Final Project

• Purpose:
  - A chance to dig in deeper into a parallel programming model and explore concepts.
  - Present results to work on communication of technical ideas.

• Write a non-trivial parallel program that combines two parallel programming languages/models. In some cases, just do two separate implementations:
  - OpenMP + SSE-3
  - OpenMP + CUDA (but need to do this in separate parts of the code)
  - MPI + OpenMP
  - MPI + SSE-3
  - MPI + CUDA

• Present results in a poster session on the last day of class.

Example Projects

• Look in the textbook or on-line
  - Recall Red/Blue from Ch. 4
    - Implement in MPI (+ SSE-3)
    - Implement main computation in CUDA
  - Algorithms from Ch. 5
  - SOR from Ch. 7
  - CUDA implementation?
  - FFT from Ch. 10
  - Jacobi from Ch. 10
  - Graph algorithms
  - Image and signal processing algorithms
  - Other domains...

Next Wednesday, November 3

• Use handin program on CADE machines
  - handin cs4961.pdesc <file, ascii or PDF ok>

• Projects can be individual or group efforts, with 1 to three students per project.

• Turn in 1 page project proposal
  - Algorithm to be implemented
  - Programming model(s)
  - Implementation plan
  - Validation and measurement plan

Final Project

• Purpose:
  - A chance to dig in deeper into a parallel programming model and explore concepts.
  - Present results to work on communication of technical ideas.

• Write a non-trivial parallel program that combines two parallel programming languages/models. In some cases, just do two separate implementations:
  - OpenMP + SSE-3
  - OpenMP + CUDA (but need to do this in separate parts of the code)
  - MPI + OpenMP
  - MPI + SSE-3
  - MPI + CUDA

• Present results in a poster session on the last day of class.

Example Projects

• Look in the textbook or on-line
  - Recall Red/Blue from Ch. 4
    - Implement in MPI (+ SSE-3)
    - Implement main computation in CUDA
  - Algorithms from Ch. 5
  - SOR from Ch. 7
  - CUDA implementation?
  - FFT from Ch. 10
  - Jacobi from Ch. 10
  - Graph algorithms
  - Image and signal processing algorithms
  - Other domains...

Next Wednesday, November 3

• Use handin program on CADE machines
  - handin cs4961.pdesc <file, ascii or PDF ok>

• Projects can be individual or group efforts, with 1 to three students per project.

• Turn in 1 page project proposal
  - Algorithm to be implemented
  - Programming model(s)
  - Implementation plan
  - Validation and measurement plan
A Few Words About Final Project

- **Purpose:**
  - A chance to dig in deeper into a parallel programming model and explore concepts.
  - Present results to work on communication of technical ideas.
- Write a non-trivial parallel program that combines two parallel programming languages/models. In some cases, just do two separate implementations.
  - OpenMP + SSE-3
  - OpenMP + CUDA (but need to do this in separate parts of the code)
  - TBB + SSE-3
  - MPI + OpenMP
  - MPI + SSE-3
  - MPI + CUDA
- Present results in a poster session on the last day of class.

Example Projects

- Look in the textbook or on-line
  - Recall Red/Blue from Ch. 4
  - Implement in MPI (+ SSE-3)
  - Implement main computation in CUDA
  - Algorithms from Ch. 5
  - SOR from Ch. 7
  - CUDA implementation?
  - FFT from Ch. 10
  - Jacobi from Ch. 10
  - Graph algorithms
  - Image and signal processing algorithms
  - Other domains...

Today's Lecture

- Message Passing, largely for distributed memory
- Message Passing Interface (MPI): a Local View language
- Chapter 7 in textbook (nicely done)
- Sources for this lecture

Message Passing and MPI

- Message passing is the principle alternative to shared memory parallel programming
- The dominant programming model for supercomputers and scientific applications on distributed clusters
  - Portable
  - Low-level, but universal and matches earlier hardware execution model
  - Based on Single Program, Multiple Data (SPMD)
  - Model with send() and recv() primitives
  - Isolation of separate address spaces
    - no data races
    - forces programmer to think about locality, so good for performance
    - architecture model exposed, so good for performance
  - Complexity and code growth!

