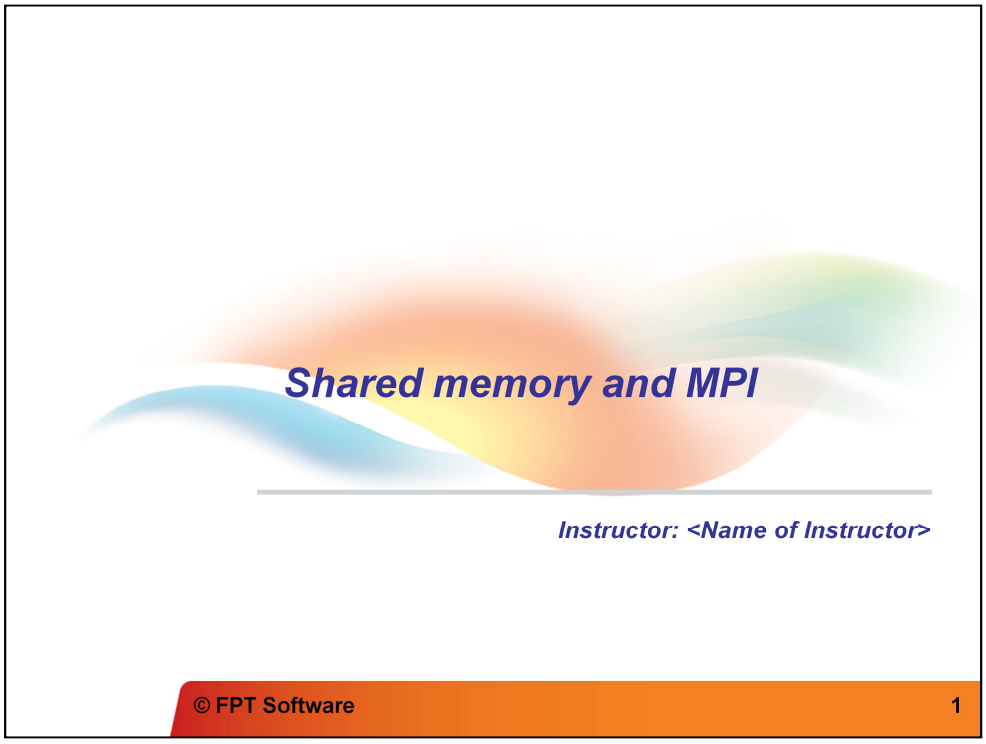


Latest updated by: HanhTT1

Agenda

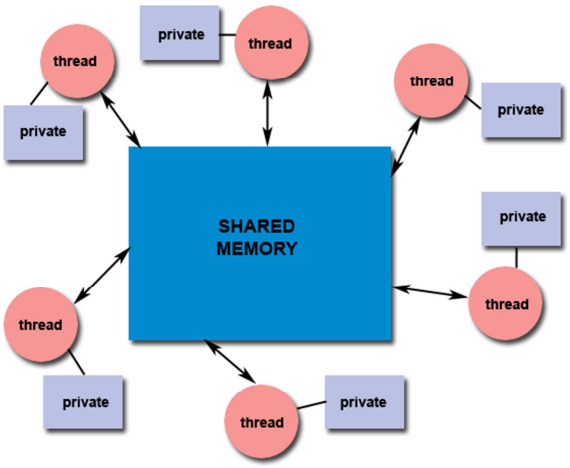
- Shared memory
- MPI

Shared memory

- Introduce about shared memory
- Managing memory in Windows
- Introduce about memory mapped file
- Memory mapped file operations
- Implement memory mapped files
- Example

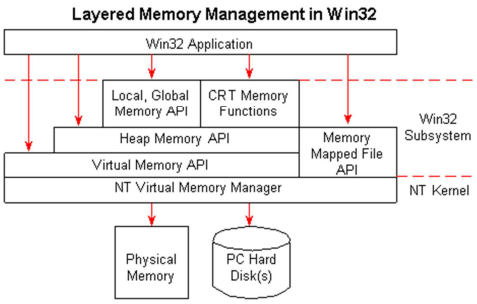
Introduce about shared memory

- Shared memory provides a way around this by letting two or more processes share



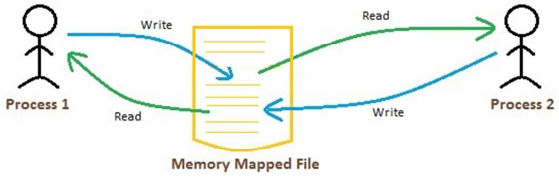
Managing memory in Windows

- Window offers three groups of functions for managing memory in applications
  - Memory mapped file functions
  - Heap memory functions
  - Virtual-memory functions



