Introduction to MPI

Norman Matloff
Department of Computer Science
University of California at Davis

May 10, 2006

Contents

1 Overview

1.1 History

Though (small) shared-memory machines have come down radically in price, to the point at which a dual-core PC is affordable in the home, historically shared-memory machines were available only to the "very rich"—large banks, national research labs and so on.

The first "affordable" message-machine type was the Hypercube, developed by a physics professor at Cal Tech. It consisted of a number of **processing elements** (PEs) connected by fast serial I/O cards. This was in the range of university departmental research labs. It was later commercialized by Intel and NCube.

Later, the notion of **networks of workstations** (NOWs) became popular. Here the PEs were entirely independent PCs, connected via a standard network. This was refined a bit, by the use of more suitable network hardware and protocols, with the new term being **clusters**.

All of this necessitated the development of standardized software tools based on a message-passing paradigm. The first popular such tool was Parallel Virtual Machine (PVM). It still has its adherents today, but has largely been supplanted by the Message Passing Interface (MPI).

MPI itself later became MPI 2. Our document here is intended mainly for the original.

1.2 Structure and Execution

MPI is merely a set of Application Programmer Interfaces (APIs). It has many implementations.

Suppose we have written an MPI program **x**, and will run it on four machines in an Ethernet-based NOW. Each machine will be running its own copy of **x**. Official MPI terminology refers to this as four **processes**, but we will use the term **nodes**, i.e. **x** is running on four nodes.

Though the nodes are all running the same program, they will likely be working on different parts of the program's data. This is called the Single Program Multiple Data (SPMD) model. It is typical, but there could be different programs running on different nodes. Most of the APIs involve a node sending information to, or receiving information from, other nodes.

1.3 Implementations

In principle, an MPI implementation could be made quite generic, applicable to virtually any platform, simply by using network (or OS) sockets for internode communication. But for performance reasons, most implementations are tailored to a particular platform.

Two of the most popular implementations of MPI are MPICH and LAM. MPICH runs both on networks and on several other platforms, including selected shared-memory machines. LAM runs on networks. Introductions to MPICH and LAM can be found, for example, at `http://heather.cs.ucdavis.edu/~matloff/MPI/NotesMPICH.NM.html` and `http://heather.cs.ucdavis.edu/~matloff/`

`MPI/NotesLAM.NM.html`, respectively.

There is considerable evolution in these tools. MPICH became MPICH 2, while LAM became OpenMPI.

1.4 Performance Issues

Mere usage of a parallel language on a parallel platform does not guarantee a performance improvement over a serial version of your program. The central issue here is the overhead involved in internode communication.

As of 2006, the **latency** of Myrinet, one of the fastest cluster networks commercially available, is about 2 microseconds. In other words, if one node sends a message to another, it will take about 2 microseconds before the first bit reaches the destination. Comparing that to the nanosecond time scale of CPU speeds, one can see that the communications overhead can destroy a program's performance. And Ethernet, is quite a bit slower than Myrinet.

Note carefully that latency is a major problem even if the **bandwidth**—the number of bits per second which are sent—is high. For this reason, it is quite possible that your parallel program may actually run more slowly than its serial version.

Of course, if your platform is a shared-memory multiprocessor (especially a multicore one, where communication between cores is particularly fast), you must make sure that your application is sufficiently **coarse-grained** that latency is not an issue. What this means is that your application can be broken down into large subproblems that rarely require communication with other nodes, relative to the amount of computation done between communications.

2 Running Example

2.1 The Algorithm

The code implements the Dijkstra algorithm for finding the shortest paths in an undirected graph. Pseudocode for the algorithm is

```
1  Done = {0}
2  NonDone = {1,2,...,N-1}
3  for J = 1 to N-1 Dist[J] = infinity‘
4  Dist[0] = 0
5  for Step = 1 to N-1
6     find J such that Dist[J] is min among all J in NonDone
7     transfer J from NonDone to Done
8     NewDone = J
9     for K = 1 to N-1
10       if K is in NonDone
11          Dist[K] = min(Dist[K],Dist[NewDone]+G[NewDone,K])
```

At each iteration, the algorithm finds the closest vertex J to 0 among all those not yet processed, and then updates the list of minimum distances to each vertex from 0 by considering paths that go through J. Two obvious potential candidate part of the algorithm for parallelization are the “find J” and “for K” lines, and the above OpenMP code takes this approach.

