INDIANA UNIVERSITY

MPI.NET Tutorial in C#

Douglas Gregor and Benjamin Martin
Open Systems Laboratory, Indiana University

Published: September 2008

Contents

Introduction

This tutorial will help you install and use MPI.NET, a .NET library that enables the creation of high-performance parallel applications that can be deployed on multi-threaded workstations and Windows clusters. MPI.NET provides access to the Message Passing Interface (MPI) in C# and all of the other .NET languages. MPI is a standard for message-passing programs that is widely implemented and used for high-performance parallel programs that execute on clusters and supercomputers.

By the end of this tutorial, you should be able to:

- Install MPI.NET and its prerequisites.
- Write parallel MPI applications for deployment on Windows workstations and clusters using point-to-point and collective communication.
- Execute parallel MPI applications locally and on a cluster.

MPI Programming Model

The MPI programming model is, as its name implies, is based on message passing. In a message-passing system, different concurrently-executing processes communicate by sending messages from one to another over a network. Unlike multi-threading, where different threads share the same program state, each of the MPI processes has its own, local program state that cannot be observed or modified by any other process except in response to a message. Therefore, the MPI processes themselves can be as distributed as the network permits, with different processes running on different machines or even different architectures.

Most MPI programs are written with the Single Program, Multiple Data (SPMD) parallel model, where each of the processes is running the same program but working on a different part of the data. SPMD processes will typically perform a significant amount of computation on the data that is available locally (within that process's local memory), communicating with the other processes in the parallel program at the boundaries of the data. For example, consider a simple program that computes the sum of all of the elements in an array. The sequential program would loop through the array summing all of the values to produce a result. In a SPMD parallel program, the array would be broken up into several different pieces (one per process), and each process would sum the values in its local array (using the same code that the sequential program would have used). Then, the processes in the parallel program would communicate to combine their local sums into a global sum for the array.

MPI supports the SPMD model by allowing the user to easily launch the same program across many different machines (nodes) with a single command. Initially, each of the processes is identical, with one distinguishing characteristic: each process is assigned a rank, which uniquely identifies that process. The ranks of MPI processes are integer values from 0 to P-1, where P is the number of processes launched as part of the MPI program. MPI processes can query their rank, allowing different processes in the MPI program to have different behavior, and exchange messages with other processes in the same job via their ranks.

Installation

Prerequisites

To develop parallel programs using MPI.NET, you will need several other tools. Note that you do not need to have a Windows cluster or even a multi-core/multi-processor workstation to develop MPI programs: any desktop machine that can run Windows XP can be used to develop MPI programs with MPI.NET.

- Microsoft Visual Studio 2005 (or newer), including Microsoft Visual C#: We will be writing all of our examples in C#, although MPI.NET can be used from any .NET language.
- Microsoft's Message Passing Interface (MS-MPI): There are actually several different ways to get MS-MPI (you only need to do one of these):
  - Microsoft HPC SDK or Microsoft Compute Cluster Pack SDK: Includes MS-MPI and the various headers one needs to build MPI programs written in C or C++ (without MPI.NET).The HPC SDK is the newer version of MS-MPI (version 2), but the Compute Cluster Pack SDK (version 1 of MS-MPI) also works with MPI.NET
  - Microsoft HPC Server 2008 or Microsoft Compute Cluster Server 2003: Both versions of Microsoft's Windows version for clusters are compatible with MPI.NET. You will need to run one of these operating systems on your Windows cluster to deploy MPI programs across multiple computers. However, for development purposes it is generally best to use one of the SDKs mentioned above, which will work on normal Windows XP or Windows Vista workstations as well as the servers.
- Windows Installer: Most Windows users will already have this program, which is used to install programs on Microsoft Windows.

Install the MPI.NET SDK

To develop programs for MPI.NET, you will need the MPI.NET software development kit, which you can download from the MPI.NET download page. Execute the installer to install the MPI.NET Software Development Kit.

Running an MPI.NET Program

Once the MPI.NET SDK has been installed, it's time to run our first MPI program. Open up a Command Prompt and navigate to the location where you installed the MPI.NET SDK. You should type what is shown in red.

```
C:\>cd "C:\Program Files\MPI.NET"
```

Then, execute the program PingPong.exe:

```
C:\Program Files\MPI.NET>PingPong.exe
Rank 0 is alive and running on jeltz
```

Here, we have executed the MPI program PingPong with a single process, which will always be assigned rank 0. The process has displayed the name of the computer it is running on (our computer is named "jeltz") and returned. When you run this program, you might get a warning from Windows Firewall like the following, because PingPong is initiating network communications. Just select "Unblock" to let the program execute.