Introduce about memory mapped file

- Memory-mapped files (MMFs) offer a unique memory management feature that allows applications to access files on disk in the same way they access dynamic memory-through pointers
- Types of Memory Mapped Files
  - Persisted Files : these files have a physical file on disk
  - Non-persisted files : these files do not have a corresponding physical file on the disk
- Increased I/O performance since the contents of the file are loaded in memory.
- Memory mapped files using a small amoun
- If the user does not c a lot of memory and all the contents will h



Types of Memory Mapped Files

Memory mapped files have two variants:

Persisted Files- These files have a physical file on disk which they relate to. These types of memory-mapped files are used when working with extremely large files. A portion of these physical files are loaded in memory for accessing their contents.

Non-persisted files - These files do not have a corresponding physical file on the disk. When the process terminates, all content is lost. These types of files are used for inter-process communication also called IPC. In such cases, processes can map to the same memory mapped file by using a common name that is assigned by the process to create the file.

Benefits of Memory Mapped Files

One of the primary benefits of using memory-mapped files is increased I/O performance since the contents of the file are loaded in memory. Accessing RAM is faster than disk I/O operation and hence a performance boost is achieved when dealing with extremely large files.

Memory mapped files also offer lazy loading which equated to using a small amount of RAM for even a large file. This works as follows. Usually an application only has to show one page's worth of data. For such applications, there is no point loading all the contents of the file in memory. Having memory mapped files and their ability to create views allows us to reduce the memory footprint of the application.

Drawbacks of Memory Mapped Files

Since memory mapped files amount to loading data in memory, if the user does

Memory mapped file operations

Below common functions use for memory mapped file

- CreateFileMapping
  - Creates or opens a named or unnamed file mapping object for a specified file
- OpenFileMapping
  - Opens a named file mapping object
- MapViewOfFile
  - Maps a view of a file mapping into the address space of a calling process
- UnMapViewOfFile
  - Maps a view of a file mapping into the address space of a calling process

Memory mapped file operations

CreateFileMapping

```
HANDLE CreateFileMapping (
        HANDLE                  hFile,          //a handle to the file from which to create a file mapping object
        LPSECURITY_ATTRIBUTES   lpAttributes,   //a pointer to a SECURITY_ATTRIBUTES structure
        DWORD                   flProtect,      //specifies the page protection of the file mapping object
        DWORD                   dwMaximumSizeHigh,  //the maximum size of the file mapping object
        DWORD                   dwMaximumSizeLow,   //the maximum size of the file mapping object
        LPCTSTR                 lpName                  //the name of the file mapping object
);
```

- Return value is a handle to the newly created file mapping object when success or NULL when fails
- If hFile is INVALID_HANDLE_VALUE, must also specify a size for the file mapping object in the dwMaximumSizeHigh and dwMaximumSizeLow parameters
- If the object exists before the unction call, the function returns a handle to the existing object, and GetLastError returns ERROR_ALREADY_EXISTS

Memory mapped file operations

OpenFileMapping

```
HANDLE OpenFileMapping (
    DWORD     dwDesiredAccess,   //the access to the file mapping object
    BOOL      bInheritHandle,    //identify can inherit or not
    LPCTSTR   lpName             //the name of the file mapping object to be opened
);
```

- Return value is an open handle to the specified file mapping object or NULL when fails
- If bInheritHandle is TRUE, a process created by the CreateProcess function can inherit the handle
- bInheritHandle almost always is FALSE

Memory mapped file operations

MapViewOfFile

```
LPVOID MapViewOfFile (
HANDLE hFileMappingObject,      //a handle to a file mapping object
DWORD dwDesiredAccess,          //the type of access to a file mapping object
DWORD dwFileOffsetHigh,         //a high-order DWORD of the file offset
DWORD dwFileOffsetLow,          //A low-order DWORD of the file offset
SIZE_T  dwNumberOfBytesToMap //the number of bytes of a file mapping to
                                //   map to the view
);
```

- Return value is the starting address of the mapped view when success or NULL when fails
- dwFileOffsetHigh and dwFileOffsetLow almost always are NULL

Memory mapped file operations

UnMapViewOfFile

```
BOOL UnmapViewOfFile (
    LPCVOID lpBaseAddress          //pointer to the base address of the mapped view
);
```