2.2 The Code

```
1  // Dijkstra.c
2
3  // MPI example program:  Dijkstra shortest-path finder in a
4  // bidirectional graph; finds the shortest path from vertex 0 to all
5  // others
6
7  // command line arguments:  nv print dbg
8
9  // where:  nv is the size of the graph; print is 1 if graph and min
10 // distances are to be printed out, 0 otherwise; and dbg is 1 or 0, 1
11 // for debug
12
13 // node 0 will both participate in the computation and serve as a
14 // "manager"
15
16 #include <stdio.h>
17 #include <mpi.h>
18
19 #define MYMIN_MSG 0
20 #define OVRLMIN_MSG 1
21 #define COLLECT_MSG 2
22
23 // global variables (but of course not shared across nodes)
24
25 int nv,  // number of vertices
26     *notdone, // vertices not checked yet
27     nnodes,  // number of MPI nodes in the computation
28     chunk,  // number of vertices handled by each node
29     startv,endv,  // start, end vertices for this node
30     me,  // my node number
31     dbg;
32 unsigned largeint,  // max possible unsigned int
33          mymin[2],  // mymin[0] is min for my chunk,
34                     // mymin[1] is vertex which achieves that min
35          othermin[2],  // othermin[0] is min over the other chunks
36                        // (used by node 0 only)
37                        // othermin[1] is vertex which achieves that min
38          overallmin[2],  // overallmin[0] is current min over all nodes,
39                          // overallmin[1] is vertex which achieves that min
40          *ohd,  // 1-hop distances between vertices; "ohd[i][j]" is
41                 // ohd[i*nv+j]
42          *mind;  // min distances found so far
43
44 double T1,T2;  // start and finish times
45
46 void init(int ac, char **av)
```

```
47 {  int i,j,tmp; unsigned u;
48    nv = atoi(av[1]);
49    dbg = atoi(av[3]);
50    MPI_Init(&ac,&av);
51    MPI_Comm_size(MPI_COMM_WORLD,&nnodes);
52    MPI_Comm_rank(MPI_COMM_WORLD,&me);
53    chunk = nv/nnodes;
54    startv = me * chunk;
55    endv = startv + chunk - 1;
56    u = -1;
57    largeint = u >> 1;
58    ohd = malloc(nv*nv*sizeof(int));
59    mind = malloc(nv*sizeof(int));
60    notdone = malloc(nv*sizeof(int));
61    // random graph
62    // note that this will be generated at all nodes; could generate just
63    // at node 0 and then send to others, but faster this way
64    for (i = 0; i < nv; i++)
65       for (j = i; j < nv; j++)  {
66          if (j == i) ohd[i*nv+i] = 0;
67          else  {
68             ohd[nv*i+j] = rand() % 20;
69             ohd[nv*j+i] = ohd[nv*i+j];
70          }
71       }
72    for (i = 0; i < nv; i++)  {
73       notdone[i] = 1;
74       mind[i] = largeint;
75    }
76    mind[0] = 0;
77    while (dbg) ;  // stalling so can attach debugger
78 }
79
80 // finds closest to 0 among notdone, among startv through endv
81 void findmymin()
82 {  int i;
83    mymin[0] = largeint;
84    for (i = startv; i <= endv; i++)
85       if (notdone[i] && mind[i] < mymin[0])  {
86          mymin[0] = mind[i];
87          mymin[1] = i;
88       }
89 }
90
91 void findoverallmin()
92 {  int i;
93    MPI_Status status;  // describes result of MPI_Recv() call
94    // nodes other than 0 report their mins to node 0, which receives
95    // them and updates its value for the global min
96    if (me > 0)
97       MPI_Send(mymin,2,MPI_INT,0,MYMIN_MSG,MPI_COMM_WORLD);
98    else  {
99       // check my own first
100      overallmin[0] = mymin[0];
101      overallmin[1] = mymin[1];
```

```
102         // check the others
103         for (i = 1; i < nnodes; i++)  {
104            MPI_Recv(othermin,2,MPI_INT,i,MYMIN_MSG,MPI_COMM_WORLD,&status);
105            if (othermin[0] < overallmin[0])  {
106               overallmin[0] = othermin[0];
107               overallmin[1] = othermin[1];
108            }
109         }
110      }
111   }
112
113   void updatemymind()  // update my mind segment
114   {  // for each i in [startv,endv], ask whether a shorter path to i
115      // exists, through mv
116      int i, mv = overallmin[1];
117      unsigned md = overallmin[0];
118      for (i = startv; i <= endv; i++)
119         if (md + ohd[mv*nv+i] < mind[i])
120            mind[i] = md + ohd[mv*nv+i];
121   }
122
123   void disseminateoverallmin()
124   {  int i;
125      MPI_Status status;
126      if (me == 0)
127         for (i = 1; i < nnodes; i++)
128            MPI_Send(overallmin,2,MPI_INT,i,OVRLMIN_MSG,MPI_COMM_WORLD);
129      else
130         MPI_Recv(overallmin,2,MPI_INT,0,OVRLMIN_MSG,MPI_COMM_WORLD,&status);
131   }
132
133   void updateallmind()  // collects all the mind segments at node 0
134   {  int i;
135      MPI_Status status;
136      if (me > 0)
137         MPI_Send(mind+startv,chunk,MPI_INT,0,COLLECT_MSG,MPI_COMM_WORLD);
138      else
139         for (i = 1; i < nnodes; i++)
140            MPI_Recv(mind+i*chunk,chunk,MPI_INT,i,COLLECT_MSG,MPI_COMM_WORLD,
141               &status);
142   }
143
144   void printmind()  // partly for debugging (call from GDB)
145   {  int i;
146      printf("minimum distances:\n");
147      for (i = 1; i < nv; i++)
148         printf("%u\n",mind[i]);
149   }
150
151   void dowork()
152   {  int step,  // index for loop of nv steps
153          i;
154      if (me == 0) T1 = MPI_Wtime();
155      for (step = 0; step < nv; step++)  {
156         findmymin();
```

```
157         findoverallmin();
158         disseminateoverallmin();
159         // mark new vertex as done
160         notdone[overallmin[1]] = 0;
161         updatemymind(startv,endv);
162      }
163      updateallmind();
164      T2 = MPI_Wtime();
165   }
166
167   int main(int ac, char **av)
168   {  int i,j,print;
169      init(ac,av);
170      dowork();
171      print = atoi(av[2]);
172      if (print && me == 0)  {
173         printf("graph weights:\n");
174         for (i = 0; i < nv; i++)  {
175            for (j = 0; j < nv; j++)
176               printf("%u  ",ohd[nv*i+j]);
177            printf("\n");
178         }
179         printmind();
180      }
181      if (me == 0) printf("time at node 0: %f\n",(float)(T2-T1));
182      MPI_Finalize();
183   }
184
```