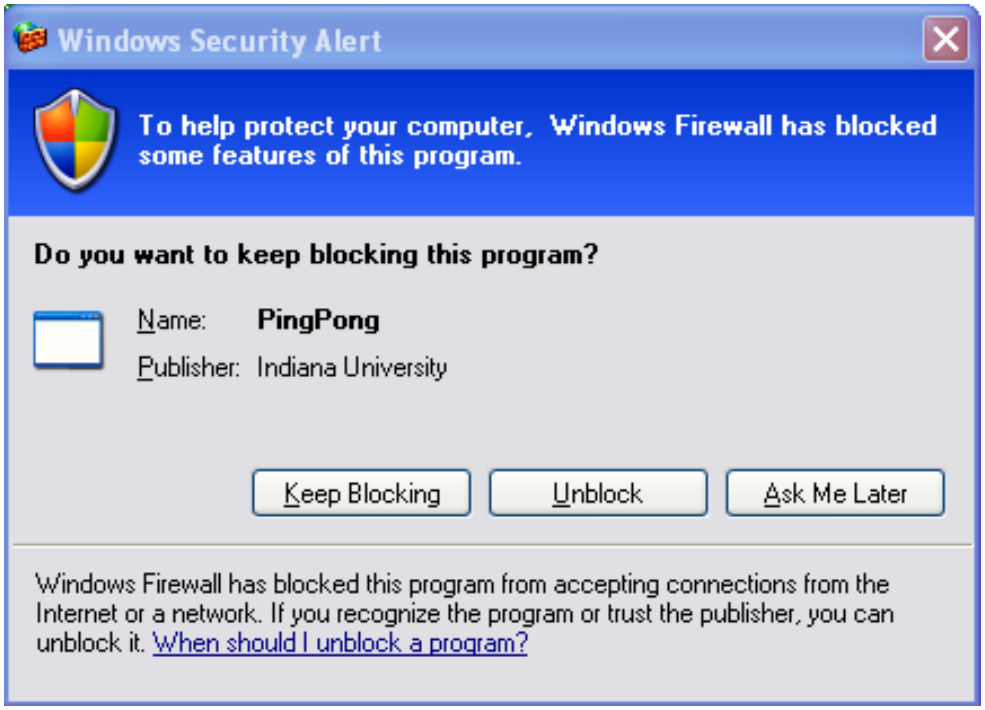


Figure 1: Windows Firewall Security Alert

To make PingPong more interesting, we will instruct MPI to execute 8 separate processes in the PingPong job, all coordinating via MPI. Each of the processes will execute the PingPong program, but because they have different MPI ranks (integers 0 through 7, inclusive), each will act slightly differently. To run MPI programs with multiple processes, we use the mpiexec

program provided by the Microsoft Compute Cluster Pack or its SDK, as follows (all on a single command line):

```
C:\Program Files\MPI.NET>"C:\Program Files\Microsoft Compute Cluster Pack\Bin\mpiexec.exe" -n 8 PingPong.exe
Rank 0 is alive and running on jeltz
Pinging process with rank 1... Pong!
  Rank 1 is alive and running on jeltz
Pinging process with rank 2... Pong!
  Rank 2 is alive and running on jeltz
Pinging process with rank 3... Pong!
  Rank 3 is alive and running on jeltz
Pinging process with rank 4... Pong!
  Rank 4 is alive and running on jeltz
Pinging process with rank 5... Pong!
  Rank 5 is alive and running on jeltz
Pinging process with rank 6... Pong!
  Rank 6 is alive and running on jeltz
Pinging process with rank 7... Pong!
  Rank 7 is alive and running on jeltz
```

That's it! The mpiexec program launched 8 separate processes that are working together as a single MPI program. The -n 8argument instructs mpiexec to start 8 processes that will all communicate via MPI (you can specify any number of processes here). In the PingPong program, the process with rank 0 will send a "ping" to each of the other processes, and report the name of the computer running that process back to the user. Since we haven't told mpiexec to run on different computers, all 8 processes are running on our workstation; however, the same program would work even if the 8 processes were running on different machines.

If PingPong ran correctly on your system, your installation of the MPI.NET SDK is complete. If you are going to be developing and running MPI programs, you will probably want to add the Compute Cluster Pack's Bin directory to your PATH environment variable, so that you can run mpiexec directly. From now on, we're going to assume that you have done so and will just use mpiexec in our examples without providing its path.

Installing MPI.NET on a Cluster

In order to use MPI.NET on a cluster, the MPI.NET Runtime must be installed on the cluster nodes. The MPI.NET Runtime can be downloaded from the MPI.NET download page.

In order to install the MPI.NET Runtime on the cluster nodes, you'll need to have administrative privileges.To run the installer across the cluster, you first need to put the installer somewhere accessible across the cluster by placing it on a drive or in a folder that is shared across the network. For now we'll assume the folder the installer is in is shared as\\Installers\. From a command line, run the following command:

```
clusrun msiexec.exe /quiet /i"\\Installers\MPI.NET Runtime.msi"
```

(Alternatively, instead of using clusrun, msiexec can be run from within Compute Cluster Administrator, using "Run Command…".)

Running an MPI.NET Program on a Cluster

The .NET security model by default requires programs to be on a local drive to be run. Therefore, to run MPI.NET programs on a cluster you must either copy the program to all nodes in the cluster, or increase the trust level of the program on all nodes using caspol. For now we'll just copy the program to all nodes.

Copy PingPong.exefrom C:\Program Files\MPI.NET\to some shared location, which we'll assume is called \\mpi.net\. Next, we need to have somewhere to put it; enter the following on the command line:

```
C:\>clusrun mkdir c:\PingPong\
```

Now we use the following command to copy it to the machines in the cluster:

```
C:\>clusrun copy \\mpi.net\PingPong.exe C:\PingPong\
```

Now we can run an MPI job to execute PingPong.exe. Enter the following on the command line:

```
C:\>job submit /stdout:\\mpi.net\out.txt /stderr:\\mpi.net\err.txt
/numprocessors:1mpiexec C:\PingPong\PingPong.exe
Job has been submitted. ID: 307.
```

The options /stdout and /stderr determine where the output and any error messages from the program should be directed (in this case to out.txt and err.txt, respectively). The /numprocessors option determines how many processors, up to the number available in the cluster, will be used to run the program. The rest of the line is the same as when we were running PingPong.exe locally.

The Compute Cluster Job Manager can be used to check on the progress of the job. When the job is finished, if there were no errors, out.txt should contain nearly the same output as before for PingPong.exe except the machine name will be different.