Like OpenMP, MPI arose as a standard to replace a large number of proprietary message passing libraries.
Message Passing Library Features

- All communication, synchronization require subroutine calls
  - No shared variables
  - Program run on a single processor just like any uniprocessor program, except for calls to message passing library
- Subroutines for
  - Communication
    - Pairwise or point-to-point: Send and Receive
    - Collectives all processor get together to
      - Move data: Broadcast, Scatter/gather
      - Compute and move: sum, product, max, … of data on many processors
  - Synchronization
    - Barrier
    - No locks because there are no shared variables to protect
  - Queries
    - How many processes? Which one am I? Any messages waiting?

Books on MPI

- Designing and Building Parallel Programs, by Ian Foster, Addison-Wesley, 1995.
- Parallel Programming with MPI, by Peter Pacheco, Morgan-Kaufmann, 1997.

MPI References

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- 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

Working through an example

We’ll write some message-passing pseudo code for Count3 (from Lecture 4):

```
1 int accept(length)
2 int i
3 int total
4 for all i in 0-L-1 do
5   int size=ptime(array[i]);
6   int myData[1 size]=time(us)
7         MPI_Sendrecv(
8             size, myData, size, array[i][size],
9             size, myData, size, array[i][size],
10            )
11   if(myData[1 i]=3)
12      total += myData[1 i]
13    end for
14 end
```

- The class is global
- Number of desired threads
- Result of computation, grand total
- Global sum size of local part of global data
- Associate my part of global data with local variable
- Local accumulation
- Compute grand total
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.

Hello (C)

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

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

Hello (Fortran)

```fortran
program main
    include 'mpif.h'
    integer ierr, rank, size
     call MPI_INIT( ierr )
    call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
    call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
    print *, 'I am ', rank, ' of ', size
    call MPI_FINALIZE( ierr )
end
```

Hello (C++)

```cpp
#include "mpi.h"
#include <iostream>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI::Init(argc, argv);
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();
    std::cout << "I am " << rank << " of " << size << "\n";
    MPI::Finalize();
    return 0;
}
```
Notes on Hello World

- All MPI programs begin with `MPI_Init` and end with `MPI_Finalize`
- `MPI_COMM_WORLD` is defined by `mpi.h` (in C) or `mpif.h` (in Fortran) and designates all processes in the MPI “job”
- Each statement executes independently in each process - including the `printf/print` statements
- I/O not part of MPI-1 but is in MPI-2
  - print and write to standard output or error not part of either MPI-1 or MPI-2
  - output order is undefined (may be interleaved by character, line, or blocks of characters)
- The MPI-1 Standard does not specify how to run an MPI program, but many implementations provide

```
mpirun -np 4 a.out
```

MPI Basic Send/Receive

- We need to fill in the details in

```
Process 0

Send(data) ~Process 1

Receive(data)
```

- 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?

Some Basic Concepts

- Processes can be collected into groups
- Each message is sent in a context, and must be received in the same context
  - Provides necessary support for libraries
- 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`

General MPI Communication Terms

- Point-to-point communication
  - Sender and receiver processes are explicitly named. A message is sent from a specific sending process (point a) to a specific receiving process (point b).
- Collective communication
  - Higher-level communication operations that involve multiple processes
  - Examples: Broadcast, scatter/gather
**MPI Datatypes**

- The data in a message to send or receive 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)
    - 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, in particular ones for subarrays.

**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.

**MPI Basic (Blocking) Send**

```c
MPI_SEND(start, count, datatype, dest, tag, comm)
```

- The message buffer is described by (start, count, datatype).
- The target process is specified by dest, which is the rank of the target process in the communicator specified by comm.
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

**MPI Basic (Blocking) Receive**

```c
MPI_RECV(start, count, datatype, source, tag, comm, status)
```

- Waits until a matching (both source and tag) message is received from the system, and the buffer can be used.
- source is rank in communicator specified by comm, or MPI_ANY_SOURCE.
- tag is a tag to be matched on or MPI_ANY_TAG.
- receiving fewer than count occurrences of datatype is OK, but receiving more is an error.
- status contains further information (e.g., size of message).
A Simple MPI Program

```c
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank, buf;
    MPI_Status status;
    MPI_Init(argc, argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    /* Process 0 sends and Process 1 receives */
    if (rank == 0) {
        buf = 123456;
        MPI_Send(&buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(&buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        printf("Received %d\n", buf);
    }
    MPI_Finalize();
    return 0;
}
```

Figure 7.1 An MPI solution to the Count 3s problem.

Figure 7.1 An MPI solution to the Count 3s problem. (cont.)