- Return value is nonzero when success or 0 when fails
- Use CloseHandle with mapped file handle after unmapping mapped view is success
- To minimize the risk of data loss in the event of a power failure or a system crash, applications should explicitly flush modified pages using the **FlushViewOfFile** function

Implement memory mapped files

First process

•Create file mapping object with handle file is INVALID_HANDLE_VALUE and a name for it with function CreateFileMapping

•Create a view of the file in the process address space by function MapViewOfFile

•When process no longer needs access to the file to the file mapping object, call UnMapViewOfFile and CloseHandle

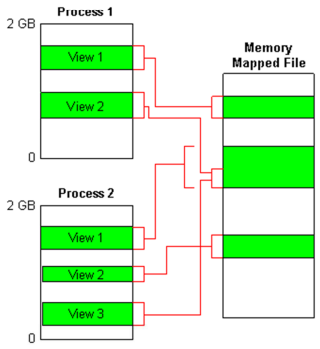
Other processes

•Access the data written to the shared memory by the first process by calling the OpenFileMapping

•Use the MapViewOfFile function to obtain a pointer to the file view

•Call function UnMapViewOfFile and CloseHandle for close handle file

When all handles are closed, the system can free the section of the paging file that the object uses



Example

- Initial Application

```
SHARE_DATA* pData = NULL;
int         nSize = sizeof(SHARE_DATA);

HANDLE handle = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0, nSize, SHARED_MEMORY_NAME);

if(handle != NULL)
{
    wcout<<_T("Create shared file mapping is success\n");

    pData = (SHARE_DATA*)MapViewOfFile(handle, FILE_MAP_ALL_ACCESS, NULL, NULL, nSize);

    if(pData != NULL)
    {
        wcout<<_T("Create map view file is success\n");
        wcscpy_s(pData->strMsg, MAX_PATH, _T("Hello world!!!"));
        wcout<<pData->strMsg<<endl;
    }
    else
    {
        wcout<<_T("Can't create map view file\n");
    }
}
else
{
        wcout<<_T("Can't create shared file mapping\n");
}
```

Example

- Reused Application

```
SHARE_DATA* pData = NULL;
int         nSize = sizeof(SHARE_DATA);

HANDLE handle = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, SHARED_MEMORY_NAME);

if(handle != NULL)
{
    wcout<<_T("Open file mapping is success\n");

    pData = (SHARE_DATA*)MapViewOfFile(handle, FILE_MAP_ALL_ACCESS, NULL, NULL, nSize);

    if(pData != NULL)
    {
        wcout<<_T("Create map view file is success\n");
        wcout<<pData->strMsg<<endl;
    }
    else
    {
        wcout<<_T("Can't create map view file\n");
    }
}
else
{
    wcout<<_T("Can't open shared file mapping\n");
}
```

MPI

- Introduce about MPI
- MPI operations
- Microsoft MPI
- Implement Microsoft MPI
- Example

Introduce about MPI

- MPI standard for a message passing library to be used for message passing parallel computing
- Developed by an ad-hoc open forum of vendors, users and researches
- MPI is used in
  - Parallel computers and clusters
  - Network of Workstations (NoW)
  - Mostly technical computing: datamining, portfolio modeling…
  - Basic programming model: communicating sequential processes
- Why use MPI?
  - Parallel computing tightly coupled
  - Distributed computing loosely coupled
  - Can trade off protection and O/S involvement for performance
  - Can provide additional functions

MPI operations

Below common functions use for MPI

- MPI_Init
  - starts MPI
- MPI_Finalize
  - exits MPI
- MPI_Send
  - Performs a standard mode send operation
- MPI_Recv
  - Performs a receive operation

Note

- Include header #include “mpi.h” provides basic MPI definitions and types
- MPI functions return error codes or MPI_SUCCESS