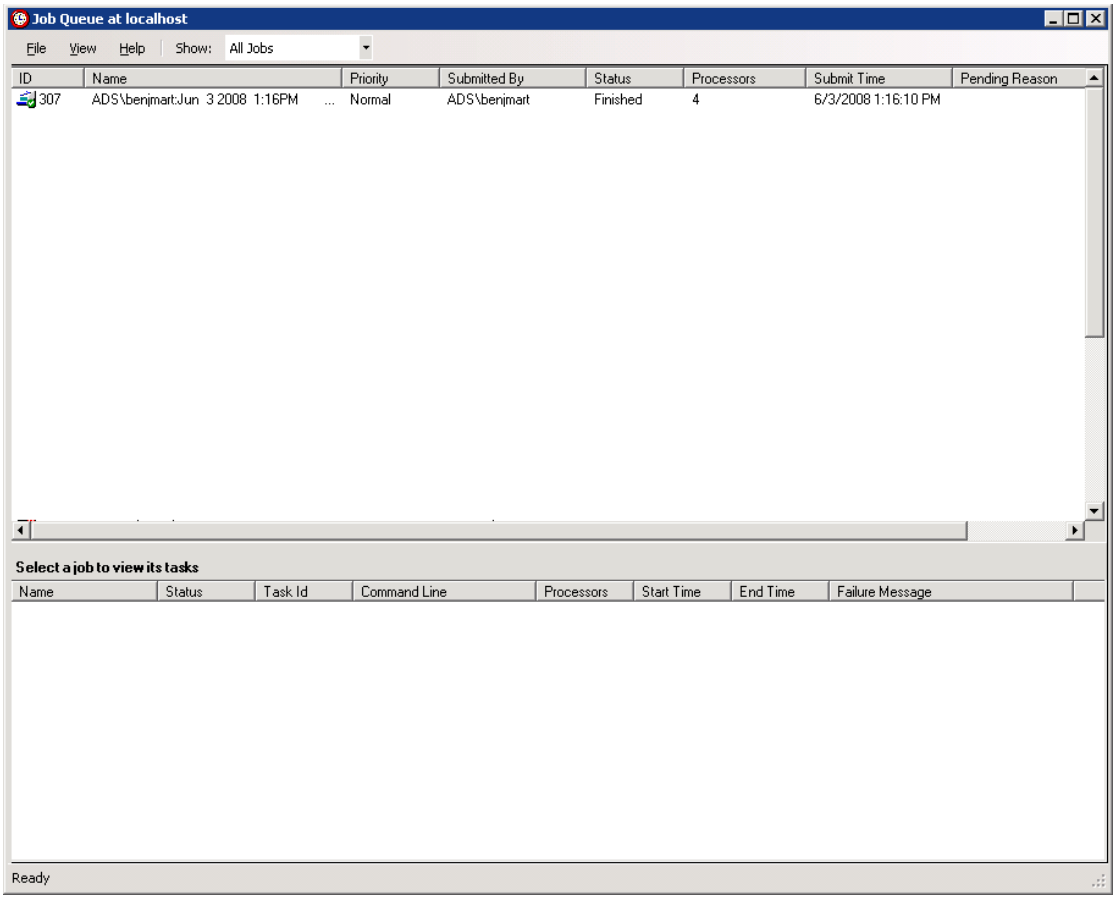


Figure 2: Compute Cluster Job Manager

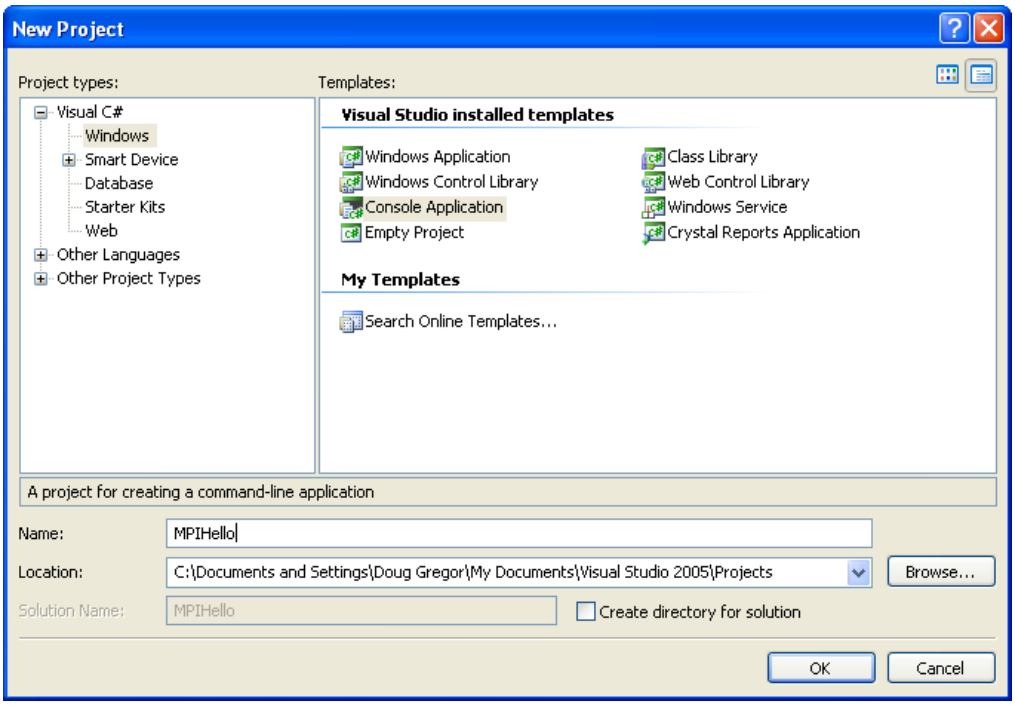
When finished, we will probably want to remove the local copies of the PingPong folder from all of the machines:

```
C:\>clusrun rmdir /s /q C:\PingPong\
```

Hello, World!

