CSC 447: Parallel Programming for Multi-Core and Cluster Systems

Message Passing with MPI

Instructor: Haidar M. Harmanani

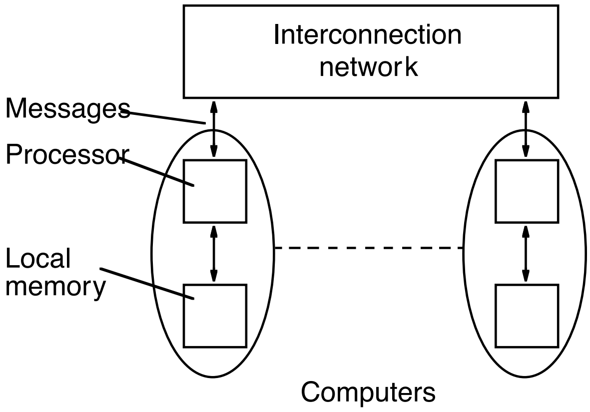
Spring 2017

Outline

- Message-passing model
- Message Passing Interface (MPI)
- Coding MPI programs
- Compiling MPI programs
- Running MPI programs
- Benchmarking MPI programs
- Mixing MPI and Pthreads

Message-Passing Multicomputer

- Complete computers connected through an interconnection network:



The Message-Passing Model

- Recall that a *process* has its own address space, and may have multiple *threads* sharing a single address space.
- MPI is used for communication among processes, which have separate address spaces.
- Interprocess communication consists of
  – Synchronization
  – Movement of data from one process's address space to another's.

Message Passing Features

- Simplicity
  - The basics of the paradigm are traditional communication operations.
- Generality
  - Can be implemented on most parallel architectures.
- Performance
  - The implementation can match the underlying hardware.
- Scalability
  - The same program can be deployed on larger systems

MPI

- Standard for operations in message passing
- Led by MPI Forum (academia & industry)
  - Standards
    - MPI-1 (1994)
    - MPI-2 standard (1997)
    - MPI-3 (2012)
- Implementations
  - Open-source: MPICH, OpenMPI
  - Proprietary: Cray, IBM, Intel

MPI Sources

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Books:
  - Using MPI: Portable Parallel Programming with the Message-Passing Interface, by Gropp, Lusk, and Skjellum, MIT Press, 1994.
  - MPI: The Complete Reference, by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1996.
  - Designing and Building Parallel Programs, by Ian Foster, Addison-Wesley, 1995.
  - Parallel Programming with MPI, by Peter Pacheco, Morgan-Kaufmann, 1997.
  - MPI: The Complete Reference Vol 1 and 2,MIT Press, 1998(Fall).
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages

What is MPI?

- A message-passing library specification
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks

Hello, MPI World!

