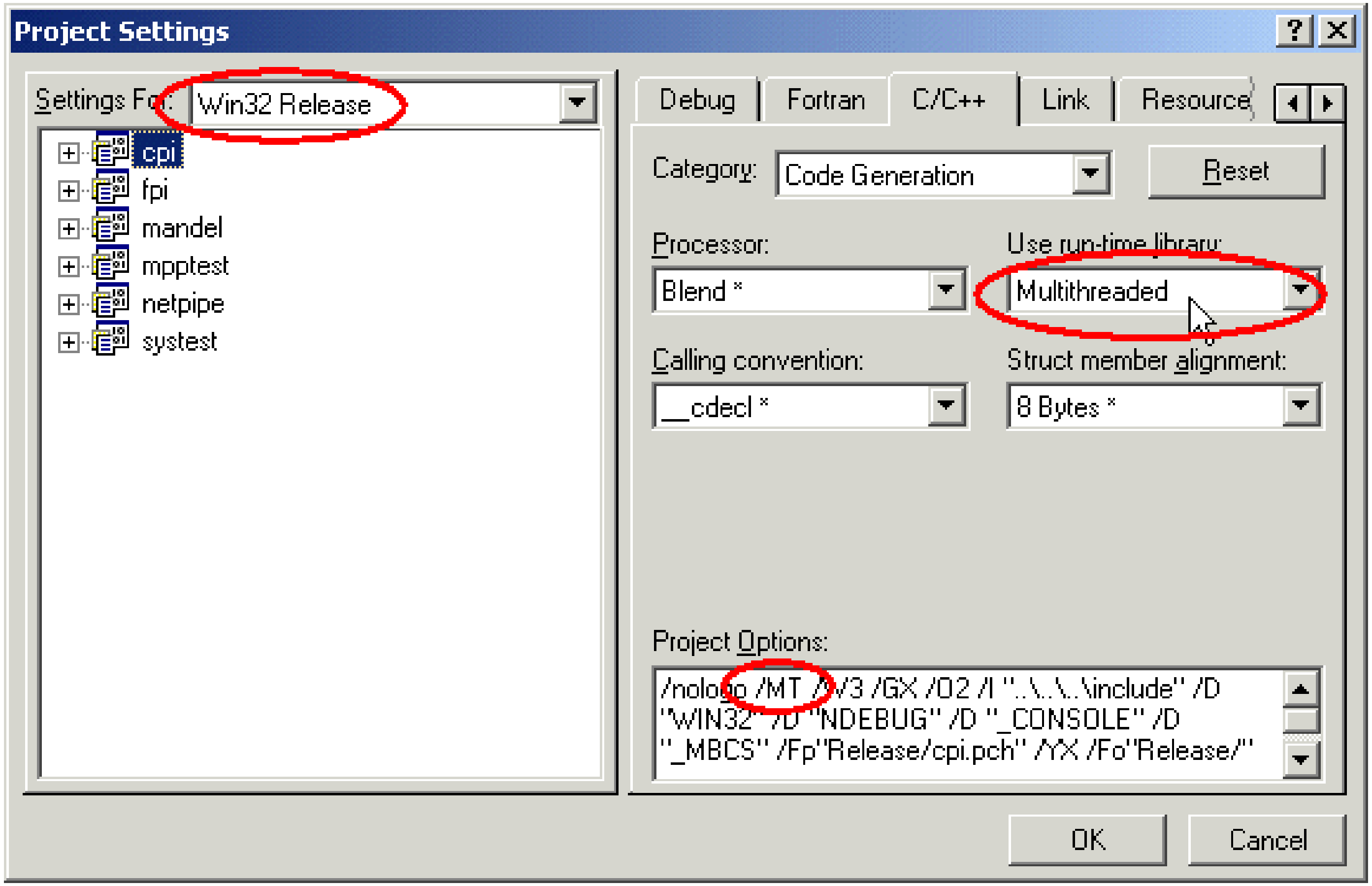
continue

3. Finish the new project wizard.
4. Go to Project->Settings or hit Alt F7 to bring up the project settings dialog box.
5. Change the settings to use the multithreaded libraries.
   Change the settings for both Debug and Release targets.