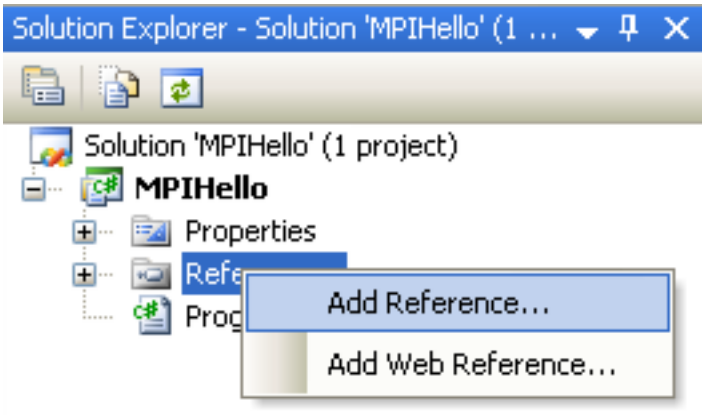
Create a New Project

To create an MPI "Hello, World!", we'll first create a new C# console application in Visual Studio. We chose the project name MPIHello for this new project.

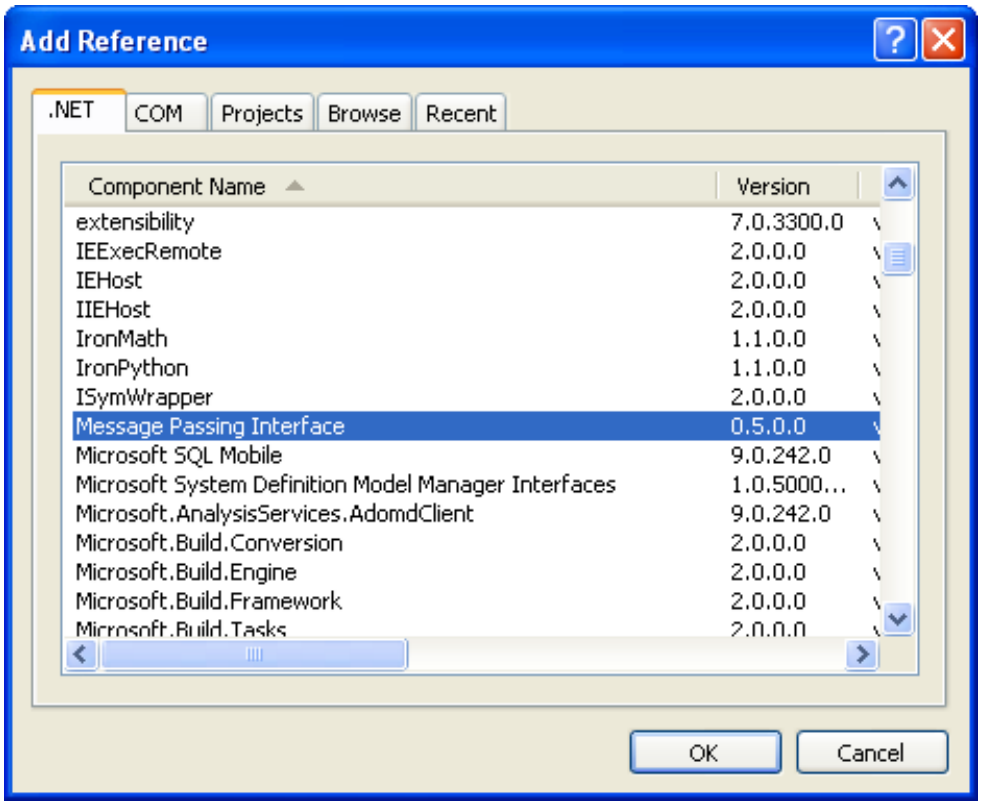


Reference the MPI.NET Assembly

Once you've created your project, you need to add a reference to the MPI.NET assembly in Visual Studio. This will allow your program to use MPI.NET's facilities, and will also give you on-line help for MPI.NET's classes and functions. In the Solution Explorer, right click on "References" and select "Add Reference...":



Next, scroll down to select the "Message Passing Interface" item from the list of components under the .NET tab, then click "OK" to add a reference to the MPI.NET assembly.



Writing Hello, World!

The first step in any MPI program is to initialize the MPI environment. All of the MPI processes will need to do this initialization before attempting to use MPI in any way. To initialize the MPI environment, we first bring in the MPI namespace with a using statement. Then, we create a new instance of MPI.Environment within our Main routine, passing the new object a reference to our command-line arguments:

```
using System;
using MPI;

class MPIHello
{
   static void Main(string[] args)
   {
      using (new MPI.Environment(ref args))
      {
         // MPI program goes here!
      }
```

```
    }
}
```

The entirety of an MPI program should be contained within the using statement, which guarantees that the MPI environment will be properly finalized (via MPI.Communicator.Dispose) before the program exits. All valid MPI programs must both initialize and finalize the MPI environment. We pass in a reference to our command-line arguments, args, because MPI implementations are permitted to use special command-line arguments to pass state information in to the MPI initialization routines (although few MPI implementations actually do this). In theory, MPI could remove some MPI-specific arguments from args, but in practice args will be untouched.

Now that we have the MPI environment initialized, we can write a simple program that prints out a string from each process. Inside the using statement, add the line:

```
Console.WriteLine("Hello, World! from rank " + Communicator.world.Rank
        + " (running on " + MPI.Environment.ProcessorName + ")");
```

Each MPI process will execute this code independently (and currently), and each will likely produce slightly different results. For example, MPI.Environment.ProcessorName returns the name of the computer on which a process is running, which could differ from one MPI process to the next (if we're running our program on a cluster). Similarly, we're printing out the rank of each process via Communicator.world.Rank. We'll talk about communicators a bit more later.