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

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, MPI world!\n" );
    MPI_Finalize();
    return 0;
}
```


Notes on MPI C

- **mpi.h** must be **#included**
- **MPI_Init** initializes the MPI execution environment
- MPI functions return error codes or **MPI_SUCCESS**
  - By default, an error causes all processes to abort.
  - The user can cause routines to return (with an error code) instead.
  - A user can also write and install custom error handlers.
  - Libraries might want to handle errors differently from applications.

Running MPI Programs

- The MPI standard does not specify how to run an MPI program
- In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment variables.
- **`mpiexec <args>`** is part of MPI-2, as a recommendation, but not a requirement
- **`mpirun -np <p> <exec> <args>`**

Execution on 3 CPUs

- `% mpirun -np 3 sat`
- `0) 0110111110011001`
- `0) 1110111111011001`
- `2) 1010111110011001`
- `1) 1110111110011001`
- `1) 1010111111011001`
- `1) 0110111110111001`
- `0) 1010111110111001`
- `2) 0110111111011001`
- `2) 1110111110111001`
- `Process 1 is done`
- `Process 2 is done`
- `Process 0 is done`

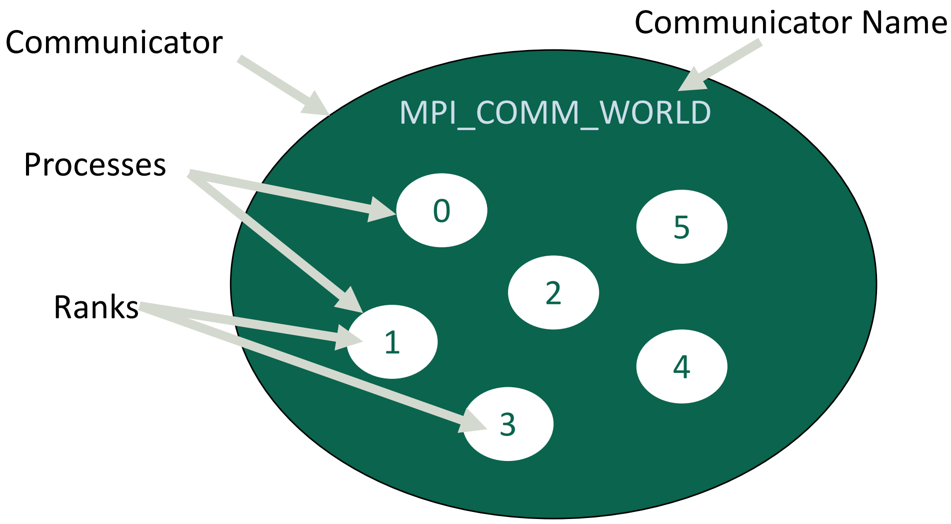
Some Basic Concepts

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

Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - **`MPI_Comm_size`** reports the number of processes.
  - **`MPI_Comm_rank`** reports the *rank*, a number between 0 and size-1, identifying the calling process
- MPI Ranks
  - Ranks have private memory
  - Each rank has a unique identification number
  - Ranks are numbered sequentially: [0, n-1]

Communicator



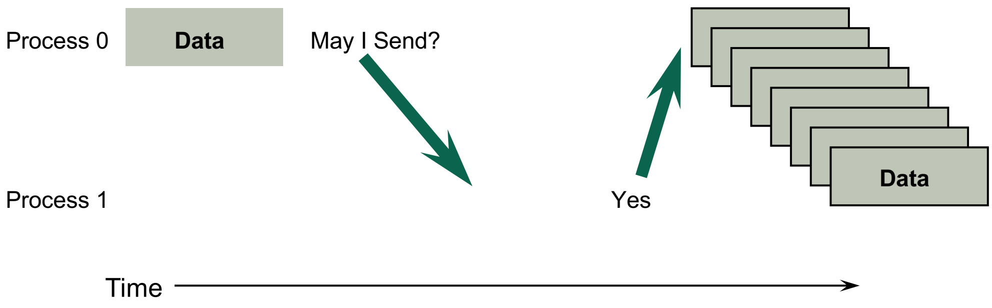
A Better Hello, MPI World!

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "Hello world from rank %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}
```

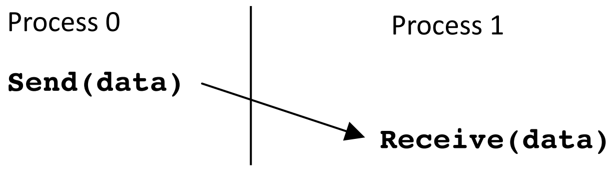
Message Passing

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



MPI Basic Send/Receive

- We need to fill in the details in



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

Single-Program Multiple-Data (SPMD)

- All processes run the same program, each accesses a different portion of data.
- All processes are launched simultaneously.
- Communication:
  - Point-to-point messages.
  - Collective communication operations.

Using SPMD Computational Model

```
main (int argc, char *argv[])
{
MPI_Init(&argc, &argv);
        .
        .
/*find process rank */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank == 0)
        master();
else
        slave();
        .
        .
MPI_Finalize();
}
```

master() and slave() are to be executed by master process and slave process, respectively.

Point to Point Communication

- Blocking Communication
  - Block until completed (*send stuff on your own*)
- Non-blocking Communication
  - Return without waiting for completion (*give them to someone else*)
- Forms of Sends:
  - Synchronous: message gets sent only when it is known that someone is already waiting at the other end (*think fax*)
  - Buffered: message gets sent and if someone is waiting for it so be it; otherwise it gets saved in a temporary buffer until someone retrieves it. (*think mail*)
  - Ready: Like synchronous, only there is no acknowledgement that there is a matching receive at the other end, just a programmer's assumption! (Use it with extreme care)

MPI: Point-to-point Communication

- MPI blocking standard send:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
              MPI_Comm comm, MPI_Status *status)
```

  - buf is the starting address of the array
  - count is its length
  - datatype is its MPI datatype
  - comm is the communicator context
  - dest is the rank of the destination process in comm
  - tag is an extra distinguishing number, like a note
  - Receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error

Example

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank); /* find rank */

if (myrank == 0) {
  int x;
  MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
} else if (myrank == 1) {
  int x;
  MPI_Recv(&x, 1,
  MPI_INT,0,msgtag,MPI_COMM_WORLD,status);
}
```

To send an integer x from process 0 to process 1

Retrieving Further Information

- **`Status`** is a data structure allocated in the user's program.