continue


Project Settings
Settings For: Win32 Debug
cpi
fpi
mandel
mpptest
netpipe
systest
Debug
Fortran
C/C++
Link
Resource
Category: Code Generation
Reset
Processor:
Blend *
Use run-time library:
Debug Multithreaded
Calling convention:
__cdecl *
Struct member alignment:
8 Bytes *
Project Options:
/nologo /MTd /W3 /Gm /GX /ZI /Od /I
"..\..\..\include" /D "WIN32" /D "_DEBUG" /D
"_CONSOLE" /D "_MBCS" /Fp"Debug/cpi.pch" /YX
OK
Cancel

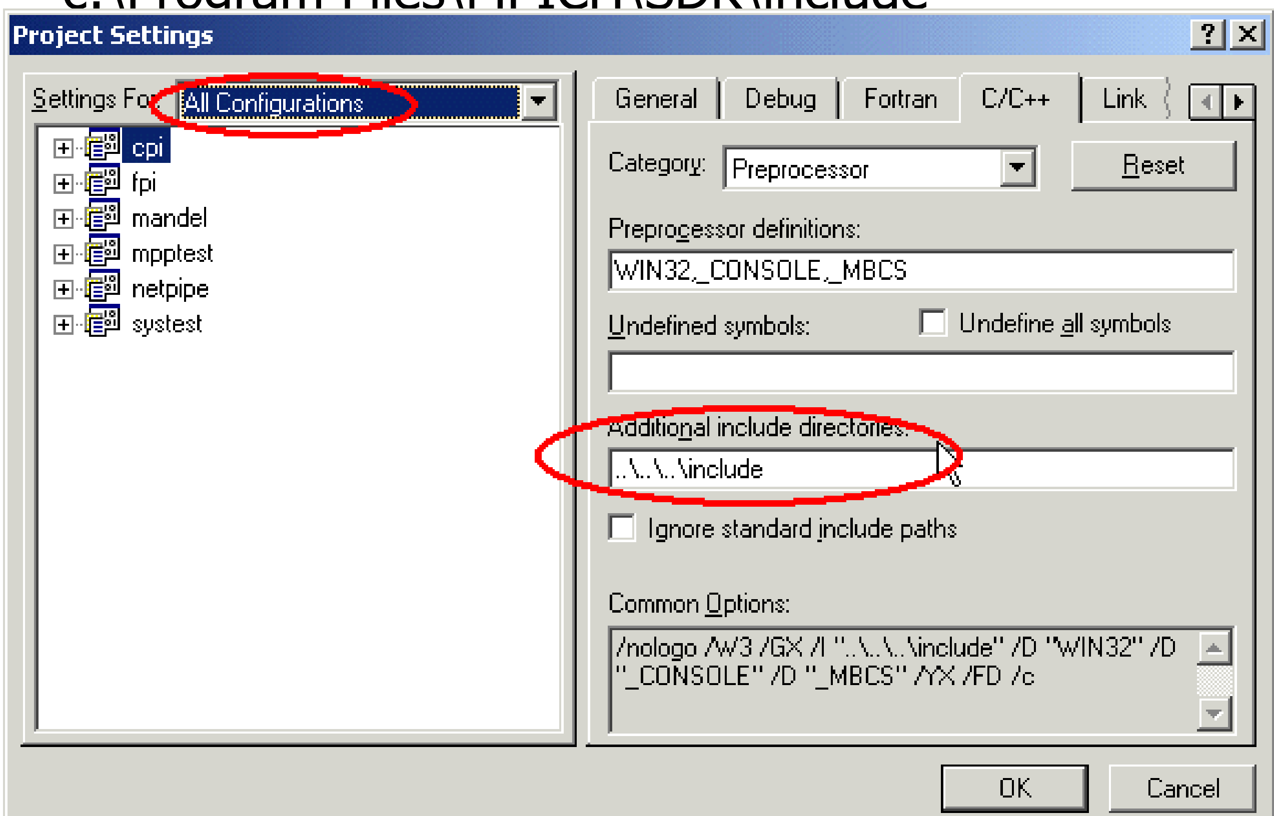
continue



Project Settings
Settings For: Win32 Release
cpi
fpi
mandel
mpptest
netpipe
systest
Debug
Fortran
C/C++
Link
Resource
Category: Code Generation
Reset
Processor:
Blend *
Use run-time library:
Multithreaded
Calling convention:
__cdecl *
Struct member alignment:
8 Bytes *
Project Options:
/nologo /MT /W3 /GX /O2 /I "..\..\..\include" /D "WIN32" /D "NDEBUG" /D "_CONSOLE" /D "_MBCS" /Fp"Release/cpi.pch" /YX /Fo"Release/"
OK
Cancel