Running Hello, World!

To execute our "Hello, World!" program, navigate to the binary directory for your project (e.g., MPIHello\bin\Debug) and run some number of copies of the program with mpiexec:

```
C:\MPIHello\bin\Debug>mpiexec -n 8 MPIHello.exe
Hello, World! from rank 0 (running on jeltz)
Hello, World! from rank 6 (running on jeltz)
Hello, World! from rank 3 (running on jeltz)
Hello, World! from rank 7 (running on jeltz)
Hello, World! from rank 4 (running on jeltz)
Hello, World! from rank 1 (running on jeltz)
Hello, World! from rank 2 (running on jeltz)
Hello, World! from rank 5 (running on jeltz)
```

Notice that we have 8 different lines of output, one for each of the 8 MPI processes we started as part of our MPI program. Each will output it rank (from 0 to 7) and the name of the processor or machine it is running on. The output you receive from running this program will be slightly different from the output shown here, and will probably differ from one invocation to the next. Since the processes are running concurrently, we don't know in what order the processes will finish the call to `WriteLine` and write that output to the screen. To actually enforce some ordering, the processes would have to communicate.

MPI Communicators

In the "Hello, World!" example, we referenced the Communicator class in MPI.NET to determine the rank of each process. MPI communicators are the fundamental abstraction that permits communication among different MPI processes, and every non-trivial MPI program will make use of some communicators.

Each communicator representations a self-contained communication space for some set of MPI processes. Any of the processes in that communicator can exchange messages with any other process in that communicator, without fear of those messages colliding with any messages being transmitted on a different communicator. MPI programs often use several different communicators for different tasks: for example, the main MPI program may use one communicator for control messages that direct the program based on user input, while certain subgroups of the processes in that program use their own communicators to collaborate on subtasks in that program. Since each of the communicators is a completely distinct communication space, there is no need to worry about having the "control" messages from the user clash with the messages that the subgroups exchange while working on a task in the program.

There are two major properties of communicators used by essentially every MPI program: the rank of the process within the communicator, which identifies that process, and the size of the communicator, which provides the number of processes in the communicator.

Every MPI program begins with only two communicators defined, world and self. The world communicator (written as Communicator.world) is a communicator that contains all of the MPI processes that the MPI program started with. So, if the user started 8 MPI processes via mpiexec, as we did above, all 8 of those processes can communicate via the world communicator. In our "Hello, World!" program, we printed out the rank of each process within the world communicator. The self communicator is quite a bit more limited: each process has its own self communicator, which contains only its own process and nothing more. We will not refer to the self communicator again in this tutorial, because it is rarely used in MPI programs. From the initial two communicators, world and self, the usercan create their own communicators, either by “cloning” a communicator (which produces a communicator with the same processes, same ranks, but a separate communicator space) or by selecting subgroups of those processes.

Now that we've written "Hello, World!" and have introduced MPI communicators, we'll move on to the most important part of MPI: passing messages between the processes in an MPI program.

Point-to-Point Communication

Point-to-point communication is the most basic form of communication in MPI, allowing a program to send a message from one process to another over a given communicator. Each message has a source and target process (with processes identified by their ranks within the communicator), an integral “tag” that identifies the kind of message, and a payload containing arbitrary data. Tags will be discussed in more detail later.

There are two kinds of communication for sending and receiving messages via MPI.NET's point-to-point facilities: blocking and non-blocking. The blocking point-to-point operations will wait until a communication has completed in its local process before continuing. For example, a blocking Send operation will not return until the message has entered into MPI's internal buffers to be transmitted, while a blocking Receive operation will wait until a message has been received and completely decoded before returning. MPI.NET's non-blocking point-to-point operations, on the other hand, will initiate a communication without waiting for that communication to be completed. The call to the non-blocking operation will return as soon as the operation is begun, not when it completes.A Request object, which can be used to query, complete, or cancel the communication, will be returned. For our initial examples, we will use blocking communication.

Ring Around the Network

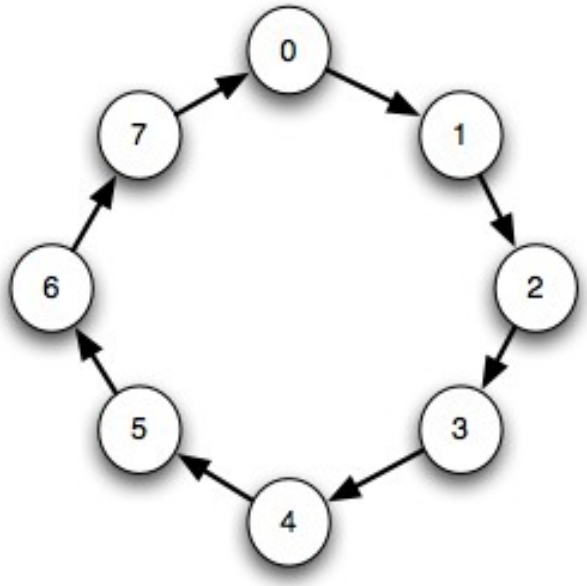


Figure 3: A ring network. Circles represent processes, and arrows represent messages.

For our first example of point-to-point communication, we will write a program that sends a message around a ring. The message will start at one of the processes--we'll pick the rank 0 process--then proceed from one process to another, eventually ending up back at the process that originally sent the data. The figure above illustrates the communication pattern, where a process is a circle and the arrows indicate the transmission of a message.