The various MPI functions will be explained in the next section.

2.3 Introduction to MPI APIs

2.3.1 MPI_Init() and MPI_Finalize()

These are required for starting and ending execution of an MPI program. Their actions may be implementation-dependent. For instance, if our platform is a NOW, **MPI_Init()** may set up the TCP/IP sockets via which the various nodes communicate with each other.

2.3.2 MPI_Comm_size() and MPI_Comm_rank()

In our function **init()** above, note the calls

```
MPI_Comm_size(MPI_COMM_WORLD,&nnodes);
MPI_Comm_rank(MPI_COMM_WORLD,&me);
```

The first call determines how many nodes are participating in our computation, placing the result in our variable **nnodes**. Here **MPI_COMM_WORLD** is our node group, termed a **communicator** in MPI par-

lance. MPI allows the programmer to subdivide the nodes into groups, to facilitate performance and clarity of code. Note that for some operations, such as barriers, the only way to apply the operation to a proper subset of all nodes is to form a group. The totality of all groups is denoted by **MPI_COMM_WORLD**. In our program here, we are not subdividing into groups.

The second call determines this node's ID number, called its **rank**, within its group. As mentioned earlier, even though the nodes are all running the same program, they are typically working on different parts of the program's data. So, the program needs to be able to sense which node it is running on, so as to access the appropriate data. Here we record that information in our variable **me**.

2.3.3 MPI_Send

To see how MPI's basic send function works, consider our line above,

```
MPI_Send(mymin,2,MPI_INT,0,MYMIN_MSG,MPI_COMM_WORLD);
```

Let's look at the arguments:

mymin :

We are sending a set of bytes. This argument states the address at which these bytes begin.