MPI operations

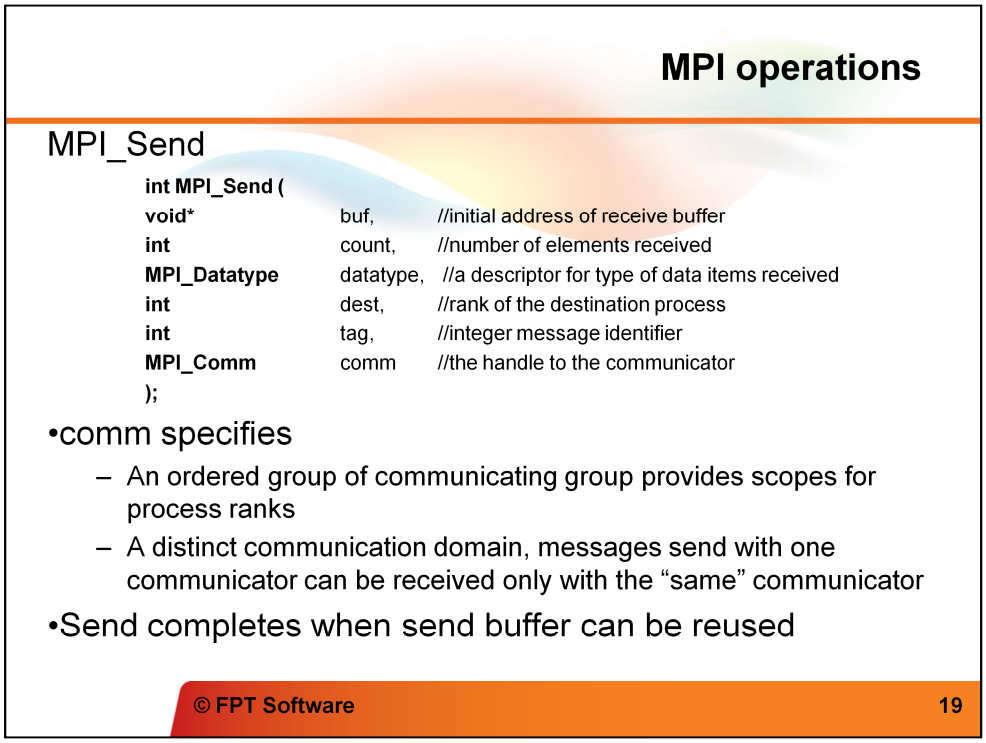
MPI_Init

```
int MPI_Init (
int*    argc, //pointer to the number of arguments
char*** argv //pointer to the argument vector
);
```

•The initialization routine MPI_INIT is the first MPI routine called
•MPI_INIT is called once

MPI_Finalize

```
int MPI_Finalize ();
```



Send completes when send buffer can be reused

- Can be before received started (if communication is buffered and message fits in buffer)
- May block until matching receive occurs (if message is not fully buffered)

MPI_Recv

```
int MPI_Recv (
void*        buf,                  //initial address of receive buffer
int                                 count,      //number of elements received
MPI_Datatype datatype,             //a descriptor for type of data items received
int                                 source,     //rank with communication group, can be
MPI_ANY_SOURCE
int          tag,                  //integer message identifier, can be MPI_ANY_TAG
MPI_Comm comm,                     //the handle to the communicator
MPI_Status* status   //a structure that provides information on completed communication
);
```

- Receive completes when data is available in receive buffer

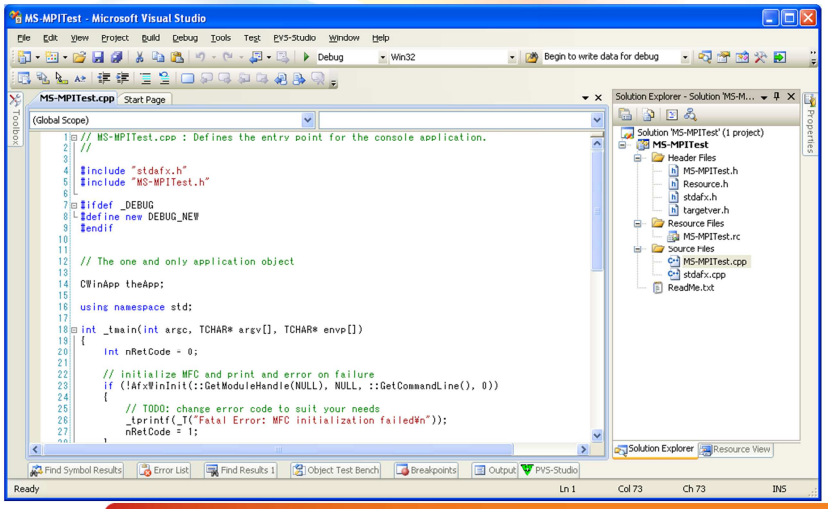
Microsoft MPI

- Microsoft MPI (MS-MPI) is a Microsoft implementation of the Message Passing Interface standard for developing and running parallel applications on the Windows platform
- MS-MPI offers several benefits
  - Ease of porting existing code that uses MPICH
  - Security based on Active Directory Domain Services
  - High performance on the Windows operating system
  - Binary compatibility across different types of interconnectivity options
- Download SDK
  - http://msdn.microsoft.com/en-us/library/cc853440(v=vs.85).aspx
- Install Microsoft HPC Pack SDK, when setup is finished it will contain two main folders
  - Lib : C:\Program Files\Microsoft HPC Pack 2008 SDK\Lib
  - Include : C:\Program Files\Microsoft HPC Pack 2008 SDK\Include