To implement our ring communication application, we start with the typical skeleton of an MPI program.Next we give ourselves an easy way to access the world communicator (via the variable comm). Then, since we have decided that process 0 will initiate the message, we give rank 0 a different code path from the other processes in the MPI program.

```
using System;
using MPI;

class Ring
{
   static void Main(string[] args)
   {
      using (new MPI.Environment(ref args))
      {
         Intraommunicator comm = Communicator.world;
         if (comm.Rank == 0)
         {
            // program for rank 0
         }
         else // not rank 0
         {
            // program for all other ranks
         }
      }
   }
}
```

This pattern of giving one of the processes (which is often called the "root", and is typically rank 0) a slightly different code path than all of the other processes is relatively common in MPI programs, which often need to perform some coordination or interaction with the user.

Rank 0 will be responsible for initiating the communication, by sending a message to rank 1. The code below initiates a (blocking) send of a piece of data. The three parameters to the Send routine are, in order:

- The data to be transmitted with the message. In this case, we're sending the string "Rosie".
- The rank of the destination process within the communicator. In this case, we're sending the message to rank 1. (We are therefore assuming that this program is going to run with more than one process!)
- The tag of the message, which will be used by the receiver to distinguish this message from other kinds of messages. We'll just use tag 0, since there is only one kind of message in our program.

```
if (comm.Rank == 0)
{
   // program for rank 0
   comm.Send("Rosie", 1, 0);

   // receive the final message
}
```

Now that we have initiated the message, we need to write code for each of the other processes. These processes will wait until they receive a message from their predecessor, print the message, and then send a message on to their successor.

```
else // not rank 0
{
   // program for all other ranks
   stringmsg = comm.Receive<string>(comm.Rank - 1, 0);

   Console.WriteLine("Rank " + comm.Rank + " received message \"" + msg + "\".");

   comm.Send(msg + ", " + comm.Rank, (comm.Rank + 1) % comm.Size, 0);
}
```

The Receive call in this example states that we will be receiving a string from the process with rank comm.Rank - 1 (our predecessor in the ring) and tag 0. This receive will match any message sent from that rank with tag zero; if that message does not contain a string, the program will fail. However, since the only Send operations in our program send strings with tag 0, we will not have a problem. Once a process has received a string from its predecessor, it will print that to the console and send another message on to its successor in the ring. This Send operation is much like rank 0's Send operation: most importantly, it sends a string with tag 0. Note that each process will add its

own rank to the message string, so that we get an idea of the path that the message took.

Finally, we return to the special-case code for rank 0. When the last process in the ring finally sends its result back to rank 0, we will need to receive that result. The Receive for rank 0 is similar to the Receive for all of the other processes, although here we use the special value Communicator.anySource for the "source" process of the receive. anySource allows the Receive operation to match a message with the appropriate tag, regardless of which rank sent the message. The corresponding value for the tag argument, Communicator.anyTag, allows a Receive to match a message with any tag.

```
if (comm.Rank == 0)
{
   // program for rank 0
   comm.Send("Rosie", 1, 0);

   // receive the final message
   stringmsg = comm.Receive<string>(Communicator.anySource, 0);

   Console.WriteLine("Rank " + comm.Rank + " received message \"" + msg + "\".");
}
```

We can now go ahead and compile this program, then run it with 8 processes to mimic the communication ring in the figure at the beginning of this section:

```
C:\Ring\bin\Debug>mpiexec -n 8 Ring.exe
Rank 1 received message "Rosie".
Rank 2 received message "Rosie, 1".
Rank 3 received message "Rosie, 1, 2".
Rank 4 received message "Rosie, 1, 2, 3".
Rank 5 received message "Rosie, 1, 2, 3, 4".
Rank 6 received message "Rosie, 1, 2, 3, 4, 5".
Rank 7 received message "Rosie, 1, 2, 3, 4, 5, 6".
Rank 0 received message "Rosie, 1, 2, 3, 4, 5, 6, 7".
```

In theory, even though the processes are each printing their respective messages in order, it is possible that the lines in the output could be printed in a different order (or even produce some unreadable interleaving of characters), because each of the MPI processes has its own "console", all of which are forwarded back to your command prompt. For simple MPI programs, however, writing to the console often works well enough.

At this point, we have completed our "ring" example, which passes a message around a ring of two or more processes and print the results. Now, we'll take a quick look at what kind of data can be transmitted via MPI.

Data Types and Serialization

MPI.NET can transmit values of essentially any data type via its point-to-point communication operations. The way in which MPI.NET transmits values differs from one kind of data type to another. Therefore, it is extremely important that the sender of a message and the receiver of a message agree on the exact type of the message. For example, sending a string "17" and trying to receive it as an integer 17 will cause your program to fail. It is often best to use different tags to send different kinds of data, so that you never try to receive data of the wrong type.

There are three kinds of types that can be transmitted via MPI.NET:

Primitive types. These are the basic types in C#, such as integers and floating-point numbers.

Public Structures. These are C# structures with public visibility. For example, the following Point structure:

```
public struct Point
{
   public float x;
   public float y;
}
```

Serializable Classes. A class can be made serializable by attaching the Serializableattribute, as shown below; for more information, see Object Serialization using C#.

```
[Serializable]
public class Employee
{
   // ...
}
```

As mentioned before, MPI.NET transmits different data types in different ways. While most of the details of value transmission are irrelevant to MPI users, there is a significant distinction between the way that .NET value types are transmitted and the way that reference types are transmitted. The differences between value types and reference types are discussed in some detail in .NET: Type Fundamentals. For MPI.NET, value types, which include primitive types and structures, are always transmitted in a single message, and provide the best performance for message-passing applications.