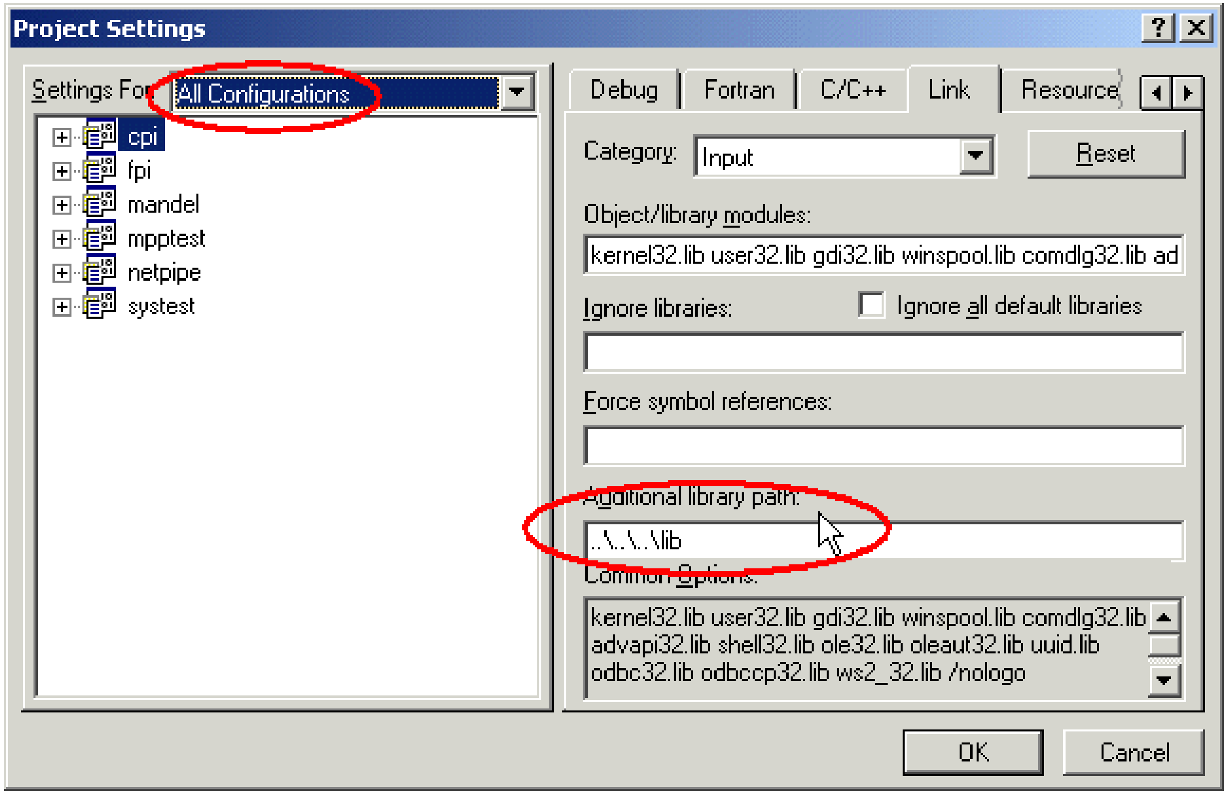
continue

6. Set the include path for all target configurations: This should be c:\Program Files\MPICH\SDK\include



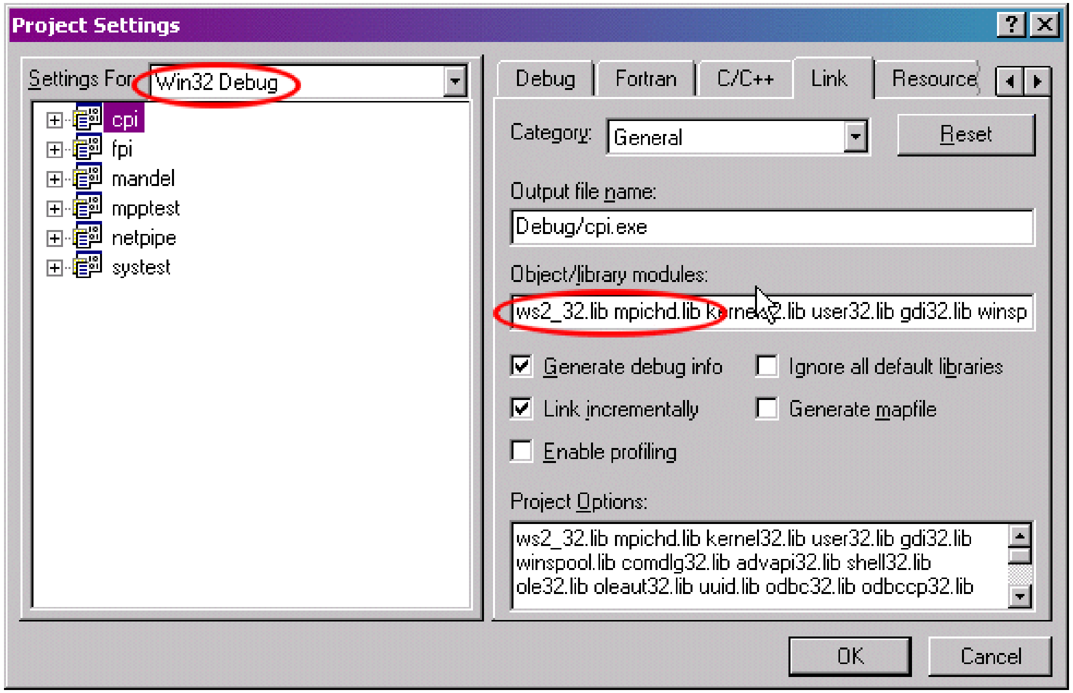
continue

7. Set the lib path for all target configurations: This should be c:\Program Files\MPICH\SDK\lib

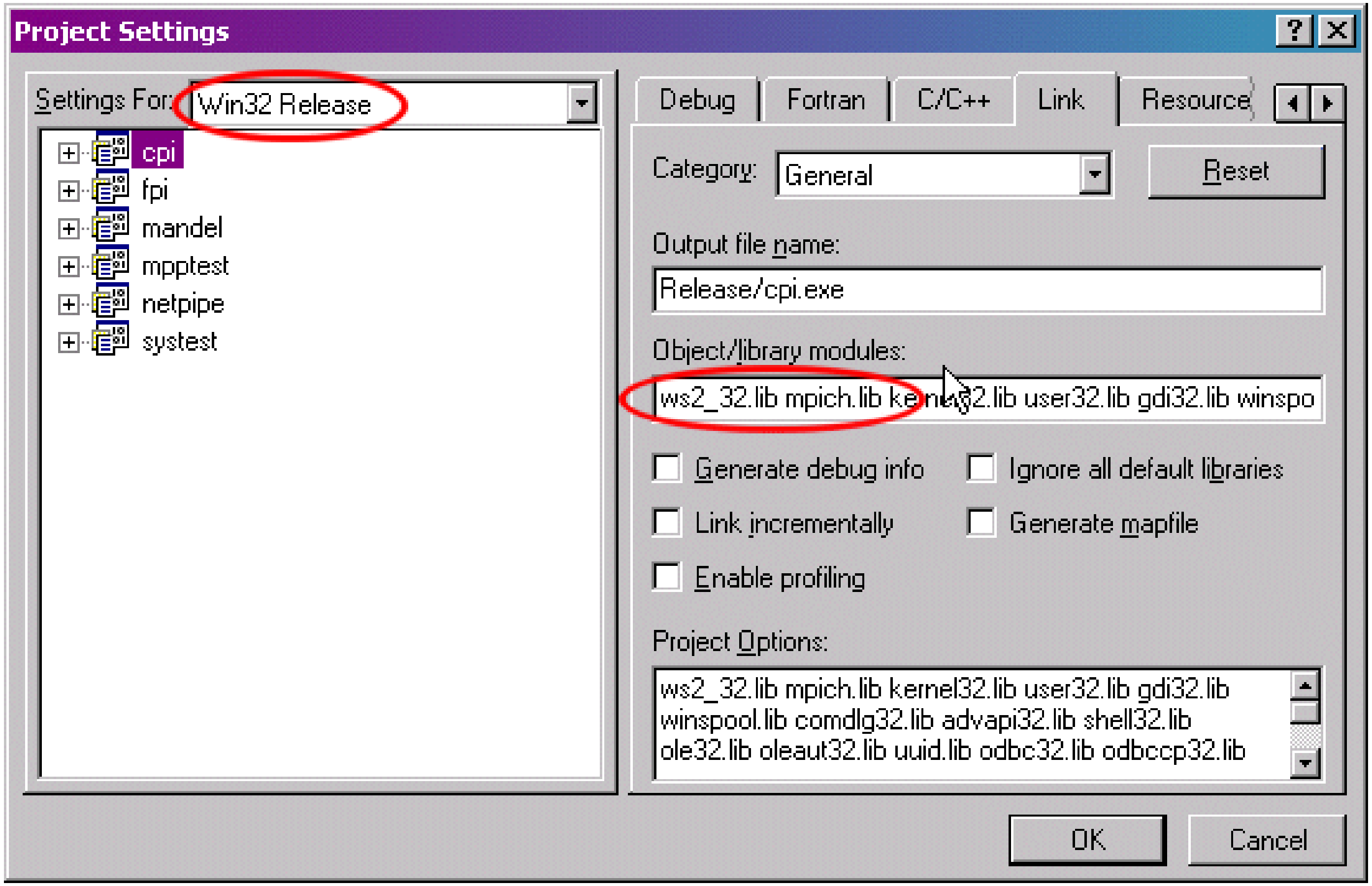


continue

8. Add the ws2_32.lib library to all configurations (This is the Microsoft Winsock2 library. It's in your default library path).
   Add mpich.lib to the release target and mpichd.lib to the debug target.