2, MPI_INT :

This says that our set of bytes to be sent consists of 2 objects of type **MPI_INT**. That means 8 bytes on today's standard 32-bit machines, so why not just collapse these two arguments to one, namely the number 8? Why did the designers of MPI bother to define data types? The answer is that we want to be able to run MPI on a heterogeneous set of machines, with MPI serving as the "broker" between them in case different architectures among those machines handle data differently.

First of all, there is the issue of **endianness**. Intel machines, for instance, are **little-endian**, which means that the least significant byte of a memory word has the smallest address among bytes of the word. Sun SPARC chips, on the other hand, are **big-endian**, with the opposite storage scheme. If our set of nodes included machines of both types, straight transmission of sequences of 8 bytes might mean that some of the machines literally receive the data backwards!

Secondly, these days 64-bit machines are becoming more and more common. Again, if our set of nodes were to include both 32-bit and 64-bit words, some major problems would occur if no conversion were done.

0 :

We are sending to node 0.

MYMIN_MSG :

This is the message type, programmer-defined in our line

```
#define MYMIN_MSG 0
```

Receive calls, described in the next section, can ask to receive only messages of a certain type.

PI_COMM_WORLD :

This is the node group to which the message is to be sent. Above, where we said we are sending to node 0, we technically should say we are sending to node 0 within the group **MPI_COMM_WORLD**.

2.3.4 MPI_Recv()

Let's now look at the arguments for a basic receive:

```
MPI_Recv(othermin,2,MPI_INT,i,MYMIN_MSG,MPI_COMM_WORLD,&status);
```

othermin :

The received message is to be placed at our location **othermin**.

2,MPI_INT :

Two objects of **MPI_INT** type are to be received.

i :

Receive only messages of from node **i**. If we did not care what node we received a message from, we could specify the value **MPI_ANY_SOURCE**.

MYMIN_MSG :

Receive only messages of type **MYMIN_MSG**. If we did not care what type of message we received, we would specify the value **MPI_ANY_TAG**.

PI_COMM_WORLD :

Group name.

status :

Recall our line

```
MPI_Status status;  // describes result of MPI_Recv() call
```

The type is an MPI **struct** containing information about the received message. Its primary fields of interest are **MPI_SOURCE**, which contains the identity of the sending node, and **MPI_TAG**, which contains the message type. These would be useful if the receive had been done with **MPI_ANY_SOURCE** or **MPI_ANY_TAG**; the status argument would then tell us which node sent the message and what type the message was.

3 Collective Communications

3.1 The MPI_Bcast Operation

In our example program above, we had a number of loops like

```
for (i = 1; i < nnodes; i++)
   MPI_Send(overallmin,2,MPI_INT,i,OVRLMIN_MSG,MPI_COMM_WORLD);
```

We can reply this by

```
MPI_Bcast(overallmin,2,MPI_INT,0,MPI_COMM_WORLD);
```

In English, this call would say,

> At this point all nodes participate in a broadcast operation, in which node 0 sends 2 objects of type **MPI_INT**. The source of the data will be located at address **overallmin** at node 0, and the other nodes will receive the data at a location of that name.

Note my word "participate" above. The name of the function is "broadcast," which makes it sound like only node 0 executes this line of code, which is not the case; all the nodes in the group (in this case that means all nodes in our entire computation) execute this line. The only difference is the action; most nodes participate by receiving, while node 0 participates by sending.

Why might this be preferable than using an explicit loop?

First, it would obviously be much clearer. That makes the program easier to write, easier to debug, and easier for others (and ourselves, later) to read.

But even more importantly, using the broadcast may improve performance. We may, for instance, be using an implementation of MPI which is tailored to the platform on which we are running MPI. If for instance we are running on a network designed for parallel computing, such as Myrinet or Infiniband, an optimized broadcast may achieve a much higher performance level than would simply a loop with individual send calls. On a shared-memory multiprocessor system, special machine instructions specific to that platform's architecture can be exploited, as for instance IBM has done for its shared-memory machines. Even on an ordinary Ethernet, one could exploit Ethernet's own broadcast mechanism, as had been done for PVM, a system like MPI (G. Davies and N. Matloff, Network-Specific Performance Enhancements for PVM, *Proceedings of the Fourth IEEE International Symposium on High-Performance Distributed Computing*, 1995, 205-210).