Reference types, on the other hand, always need to be serialized (because they refer to objects on the heap) and (typically) are split into several messages for transmission. Both of these operations make the transmission of reference types significantly slower than value types. However, reference types are often necessary for complicated data structures, and provide one other benefit: unlike with value types, which require the data types at the sender and receiver to match exactly, one can send an object for a derived class and receive it via its base class, simplifying some programming tasks.

MPI.NET's point-to-point operations also provide support for arrays. As with objects, arrays are transmitted in different ways depending on whether the element type of the array is a value type or a reference type. In both cases, however, when you are receiving an array you must provide an array with at least as many elements as the sender has sent. Note that we provide the array to receive into as our last argument to Receive, using the ref keyword to denote that the routine will modify the array directly (rather than allocating a new array). For example:

```
if (comm.Rank == 0)
{
   int[] values = new int [5];
   comm.Send(values, 1, 0);
}
else if (comm.Rank == 1)
{
   int[] values = new int [10];
   comm.Receive(0, 0, ref values); // okay: array of 10 integers has enough space to
                                   // receive 5 integers
}
```

MPI.NET can transmit most kinds of data types used in C# and .NET programs. The most important rule with sending and receiving messages, however, is that the data types provided by the sender and receiver must match directly (for value types) or have a derived-base relationship (for reference types).

Collective Communication

Collective communication provides a more structured alternative to point-to-point communication. With collective communication, all of the processes within a communicator collaborate on a single communication operation that fits one of several common communication patterns used in message-passing applications. Collective operations include simple “barriers”; one-to-all, all-to-one, and all-to-all communications; and parallel reduction operations that combine the values provided by each of the processes in the communication.

Although it is possible to express parallel programs entirely through point-to-point operations (some even call send and receive the "assembly language" of distributed-memory parallel programming), collectives provide several advantages for writing parallel programs. For these reasons, it is generally preferred to use collectives whenever possible, falling back to point-to-point operations when no suitable collective exists.

- Code Readability/Maintainability

  It is often easier to write and reason about programs that use collective communication than the equivalent program using point-to-point communication. Collectives express the intent of a communication better (e.g., a Scatter operation is clearly distributing data from one process to all of the other processes), and there are often far fewer collective operations needed to accomplish a task than point-to-point messages (e.g., a single all-to-all operation instead of N^2 point-to-point operations), making it easier to debug programs using collectives.

- Performance

  MPI implementations typically contain optimized algorithms for collective operations that take advantage of knowledge of the network topology and hardware, even taking advantage of hardware-based implementations of some collective operations. These optimizations are hard to implement directly over point-to-point, without the knowledge already available in the MPI implementation itself. Therefore, using collective operations can help improve the performance of parallel programs and make that performance more portable to other clusters with different configurations.

Barrier: Marching Computations

In an MPI program, the various processes perform their local computations without regard to the behavior of the other processes in the program, except when the processes are waiting for some inter-process communication to

complete. In many parallel programs, all of the processes work more or less independently, but we still want to make sure that all of the processes are on the same step at the same time. The Barrier collective operation is used for precisely this operation. When a process enters the barrier, it does not exit the barrier until all processes have entered the barrier. Place barriers before or after a step of the computation that all processes need to perform at the same time.

In the example program below, each of the iterations of the loop is completely synchronized, so that every process is on the same iteration at the same time.

```
using System;
using MPI;

class Barrier
{
   static void Main(string[] args)
   {
      using (new MPI.Environment(ref args))
      {
         Intracommunicator comm = Communicator.world;
         for (inti = 1; i<= 5; ++i)
         {
            comm.Barrier();
            if (comm.Rank == 0)
               Console.WriteLine("Everyone is on step " + i + ".");
         }
      }
   }
}
```

Executing this program with any number of processes will produce the following output (here, we use 8 processes).

```
C:\Barrier\bin\Debug>mpiexec -n 8 Barrier.exe
Everyone is on step 1.
Everyone is on step 2.
Everyone is on step 3.
Everyone is on step 4.
Everyone is on step 5.
```

All-to-one: Gathering Data

The MPI Gather operation collects data provided by all of the processes in a communicator on a single process, called the root process. Gather is typically used to bring summary data from a computation back to the process responsible for communicating that information to the user. For example, in the following program, we gather the names of the processors (or hosts) on which each process is executing, then sort and display that information at the root.

```
using System;
using MPI;

class Hostnames
{
   static void Main(string[] args)
   {
      using (new MPI.Environment(ref args))
      {
         Intracommunicator comm = Communicator.world;
         string[] hostnames = comm.Gather(MPI.Environment.ProcessorName, 0);
         if (comm.Rank == 0)
         {
            Array.Sort(hostnames);
            foreach(string host in hostnames)
               Console.WriteLine(host);
         }
      }
   }
}
```

In the call to Gather, each process provides a value (in this case, the string produced by reading the ProcessorName property) to the Gather operation, along with the rank of the "root" node (here, process zero). The Gather operation will return an array of values to the root node, where the *i*th value in the array corresponds to the value provided by the process with rank *i*. All other processes receive a null array.

To gather all of the data from all of the nodes, use the Allgather collective. Allgather is similar to Gather, with two differences: first, there is no parameter identifying the "root" process, and second, all processes receive the same array containing the contributions from every process. An Allgather is, therefore, the same as a Gather followed by a Broadcast, described below.

One-to-all: Spreading the Message

While the Gather and Allgather collectives bring together the data from all of the processes, the Broadcast and Scatter collectives distribute data from one process to all of the processes.