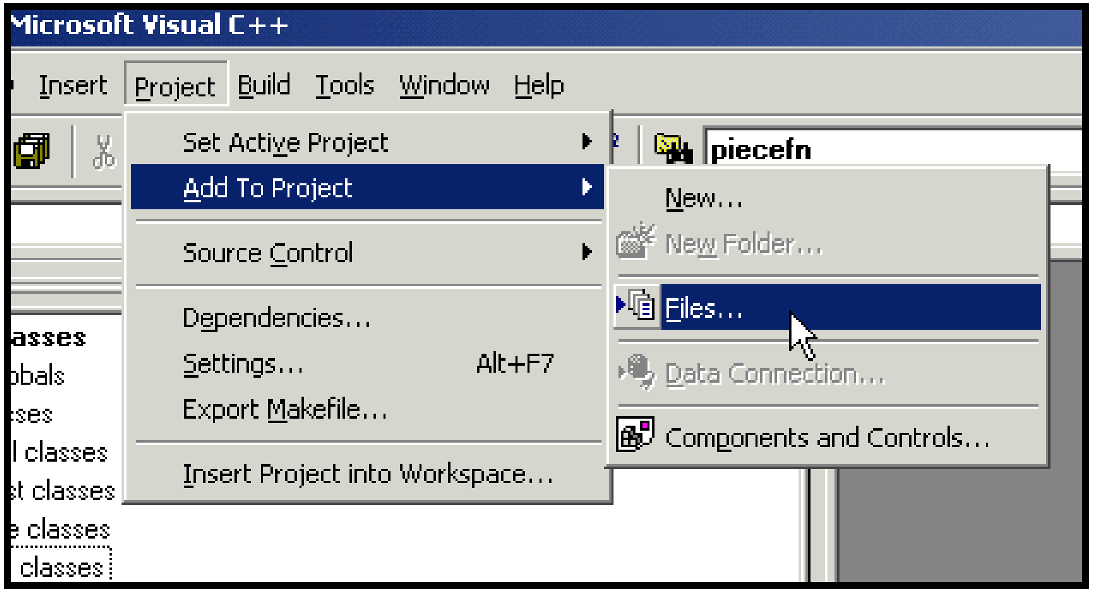


continue


Project Settings
Settings For: Win32 Release
cpi
fpi
mandel
mpptest
netpipe
systest
Debug
Fortran
C/C++
Link
Resource
Category: General
Reset
Output file name:
Release/cpi.exe
Object/library modules:
ws2_32.lib mpich.lib kernel32.lib user32.lib gdi32.lib winspo
Generate debug info
Ignore all default libraries
Link incrementally
Generate mapfile
Enable profiling
Project Options:
ws2_32.lib mpich.lib kernel32.lib user32.lib gdi32.lib
winspool.lib comdlg32.lib advapi32.lib shell32.lib
ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib
OK
Cancel

continue

9. Close the project settings dialog box.
10. Add your source files to the project



Useful MPI function to test your program

MPI_Get_processor_name(name, resultlen)

- **name** is a unique specifier for the actual node. (string)
- **resultlen** is length of the result returned in name(integer)

This routine returns the name of the processor on which it was called at the moment of the call. The number of characters actually written is returned in the output argument **resultlen**.

How to use Microsoft HPC 2008 Cluster to run MPI applications