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

Other Blocking Sends

- `MPI_Ssend`–Blocking Synchronous send
  - The sender notifies the receiver; after the matching receive is posted
  - The receiver acks back and the sender sends the message.
- `MPI_Bsend`–Blocking Buffered (asynchronous) send
  - The sender notifies the receiver and the message is either buffered on the sender side or the receiver side according to size until a matching receive forces a network transfer or a local copy respectively.
- `MPI_Rsend`–Blocking Ready send
  - The receiver is notified and the data starts getting sent immediately following that

Blocking Send Performance

- Synchronous sends offer the highest asymptotic data rate (AKA bandwidth) but the startup cost (latency) is very high, and they run the risk of deadlock.
- Buffered sends offer the lowest latency but:
  - suffer from buffer management complications
  - have bandwidth problems because of the extra copies and system calls
- Ready sends *should* offer the best of both worlds but are so prone to cause trouble they are to be avoided!
- Standard sends are usually the ones that are most carefully optimized by the implementers.
  - For large message sizes they can always deadlock.

MPI: Collective Communication

- Collective operations are called by all processes in a communicator.
- **`MPI_BCAST`** distributes data from one process (the root) to all others in a communicator.
- **`MPI_REDUCE`** combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, **`SEND/RECEIVE`** can be replaced by **`BCAST/REDUCE`**, improving both simplicity and efficiency.

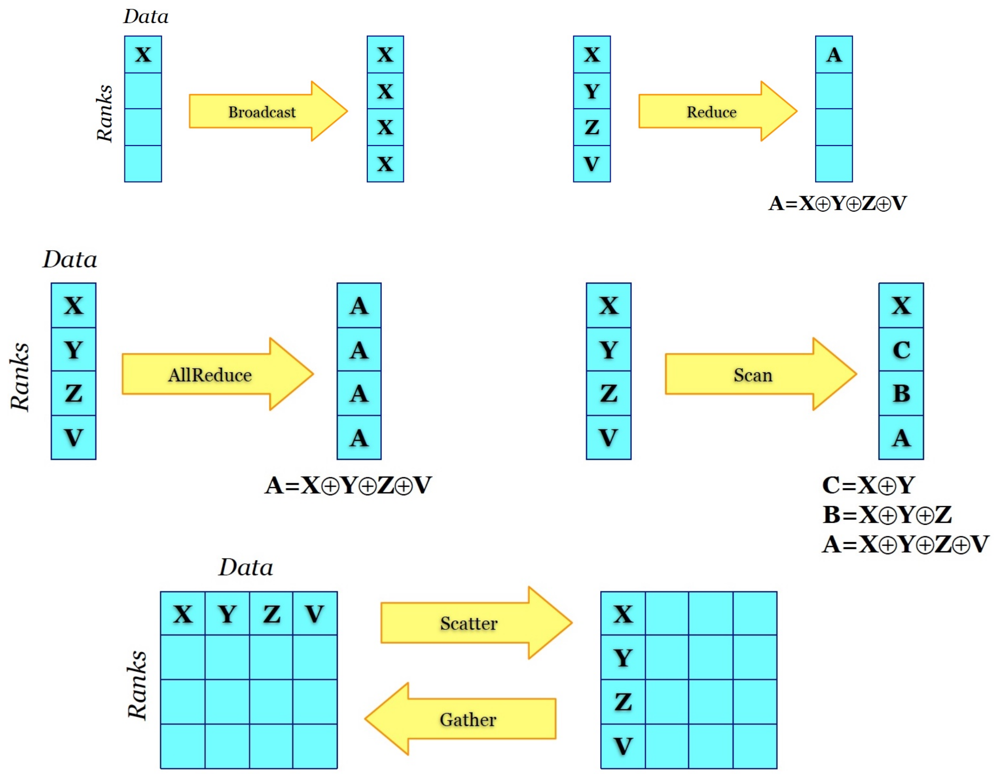
MPI: Collective Communication

- Instructions to exchange data including all the ranks in a communicator
- The root rank indicates the source or destination of the operation
- Broadcast: one to many