Code Spec 7.8 MPI Scatter().

```c
MPI_Scatter();
int MPI_Scatter(); // Scatter routine
void *sendbuf[], // Address of the data to send
int sendcount, // Number of data elements to send
MPI_Datatype sendtype, // Type of data elements to send
void *recvbuf[], // Address of buffer to receive data
int recvcount, // Number of data elements to receive
MPI_Datatype recvtype, // Type of data elements to receive
int MPI_COMM_V [nproc, // Rank of the root process
    MPI_Comm root], // MPI root communicator

Argumens:
* The first three arguments specify the address, size, and type of the data elements to send to each process. These arguments only have meaning for the root process.
* The second four arguments specify the address, size, and type of the data elements to receive from each process. The size and type of the existing data and the receiving data may differ as a means of converting data types.
**Code Spec 7.8 MPI Scatter() \(\text{(cont.)}\)**

- The seventh argument specifies the root process that is the source of the data.
- The eighth argument specifies the MPI communicator to use.

**Notes:**

This routine distributes data from the root process to all other processes, including the root. A more sophisticated version of the routine, `MPI_Shuffle()`, allows the root process to send different amounts of data to the various processes. Details can be found in the MPI standard.

Return value:

An MPI error code.

---

**Figure 7.2 Replacement code (for lines 16-48 of Figure 7.1) to distribute data using a scatter operation.**

```c
16 length_per_process = length/size;
17 my_rank = MPI.COMM_RANK(MPI_COMM_WORLD, proc_rank);
18 my_size = MPI.COMM_SIZE(MPI_COMM_WORLD);
19 if (my_rank == root_rank)
20 { for (i=0; i<length; i++)
21 \quad temp[i] = data[i];
22 }
23 else
24 \quad size = length_per_process;
25 \quad buf = &data[my_rank * length_per_process];
26 \quad mpi_data = (char *)malloc(sizeof(char) * size);
27 \quad for (i=0; i<length_per_process; i++)
28 \quad mpi_data[i] = temp[i];
29 \quad mpi_size = size;
30 \quad \text{MPI_Scatter} (temp, mpi_data, mpi_size, MPI_CHAR, root_rank, size, size);
31 \quad \text{free(mpi_data)};
32 return 0;
```

---

**Other Basic Features of MPI**

- **MPI_Gather**
  - Analogous to MPI_Scatter
- **Scans and reductions**
- **Groups, communicators, tags**
  - Mechanisms for identifying which processes participate in a communication
- **MPI_Bcast**
  - Broadcast to all other processes in a "group"

---

**Figure 7.4 Example of collective communication within a group.**

```c
1 \text{void broadcast();}
2 \text{/* initialize all processes */}
3 \text{MPI_Status *status;}
4 \text{MPI_Group group;}
5 \text{MPI_Comm comm;}
6 \text{int rank, comm_rank, size, num_groups;}
7 \text{MPI_Bcast(comm, size, MPI_CHAR, 0, status);}
8 \text{MPI_Comm_get_size(comm, &size);}
9 \text{MPI_Group_get_size(group, &num_groups);}
10 \text{MPI_Group_create(comm, group);}
11 \text{MPI_Group_translate(group, &group);}
12 \text{MPI_Group_incl(group, 1, &rank, status);}
13 \text{MPI_Group_translate(group, &group);}
14 \text{MPI_Bcast(comm, size, MPI_CHAR, 0, status);}
15 \text{MPI_Group_free(group);}
```

---
Figure 7.5 A 2D relaxation replaces—on each iteration—all interior values by the average of their four nearest neighbors.

Sequential code:
for (i=1; i<n-1; i++)
for (j=1; j<n-1; j++)
b[i][j] = (a[i-1][j]+a[i][j-1]+a[i+1][j]+a[i][j+1])/4.0;

Figure 7.6 MPI code for the main loop of the 2D SOR computation.

MPI code for the main loop of the 2D SOR computation. (cont.)

Figure 7.6 MPI code for the main loop of the 2D SOR computation. (cont.)
MPI Critique (Snyder)

- Message passing is a very simple model
- Extremely low level; heavy weight
  - Expense comes from $\lambda$ and lots of local code
  - Communication code is often more than half
  - Tough to make adaptable and flexible
  - Tough to get right and know it
  - Tough to make perform in some (Snyder says most) cases
- Programming model of choice for scalability
- Widespread adoption due to portability, although not completely true in practice

Next Time

- More detail on communication constructs
  - Blocking vs. non-blocking
  - One-sided communication
- Support for data and task parallelism