Acknowledgments for Today’s Lecture

• “Principles of Parallel Programming”, Calvin Lin & Lawrence Snyder
  — Includes resources available at http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html

• “Parallel Architectures”, Calvin Lin
  — Lectures 5 & 6, CS380P, Spring 2009, UT Austin


• MPI slides from “High Performance Computing: Models, Methods and Means”, Thomas Sterling, CSC 7600, Spring 2009, LSU
  — http://www.cct.lsu.edu/csc7600/coursemat/index.html

• mpiJava home page: http://www.hpjava.org/mpiJava.html

• MPI lectures given at Rice HPC Summer Institute 2009, Tim Warburton, May 2009
Block Distribution

- `dist.factory.block([lo:hi])` creates a block distribution over the one-dimensional region, `lo:hi`.

- A block distribution splits the region into contiguous subregions, one per place, while trying to keep the subregions as close to equal in size as possible.

- Block distributions can improve the performance of parallel loops that exhibit spatial locality across contiguous iterations.

- Example: `dist.factory.block([0:15])` for 4 places

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place id</td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cyclic Distribution

- `dist.factory.cyclic([lo:hi])` creates a cyclic distribution over the one-dimensional region, lo:hi.
- A cyclic distribution “cycles” through places 0 … place.MAX PLACES – 1 when spanning the input region.
- Cyclic distributions can improve the performance of parallel loops that exhibit load imbalance.
- Example: `dist.factory.cyclic([0:15])` for 4 places

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place id</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

- Example: `dist.factory.cyclic([0:7,0:1])` for 4 places

<table>
<thead>
<tr>
<th>Index</th>
<th>[0,0]</th>
<th>[0,1]</th>
<th>[1,0]</th>
<th>[1,1]</th>
<th>[2,0]</th>
<th>[2,1]</th>
<th>[3,0]</th>
<th>[3,1]</th>
<th>[4,0]</th>
<th>[4,1]</th>
<th>[5,0]</th>
<th>[5,1]</th>
<th>[6,0]</th>
<th>[6,1]</th>
<th>[7,0]</th>
<th>[7,1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place id</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Worksheet #30: impact of distribution on parallel completion time

1. public void sampleKernel(int iterations,
2. int numChunks, dist d) {
3.     for (int iter = 0; iter < iterations; iter++) {
4.         finish for (point [jj] : [0:numChunks-1])
5.             async at(d.get(jj)) {
6.                 perf.doWork(jj);
7.             } // Assume that time to process chunk jj = jj units
8.         } // finish-for-async
9.         double[] temp = myNew; myNew = myVal; myVal = temp;
10.     } // for iter
11. } // sample kernel

• Assume an execution with n places using the option, -places n:1
• Will a block or cyclic distribution for d have a smaller abstract completion time, assuming that all tasks on the same place are serialized?

Answer: cyclic distribution because it leads to better load balance (locality was not a consideration in this problem)
Consider a pipeline of actors where an item is produced in each actor and then transferred between actors using messages. Would a block or cyclic assignment of actors to places have better data locality?

- Example with 4 places:
  - Block Distribution:
    - Place 0: A-0...A-4;
    - Place 1: A-5...A-9, ...
  - Cyclic Distribution:
    - Place 0: A-0, A-5, A-10, A-15;
    - Place-1: A-1, A-6, A-11, A16, ...

Answer: block distribution because it leads to better locality
Distributed Parallel Loops

- The listing below shows the typical pattern used to iterate over an input region \( r \), while creating one async task for each iteration \( p \) at the place dictated by distribution \( d \) i.e., at place \( d.get(p) \).

- This pattern works correctly regardless of the rank and contents of input region \( r \) and input distribution \( d \) i.e., it is not constrained to block distributions.

```plaintext
finish {
  region r = ...; // e.g., [0:15] or [0:7,0:1]
  dist d = dist.factory.block(r);
  for (point p:r)
    async at(d.get(p)) {
      // Execute iteration p at place specified by distribution d
      ...
    }
} // finish
.
.
```
Chunked Fork-Join Iterative Averaging Example with Places

1. public void runDistChunkedForkJoin(int iterations,
2. int numChunks, dist d) {
3. for (int iter = 0; iter < iterations; iter++) {
4.     finish for (point [jj] : [0:numChunks-1])
5.     async at(d.get(jj)) {
6.         for (point [j] : getChunk([1:n],numChunks,jj))
7.             myNew[j] = (myVal[j-1] + myVal[j+1]) / 2.0;
8.     } // finish-for-async
9.     double[] temp = myNew; myNew = myVal; myVal = temp;
10. } // for iter
11. } // runDistChunkedForkJoin

• Chunk jj is always executed in the same place for each iter
• Method runDistChunkedForkJoin can be called with different values of distribution parameter d
Analyzing Locality of Fork-Join Iterative Averaging Example with Block & Cyclic Distributions

Both Block and Cyclic distributions show locality benefits --- Block distribution may be better due to spatial locality as well.

Locality benefits will be realized if all instances of chunk 0 execute on the same core and reuse data from the same cache.

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place id</td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
<td>4</td>
<td></td>
<td>5</td>
<td></td>
<td>6</td>
<td></td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place id</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Cyclic distribution for a 8×8 sized region (e.g., [1:8,1:8]) mapped on to 5 places

Figure source: “Principles of Parallel Programming”, Calvin Lin & Lawrence Snyder, http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html
Block-Cyclic Distribution

- `dist.factory.blockCyclic([lo:hi], b)` creates a block-cyclic distribution over the one-dimensional region, `lo:hi`.