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm)
```

- Reduction: many to one

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```



MPI Datatypes

- The data in a message to sent or received is described by a triple (address, count, datatype), where
- An MPI *datatype* is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE_PRECISION)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, such an array of (int, float) pairs, or a row of a matrix stored column-wise.

Other MPI Datatypes

- MPI_CHAR
- MPI_BYTE
- MPI_SHORT
- MPI_FLOAT
- MPI_LONG
- MPI_UNSIGNED_CHAR
- MPI_UNSIGNED_SHORT
- MPI_UNSIGNED
- MPI_UNSIGNED_LONG
- MPI_LONG_DOUBLE
- MPI_LONG_LONG_INT
- MPI_PACKED
- MPI_FLOAT_INT
  - struct { float, int }
- MPI_LONG_INT
- MPI_DOUBLE_INT
- MPI_SHORT_INT
- MPI_2INT
- MPI_LONG_DOUBLE_INT

Why Datatypes?

- Since all data is labeled by type, an MPI implementation can support communication between processes on machines with very different memory representations and lengths of elementary datatypes (heterogeneous communication).
- Specifying application-oriented layout of data in memory
  - reduces memory-to-memory copies in the implementation
  - allows the use of special hardware (scatter/gather) when available

MPI Tags

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

Tags and Contexts

- Separation of messages used to be accomplished by use of tags, but
  - this requires libraries to be aware of tags used by other libraries.
  - this can be defeated by use of "wild card" tags.
- Contexts are different from tags
  - no wild cards allowed
  - allocated dynamically by the system when a library sets up a communicator for its own use.
- User-defined tags still provided in MPI for user convenience in organizing application
- Use MPI_Comm_split to create new communicators

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - `MPI_INIT`
  - `MPI_FINALIZE`
  - `MPI_COMM_SIZE`
  - `MPI_COMM_RANK`
  - `MPI_SEND`
  - `MPI_RECV`
- Point-to-point (send/recv) isn't the only way...

Example: PI in C

```
#include <mpi.h>
#include <math.h>
int main(int argc, char *argv[])
{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.141592653589793238462643;
   double mypi, pi, h, sum, x, a;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   while (!done)  {
     if (myid == 0) {
       printf("Enter the number of intervals: (0 quits) ");
       scanf("%d",&n);
     }
     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
     if (n == 0) break;
```

Example: PI in C (Continued)

```
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0)
      printf("pi is approximately %.16f, Error is %.16f\n",
              pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
  return 0;
}
```


Example 2:

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#define MAXSIZE 1000
void main(int argc, char *argv)
{
   int myid, numprocs;
   int data[MAXSIZE], i, x, low, high, myresult, result;
   char fn[255];
   char *fp;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

(Continued)

```c
if (myid == 0) {  /* Open input file and initialize data */
          strcpy(fn, getenv("HOME"));
          strcat(fn,"/MPI/rand_data.txt");
          if ((fp = fopen(fn,"r")) == NULL) {
                     printf("Can't open the input file: %s\n\n", fn);
                     exit(1);
          }
          for(i = 0; i < MAXSIZE; i++) fscanf(fp,"%d", &data[i]);
}
MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD); /* broadcast data */
x = n/nproc; /* Add my portion Of data */
low = myid * x;
high = low + x;
for(i = low; i < high; i++)
    myresult += data[i];
printf("I got %d from %d\n", myresult, myid); /* Compute global sum */
MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
if (myid == 0) printf("The sum is %d.\n", result);
MPI_Finalize();
}
```


Alternative set of 6 Functions for Simplified MPI

- MPI_INIT
- MPI_FINALIZE
- MPI_COMM_SIZE
- MPI_COMM_RANK
- MPI_BCAST
- MPI_REDUCE

Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

This is called "unsafe" because it depends on the availability of system buffers

Some Solutions to the "unsafe" Problem

- Order the operations more carefully:

Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

Use non-blocking operations:

Process 0	Process 1
`Isend(1)`	`Isend(0)`
`Irecv(1)`	`Irecv(0)`
`Waitall`	`Waitall`

When to use MPI

- Portability and Performance
- Irregular Data Structures
- Building Tools for Others
  - Libraries
- Need to Manage memory on a per processor basis

Benchmarking MPI Programs

- **`MPI_Barrier`** — barrier synchronization
- **`MPI_Wtick`** — timer resolution
- **`MPI_Wtime`** — current time

Benchmarking MPI Programs