The function **MPI_Bcast()** is an example of MPI's **collective communication** capabilities, a number of which are used in the following refinement of the Dijkstra program above:

3.2 Example

```
1  // Dijkstra.coll1.c
2
3  // MPI example program:  Dijkstra shortest-path finder in a
4  // bidirectional graph; finds the shortest path from vertex 0 to all
5  // others; this version uses collective communication
6
7  // command line arguments:  nv print dbg
8
9  // where:  nv is the size of the graph; print is 1 if graph and min
10 // distances are to be printed out, 0 otherwise; and dbg is 1 or 0, 1
11 // for debug
12
13 // node 0 will both participate in the computation and serve as a
14 // "manager"
15
16 #include <stdio.h>
17 #include <mpi.h>
18
19 // global variables (but of course not shared across nodes)
20
21 int nv,  // number of vertices
22     *notdone, // vertices not checked yet
23     nnodes,  // number of MPI nodes in the computation
24     chunk,  // number of vertices handled by each node
25     startv,endv,  // start, end vertices for this node
26     me,  // my node number
27     dbg;
28 unsigned largeint,  // max possible unsigned int
29          mymin[2],  // mymin[0] is min for my chunk,
30                     // mymin[1] is vertex which achieves that min
31          overallmin[2],  // overallmin[0] is current min over all nodes,
32                          // overallmin[1] is vertex which achieves that min
33          *ohd,  // 1-hop distances between vertices; "ohd[i][j]" is
34                 // ohd[i*nv+j]
35          *mind;  // min distances found so far
36
37 double T1,T2;  // start and finish times
38
39 void init(int ac, char **av)
40 {  int i,j,tmp; unsigned u;
41    nv = atoi(av[1]);
42    dbg = atoi(av[3]);
43    MPI_Init(&ac,&av);
44    MPI_Comm_size(MPI_COMM_WORLD,&nnodes);
45    MPI_Comm_rank(MPI_COMM_WORLD,&me);
46    chunk = nv/nnodes;
47    startv = me * chunk;
48    endv = startv + chunk - 1;
49    u = -1;
50    largeint = u >> 1;
51    ohd = malloc(nv*nv*sizeof(int));
52    mind = malloc(nv*sizeof(int));
53    notdone = malloc(nv*sizeof(int));
```

```
54    // random graph
55    // note that this will be generated at all nodes; could generate just
56    // at node 0 and then send to others, but faster this way
57    for (i = 0; i < nv; i++)
58       for (j = i; j < nv; j++)  {
59          if (j == i) ohd[i*nv+i] = 0;
60          else  {
61             ohd[nv*i+j] = rand() % 20;
62             ohd[nv*j+i] = ohd[nv*i+j];
63          }
64       }
65    for (i = 0; i < nv; i++)  {
66       notdone[i] = 1;
67       mind[i] = largeint;
68    }
69    mind[0] = 0;
70    while (dbg) ;  // stalling so can attach debugger
71 }
72
73 // finds closest to 0 among notdone, among startv through endv
74 void findmymin()
75 {  int i;
76    mymin[0] = largeint;
77    for (i = startv; i <= endv; i++)
78       if (notdone[i] && mind[i] < mymin[0])  {
79          mymin[0] = mind[i];
80          mymin[1] = i;
81       }
82 }
83
84 void updatemymind()  // update my mind segment
85 {  // for each i in [startv,endv], ask whether a shorter path to i
86    // exists, through mv
87    int i, mv = overallmin[1];
88    unsigned md = overallmin[0];
89    for (i = startv; i <= endv; i++)
90       if (md + ohd[mv*nv+i] < mind[i])
91          mind[i] = md + ohd[mv*nv+i];
92 }
93
94 void printmind()  // partly for debugging (call from GDB)
95 {  int i;
96    printf("minimum distances:\n");
97    for (i = 1; i < nv; i++)
98       printf("%u\n",mind[i]);
99 }
100
101 void dowork()
102 {  int step,  // index for loop of nv steps
103        i;
104    if (me == 0) T1 = MPI_Wtime();
105    for (step = 0; step < nv; step++)  {
106       findmymin();
107       MPI_Reduce(mymin,overallmin,1,MPI_2INT,MPI_MINLOC,0,MPI_COMM_WORLD);
108       MPI_Bcast(overallmin,1,MPI_2INT,0,MPI_COMM_WORLD);
```

```
109        // mark new vertex as done
110        notdone[overallmin[1]] = 0;
111        updatemymind(startv,endv);
112     }
113     // now need to collect all the mind values from other nodes to node 0
114     MPI_Gather(mind+startv,chunk,MPI_INT,mind,chunk,MPI_INT,0,MPI_COMM_WORLD);
115     T2 = MPI_Wtime();
116  }
117
118  int main(int ac, char **av)
119  {  int i,j,print;
120     init(ac,av);
121     dowork();
122     print = atoi(av[2]);
123     if (print && me == 0)  {
124        printf("graph weights:\n");
125        for (i = 0; i < nv; i++)  {
126           for (j = 0; j < nv; j++)
127              printf("%u  ",ohd[nv*i+j]);
128           printf("\n");
129        }
130        printmind();
131     }
132     if (me == 0) printf("time at node 0: %f\n",(float)(T2-T1));
133     MPI_Finalize();
134  }
```