  with C:\Program Files\Microsoft HPC Pack 2008 SDK\ is install directory and Microsoft HPC Pack 2008 SDK is version of MS-MPI

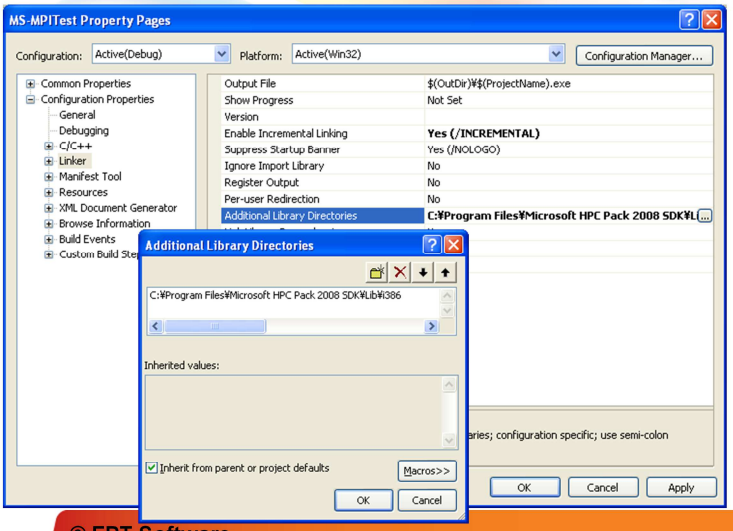
Implement Microsoft MPI

- Create new project C++ in Visual Studio



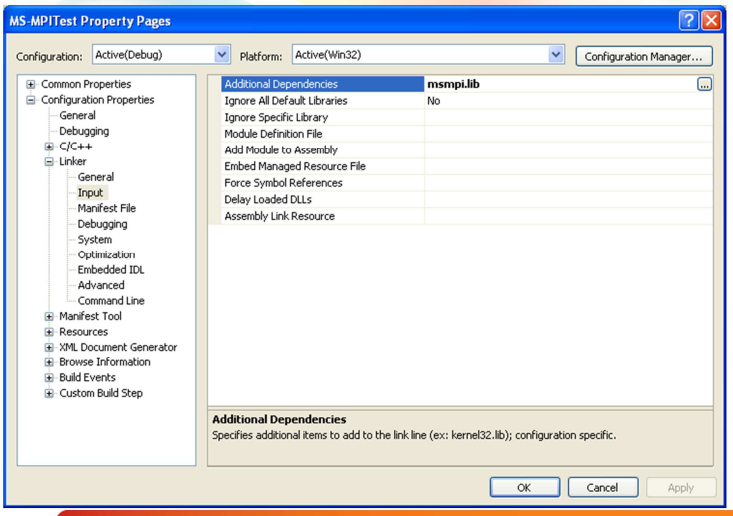
Implement Microsoft MPI

- Add Linker Additional Library Directories as following



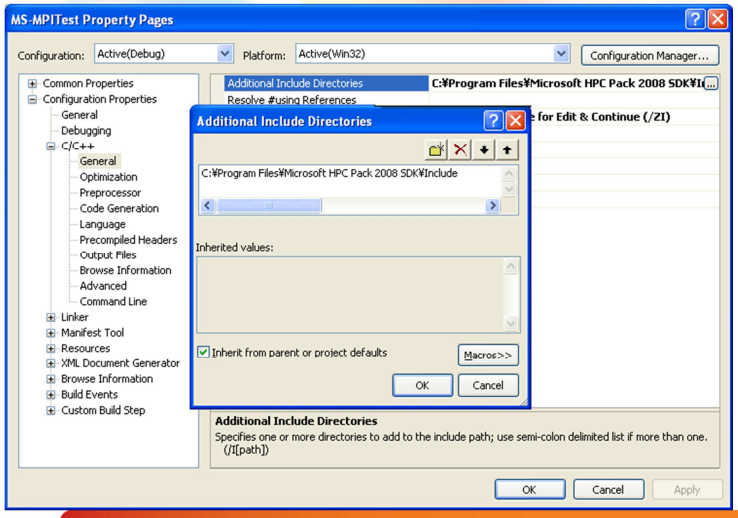
Implement Microsoft MPI

- Add Linker's Input Additional Dependencies. Type msmpi.lib to the list



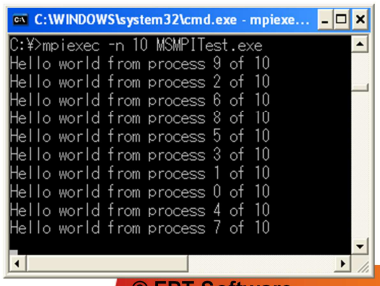
Implement Microsoft MPI

- Add the location of header file to C++ Compiler property



Example

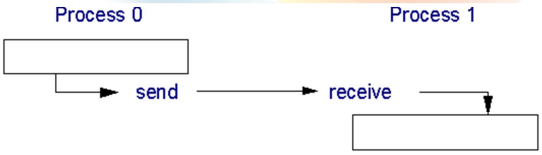
mpiexec –n 10 MyMPIProject.exe (run 10 apps)




```
int main(int argc, char* argv[])
{
    int nNode   = 0;
    int nTotal  = 0;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &nTotal);
    MPI_Comm_rank(MPI_COMM_WORLD, &nNode);

    cout<<"Hello world from process "<<nNode;
    cout<<" of "<<nTotal<<endl;

    MPI_Finalize();

    return 0;
}
```

Example

- Example send and receive message between 2 processes



Example

- Initialize MPI

```
const int   nTag        = 42;       /* Message tag                      */
int         nID         = 0;        /* Process ID                       */
int         nTasks      = 0;        /* Total current process            */
int         nSourceID   = 0;        /* Process ID of sended process     */
int         nDestID     = 0;        /* Process ID of received process   */