- ```
  double elapsed_time;
  …
  ```
- ```
  MPI_Init (&argc, &argv);
  MPI_Barrier (MPI_COMM_WORLD);
  elapsed_time = - MPI_Wtime();
  …
  ```
- ```
  MPI_Reduce (…);
  elapsed_time += MPI_Wtime();
  ```


When *not* to use MPI

- Regular computation matches HPF
  - But see PETSc/HPF comparison (ICASE 97-72)
- Solution (e.g., library) already exists
  - http://www.mcs.anl.gov/mpi/libraries.html
- Require Fault Tolerance
  - Sockets
- Distributed Computing
  - CORBA, DCOM, etc.
  ```

Mixing MPI with Pthreads

- Each MPI process typically creates and then manages **N** threads, where **N** makes the best use of the available cores/node.
- Finding the best value for **N** will vary with the platform and your application's characteristics.
- In general, there may be problems if multiple threads make MPI calls.
  - The program may fail or behave unexpectedly.
- If MPI calls must be made from within a thread, they should be made only by one thread.

Mixing MPI with Pthreads: `dotprod`

```
void *dotprod(void *arg)
{
   int i, start, end, len, numthrds, myid;
   long mythrd;
   double mysum, *x, *y;

   mythrd = (long)arg;
   MPI_Comm_rank (MPI_COMM_WORLD, &myid);

   numthrds = dotstr.numthrds;
   len = dotstr.veclen;
   start = myid*numthrds*len + mythrd*len;
   end   = start + len;
   x = dotstr.a;
   y = dotstr.b;
```

Continued

Mixing MPI with Pthreads: `dotprod`

```
 /* Perform the dot product and assign result to the appropriate variable in the
 structure. */

 mysum = 0;
 for (i=start; i<end ; i++)
  {
    mysum += (x[i] * y[i]);
  }

 /* Lock a mutex prior to updating the value in the structure, and unlock it
 upon updating.*/

 pthread_mutex_lock (&mutexsum);
 printf("Task %d thread %ld adding partial sum of %f to node sum of %f\n",
         myid, mythrd, mysum, dotstr.sum);
 dotstr.sum += mysum;
 pthread_mutex_unlock (&mutexsum);

 pthread_exit((void*)0);
}
```

End dotprod()

Mixing MPI with Pthreads: `main`

```
int main(int argc, char* argv[])
{
int len=VECLEN, myid, numprocs;
long i;
int nump1, numthrds;
double *a, *b;
double nodesum, allsum;
void *status;
pthread_attr_t attr;

/* MPI Initialization */
MPI_Init (&argc, &argv);
MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &myid);

/* Assign storage and initialize values */
numthrds=MAXTHRDS;
a = (double*) malloc (numprocs*numthrds*len*sizeof(double));
b = (double*) malloc (numprocs*numthrds*len*sizeof(double));
...
```

Continued

Mixing MPI with Pthreads: `main`

```
a = (double*) malloc (numprocs*numthrds*len*sizeof(double));
b = (double*) malloc (numprocs*numthrds*len*sizeof(double));

for (i=0; i<len*numprocs*numthrds; i++) {
  a[i]=1;
  b[i]=a[i];
  }

dotstr.veclen = len;
dotstr.a = a;
dotstr.b = b;
dotstr.sum=0;
dotstr.numthrds=MAXTHRDS;

/* Create thread attribute to specify that the main thread needs
to join with the threads it creates. */

pthread_attr_init(&attr );
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
```

Continued

Mixing MPI with Pthreads: `main`

```
/* Create a mutex */
pthread_mutex_init (&mutexsum, NULL);

/* Create threads within this node to perform the dotproduct  */
for(i=0;i<numthrds;i++) {
  pthread_create( &callThd[i], &attr, dotprod, (void *)i);
  }

/* Release the thread attribute handle as it is no longer needed */
pthread_attr_destroy(&attr );

/* Wait on the other threads within this node */
for(i=0;i<numthrds;i++) {
  pthread_join( callThd[i], &status);
  }

nodesum = dotstr.sum;
printf("Task %d node sum is %f\n",myid, nodesum);
```

Continued

Mixing MPI with Pthreads: `main`

```
/* After the dot product, perform a summation of results on each node */
MPI_Reduce (&nodesum, &allsum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if (myid == 0)
printf ("Done. MPI with threads version: sum  =  %f \n", allsum);
MPI_Finalize();
free (a);
free (b);
pthread_mutex_destroy(&mutexsum);
exit (0);
}
```

End main()

Summary

- The parallel computing community has cooperated on the development of a standard for message-passing libraries.
- There are many implementations, on nearly all platforms.
- MPI subsets are easy to learn and use.
- Lots of MPI material is available.