The new calls will be explained in the next section.

3.3 Introduction to MPI APIs for Collective Operations

3.3.1 MPI_Reduce

Look at our call

```
MPI_Reduce(mymin,overallmin,1,MPI_2INT,MPI_MINLOC,0,MPI_COMM_WORLD);
```

above. In English, this would say,

> At this point all nodes in this group participate in a "reduce" operation. The type of reduce operation is **MPI_MINLOC**, which means that the minimum value among the nodes will be computed, and the index attaining that minimum will be recorded as well. Each node contributes a value to be checked, and an associated index, from a location **mymin** in their programs; the type of the pair is **MPI_2INT**. The min value/index will be computed at node 0, where they will be placed at a location **overallmin**.

There is also the function **MPI_Allreduce()**, which does the same operation, except that instead of just depositing the result at one node, it does so at all nodes. So for instance our code above,

```
MPI_Reduce(mymin,overallmin,1,MPI_2INT,MPI_MINLOC,0,MPI_COMM_WORLD);
MPI_Bcast(overallmin,1,MPI_2INT,0,MPI_COMM_WORLD);
```

could be replaced by

```
MPI_Allreduce(mymin,overallmin,1,MPI_2INT,MPI_MINLOC,MPI_COMM_WORLD);
```

Again, these can be optimized for particular platforms.

3.3.2 The MPI_Gather Operation

A classical approach to parallel computation is to first break the data for the application into chunks, then have each node work on its chunk, and then gather all the processed chunks together at some node. The MPI function **MPI_Gather()** does this.

In our program above, look at the line

```
MPI_Gather(mind+startv,chunk,MPI_INT,mind,chunk,MPI_INT,0,MPI_COMM_WORLD);
```

In English, this says,

> At this point all nodes participate in a gather operation, in which each node contributes data, consisting of **chunk** number of MPI integers, from a location **mind+startv** in its program. All that data is strung together and deposited at the location **mind** in the program running at node 0.

There is also **MPI_Allgather()**, which places the result at all nodes, not just one.

3.3.3 The MPI_Scatter Operation

This is the opposite of **MPI_Gather()**, i.e. it breaks long data into chunks which it parcels out to individual nodes.

3.3.4 The MPI_Barrier Operation

This implements a barrier for a given communicator. The name of the communicator is the sole argument for the function.

3.4 Creating Communicators

Again, a communicator is a subset (either proper or improper) of all of our nodes. MPI includes a number of functions for use in creating communicators. Some set up a virtual "topology" among the nodes.

For instance, many physics problems consist of solving differential equations in two- or three-dimensional space, via approximation on a grid of points. In two dimensions, groups may consists of rows in the grid.

We will not pursue this further here.

4 Buffering, Synchrony and Related Issues

As noted several times so far, interprocess communication in parallel systems can be quite expensive in terms of time delay. In this section we will consider some issues which can be extremely important in this regard.

4.1 Buffering

To understand this point, first consider situations in which MPI is running on some network, under the TCP/IP protocol. Say node A is sending to node B.

The program at node A will have set up a socket to B during the call to **MPI_Init()**. The other end of the socket will be a corresponding one at B. We describe the setting up of this socket pair as establishing a **connection** between A and B. When node A calls **MPI_Send()**, the latter function will write to the socket. When node B calls **MPI_Recv()**, it will read from its socket.

Now, it is important to recall that the totality of bytes sent by A to B during lifetime of the connection is considered one long message. So for instance if A writes to the socket five times, it will not be perceived at B as five messages, but rather just one long message (in fact, only part of one long message, since more may be yet to come).