int         nErr        = 0;        /* Error                            */
int         msg[2];                 /* Message array                    */
MPI_Status  mpi_status;             /* MPI Status                       */

nErr = MPI_Init(&argc, &argv);      /* Initialize MPI */
if (nErr != MPI_SUCCESS)
{
    printf("MPI initialization failed!\n");
    return 1;
}

nErr = MPI_Comm_size(MPI_COMM_WORLD, &nTasks);  /* Get nr of tasks */
nErr = MPI_Comm_rank(MPI_COMM_WORLD, &nID);     /* 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 */
    return 0;
}
```

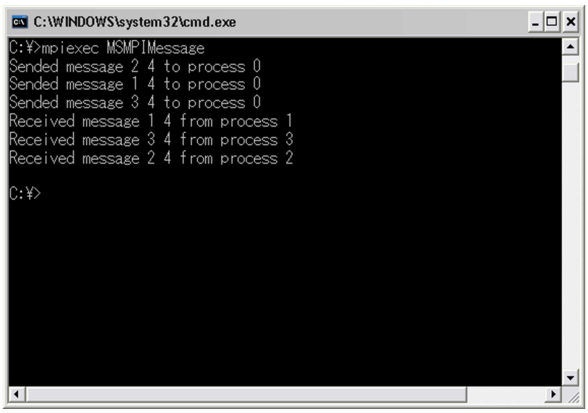
Example

- Send and receive message

```
if (nID == 0)
{
    /* Process 0 (the receiver) does this */
    for (int i = 1; i < nTasks; i++)
    {
        nErr = MPI_Recv(msg, 2, MPI_INT, MPI_ANY_SOURCE, nTag, MPI_COMM_WORLD, &mpi_status);

        /* Get id of sender */
        nSourceID = mpi_status.MPI_SOURCE;
        printf("Received message %d %d from process %d¥n", msg[0], msg[1], nSourceID);
    }
}
else
{
    /* Processes 1 to N-1 (the senders) do this */
    msg[0]  = nID;          /* Put own identifier in the message */
    msg[1]  = nTasks;       /* and total number of processes */
    nDestID = 0;            /* Destination address */

    printf("Sended message %d %d to process %d¥n", msg[0], msg[1], nDestID);
    nErr = MPI_Send(msg, 2, MPI_INT, nDestID, nTag, MPI_COMM_WORLD);
}

nErr = MPI_Finalize();      /* Terminate MPI */

return 0;
```

Example

- Run command line
  - mpiexec [Application_Dir]\[Application_Name]
- Result




Q&A