The Broadcast operation takes a value from one process and broadcasts it to every other process. For example, consider a system that takes user input from a single process (rank 0) and distributes that command to all of the processes so that they all execute the command concurrently (and coordinate to complete the command). Such a system could be implemented with MPI as follows, using Broadcast to distribute the command:

```
using System;
using MPI;

class CommandServer
{
   static void Main(string[] args)
   {
      using (new MPI.Environment(ref args))
      {
         Intracommunicator comm = Communicator.world;
         string command = null;
         do
         {
            if (comm.Rank == 0)
               command = GetInputFromUser();

            // distribute the command
            comm.Broadcast(ref command, 0);

            // each process handles the command
         } while (command != "quit");
      }
   }
}
```

The Broadcast operation requires only two arguments; the second, familiar argument is the rank of the root process, which will supply the value. The first argument contains the value to send (at the root) or the place in which the received value will be stored (for every process). The pattern used in this example is quite common for Broadcast: all processes define the same variable, but only the root process gives it a meaningful value. Then the

processes coordinate to broadcast the root's value to every process, and all processes follow the same code path to handle the data.

The Scatter collective, like Broadcast, broadcasts values from a root process to every other process. Scatter, however, will broadcast different values to each of the processes, allowing the root to hand out different tasks to each of the other processes. The root process provides an array of values, in which the *i*th value will be sent to the process with rank *i*.Scatter returns the data received by each process.

All-to-all: Something for Everyone

The Alltoall collective transmits data from every process to every other process. Each process will provide an array whose *i*th value will be sent to the process with rank *i*. Each process will then receive in return a different array, whose *j*th value will be the value received from the process with rank *j*. In the example below, we generate unique strings to be sent from each process to every other process.

```
string[] data = new string[comm.Size];
for (intdest = 0; dest<comm.Size; ++dest)
   data[dest] = "From " + comm.Rank + " to " + dest;

string[] results = comm.Alltoall(data[]);
```

When executed with 8 processes, rank 1 will receive an array containing the following strings (in order):

```
From 0 to 1.
From 1 to 1.
From 2 to 1.
From 3 to 1.
From 4 to 1.
From 5 to 1.
From 6 to 1.
From 7 to 1.
```

Combining Results with Parallel Reduction

MPI contains several parallel reduction operations that combine the values provided by each of the processes into a single value that somehow sums up the results. The way in which results are combined is determined by the user, allowing many different kinds of computations to be expressed. For example, one can compute the sum or product of values produced by the processes, find the minimum or maximum of those values, or concatenate the results computed by each process.

The most basic reduction operation is the Reduce collective, which combines the values provided by each of the processes and returns the result at the designated root process. If the process with rank *i* contributes the value v_i, the result of the reduction for *n* processes is $v_1 + v_2 + ... + v_n$, where + can be any associative operation.

To illustrate the use of Reduce, we're going to use MPI to compute an approximation of pi. The algorithm is relatively simple: inscribe a unit circle within a unit square, and then randomly throw darts within the unit square. The ratio of the number of darts that land within the circle to the number of darts that land within the square is the same as the ration of the area of the circle to the area of the square, and therefore can be used to compute pi. Using this principle, the following sequential program computes an approximation of pi:

```
using System;

class SequentialPi
{
   static void Main(string[] args)
   {
      int dartsPerProcessor = 10000;
      Random random = new Random();
      int dartsInCircle = 0;
      for (inti = 0; i<dartsPerProcessor; ++i)
      {
         double x = (random.NextDouble() - 0.5) * 2;
         double y = (random.NextDouble() - 0.5) * 2;
         if (x * x + y * y <= 1.0)
            ++dartsInCircle;
      }

      Console.WriteLine("Pi is approximately {0:F15}.",
                  4*(double)totalDartsInCircle/(double)dartsPerProcessor);
   }
}
```

When running this program, the more darts you throw, the better the approximation to pi. To parallelize this program, we'll use MPI to run several processes, each of which will throw darts independently. Once all of the processes have finished, we'll sum up the results (the total number of darts that landed inside the circle on all processes) to compute pi. The complete code for the parallel calculation of pi follows, but the most important line uses

Reduce to sum the total number of darts that landed in the circle across all of the processes:

```
int totalDartsInCircle = comm.Reduce(dartsInCircle, Operation<int>.Add, 0);
```

The three arguments to Reduce are the number of darts that landed in the circle locally, a delegate Operation<int>.Add that sums integers, and the rank of the root process (here, 0). Any other .NET delegate would also work, e.g.,

```
public static intAddInts(int x, int y) { return x + y; }
// ...
inttotalDartsInCircle = comm.Reduce(dartsInCircle, AddInts, 0);
```

However, using the MPI.NET Operation class permits better optimizations within MPI.NET. Without further delay, here is the complete MPI program for computing an approximation to pi in parallel:

```
using System;
using MPI;

class Pi
{
   static void Main(string[] args)
   {
      using (new MPI.Environment(ref args))
      {
         Intracommunicator comm = Communicator.world;
         int dartsPerProcessor = 10000;
         Random random = new Random(5 * world.Rank);
         int dartsInCircle = 0;
         for (inti = 0; i<dartsPerProcessor; ++i)
         {
            double x = (random.NextDouble() - 0.5) * 2;
            double y = (random.NextDouble() - 0.5) * 2;
            if (x * x + y * y <= 1.0)
               ++dartsInCircle;
         }

         int totalDartsInCircle = comm.Reduce(dartsInCircle, Operation<int>.Add, 0);
         if (comm.Rank == 0)
            Console.WriteLine("Pi is approximately {0:F15}.",
                  4*(double)totalDartsInCircle/
(world.Size*(double)dartsPerProcessor));
      }
   }
}
```

More Information and Downloads

MPI.NET homepage

http://www.osl.iu.edu/research/mpi.net/

MPI.NET reference documentation
http://www.osl.iu.edu/research/mpi.net/documentation/reference/current/Index.html

MPI.NET mailing list

http://www.osl.iu.edu/mailman/listinfo.cgi/mpi.net