On the other hand, even though that data is considered one long message, it may physically be sent out in pieces. This doesn't correspond to the pieces written to the socket. Rather, the breaking into pieces is done for the purpose of **flow control**, meaning that for example A will not send data to B if the operating system (OS) at B has no room for it. The **buffer** space the OS at B has set up for receiving data is limited. As A is sending to B, the TCP layer at B is telling its counterpart at A when A is allowed to send more data.

Let's say that our MPI implementation at the internal level is threaded, with one thread for the application and one from doing network I/O. Again, this is internal, unseen by the application programmer. Let's assume that the application itself is not threaded. The I/O thread is using something like a **select()** call to determine when new data has arrived from the network.

Think of what happens B calls **MPI_Recv()**, requesting to receive from A, with a certain tag. Say the first argument is named **x**, i.e. the data to be received is to be deposited at **x**. The **MPI_Recv()** function will look

at the byte stream accumulated by the I/O thread, and search within that stream for a message from A of the given type. If found, the function will remove that message from the stream, and place the data in **x**.

4.2 Nonbuffered Communication

You can see for all this that MPI applications which run on top of TCP/IP have a natural buffering system. In fact, there is likely additional buffering as well. By contrast, some other platforms may not have any buffering at all. This is not the usual situation, but it could be the case, for instance, when the underlying platform is a shared-memory multiprocessor.

Furthermore, buffering slows everything down. In our TCP scenario above, **MPI_Recv()** at B must copy the message in the incoming byte stream to **x**. This is definitely a blow to performance. That in fact is why networks developed specially for parallel processing typically include mechanisms to avoid the copying. Infiniband, for example, has a Remote Direct Memory Access capability, meaning that A can write directly to **x** at B.

So, we may either have a no-buffering situation forced upon us, or may opt for no buffering for performance reasons. But that has a big implication: Node A cannot call **MPI_Send()** until node B has called **MPI_Recv()**; otherwise B may be using the space at **x**, in which case A's premature **MPI_Send()** would ruin things at that location. That would mean that B would have to inform A when it calls **MPI_Recv()**. This is called **synchronous** communication. Clearly, this can be a major cause of slowdown if not handled carefully.

4.3 Safety

Moreover, synchronous communication has a risk of setting up deadlocks. Say A wants to send two messages to B, of types U and V, but that B wants to receive V first. Then A won't even get to send V, because in preparing to send U it must wait for a notice from B that B wants to read U—a notice which will never come, because B sends such a notice for V first. This would not occur if the communication were asynchronous.

But beyond formal deadlock, programs can fail in other ways, even with buffering, as buffer space is always by nature finite. A program can fail if it runs out of buffer space, either at the sender or the receiver. See `www.llnl.gov/computing/tutorials/mpi_performance/samples/unsafe.c` for an example of a test program which demonstrates this on a certain platform, by deliberating overwhelming the buffers at the receiver.

In MPI terminology, asynchronous communication is considered **unsafe**. The program may run fine on most systems, as most systems are buffered, but fail on some systems. Of course, as long as you know your program won't be run in nonbuffered settings, it's fine, and since there is potentially such a performance penalty for doing things synchronously, most people are willing to go ahead with their "unsafe" code.

4.4 Living Dangerously

If one is sure that there will be no problems of buffer overflow and so on, one can use variant send and receive calls provided by MPI, such as **MPI_Isend()** and **MPI_Irecv()**. The key difference between them and **MPI_Send()** and **MPI_Recv()** is that they return immediately, and thus are termed **nonblocking**. Your code can go on and do other things, not having to wait.

4.5 Safe Exchange Operations

In many applications A and B are swapping data, so both are sending and both are receiving. This too can lead to deadlock. An obvious solution would be, for instance, to have the lower-rank node send first and the higher-rank node receive first. But a more convenient, safer and possibly faster alternative would be to use MPI's **MPI_Sendrecv()** function.

This does mean that at A you cannot touch the data you are sending until you determine that it has either been buffered somewhere or has reached **x** at B. Similarly, at B you can't use the data at **x** until you determine that it has arrived. Such determinations can be made via **MPI_Wait()**. In other words, you can do your send or receive, then perform some other computations for a while, and then call **MPI_Wait()** to determine whether you can go on.