- A block-cyclic distribution combines the locality benefits of the block distribution with the load-balancing benefits of the cyclic distribution by introducing a block size parameter, `b`.

- The linearized region is first decomposed into contiguous blocks of size `b`, and then the blocks are distributed in a cyclic manner across the places.

- Example in Table 5: `dist.factory.blockCyclic([0:15], 2)` for 4 place with block size `b = 2`

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place id</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Organization of a Shared-Memory Multicore Symmetric Multiprocessor (SMP)

- Memory hierarchy for a single Intel Xeon Quad-core E5440 HarperTown processor chip
  - A SUG@R node contains TWO such chips, for a total of 8 cores
Organization of a Distributed-Memory Multiprocessor

Figure (a)

- Host node (Pc) connected to a cluster of processor nodes (P₀ … Pₘ)
- Processors P₀ … Pₘ communicate via a dedicated high-performance interconnection network (e.g., Infiniband)
  —Supports much lower latencies and higher bandwidth than standard TCP/IP networks

Figure (b)

- Each processor node consists of a processor, memory, and a Network Interface Card (NIC) connected to a router node (R) in the interconnect
Principles of Message-Passing Programming

- The logical view of a machine supporting the message-passing paradigm consists of \( p \) processes, each with its own exclusive address space.
  1. Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.
  2. All interactions (read-only or read/write) require cooperation of two processes - the process that has the data and the process that wants to access the data.

- These two constraints, while onerous, make underlying costs very explicit to the programmer.

- In this loosely synchronous model, processes synchronize infrequently to perform interactions. Between these interactions, they execute completely asynchronously.

- Most message-passing programs are written using the single program multiple data (SPMD) model.
SPMD Pattern

- **SPMD:** Single Program Multiple Data  
- Run the same program on P processing elements (PEs)  
- Use the “rank” ... an ID ranging from 0 to (P-1) ... to determine what computation is performed on what data by a given PE  
- Different PEs can follow different paths through the same code  
- Convenient pattern for hardware platforms that are not amenable to efficient forms of dynamic task parallelism  
  —General-Purpose Graphics Processing Units (GPGPUs)  
  —Distributed-memory parallel machines  
- Key design decisions --- how should data and computation be distributed across PEs?
Using the SPMD model with a Global View of Data: Iterative Averaging

1. double[] gVal = new double[n+2]; double[] gNew = new double[n+2];
2. gVal[n+1] = 1; // Boundary condition
3. int Cj = Runtime.getNumOfWorkers();
4. forall (point [jj]:[0:Cj-1]) { // SPMD computation with “id” = jj
   5. double[] myVal = gVal; double[] myNew = gNew; // Local copy
   6. for (point [iter] : [0:numIters-1]) {
      7. // Compute MyNew as function of input array MyVal
      8. for (point [j]:getChunk([1:n],[Cj],[jj]))
      9.   myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
     10. next; // Barrier before executing next iteration of iter loop
   11. // Swap myVal and myNew (replicated computation)
   12. double[] temp = myVal; myVal = myNew; myNew = temp;
   13. // myNew becomes input array for next iter
   14. } // for
5. } // forall

Data Distribution: Local View in Distributed-Memory Systems

Distributed memory
- Each process sees a local address space
- Processes send messages to communicate with other processes

Data structures
- Presents a Local View instead of Global View
- Programmer must make the mapping

![Diagram of Global and Local View](image)

Global View
Local View (4 processes)
Using the SPMD model with a Local View

SPMD code
- Write one piece of code that executes on each processor

Processors must communicate via messages for non-local data accesses
- Similar to communication constraint for actors (except that we allowed hybrid combinations of global task parallelism and local actor parallelism in HJ)
MPI: The Message Passing Interface

- Sockets and Remote Method Invocation (RMI) are communication primitives used for distributed Java programs.
  - Designed for standard TCP/IP networks rather than high-performance interconnects

- The Message Passing Interface (MPI) standard was designed to exploit high-performance interconnects
  - MPI was standardized in the early 1990s by the MPI Forum—a substantial consortium of vendors and researchers
  - It is an API for communication between nodes of a distributed memory parallel computer
  - The original standard defines bindings to C and Fortran (later C++)
    - Java support is available from a research project, mpiJava, developed at Indiana University 10+ years ago
      - [http://www.hpjava.org/mpiJava.html](http://www.hpjava.org/mpiJava.html)
Features of MPI

- MPI is a platform for Single Program Multiple Data (SPMD) parallel computing on distributed memory architectures, with an API for sending and receiving messages.
- It includes the abstraction of a “communicator”, which is like an N-way communication channel that connects a set of N cooperating processes (analogous to a phaser).
- It also includes explicit datatypes in the API, that are used to describe the contents of communication buffers.
The Minimal Set of MPI Routines (mpiJava)

- MPI.Init(args)
  - initialize MPI in each process
- MPI.Finalize()
  - terminate MPI
- MPI.COMM_WORLD.Size()
  - number of processes in COMM_WORLD communicator
- MPI.COMM_WORLD.Rank()
  - rank of this process in COMM_WORLD communicator

Note:
- In this subset, processes act independently with no information communicated among the processes.
- “embarrassingly parallel”, Cleve Moler.
Our First MPI Program (mpiJava version)

1. import mpi.*;
2. class Hello {
3.     static public void main(String[] args) {
4.         // Init() be called before other MPI calls
5.         MPI.Init(args); /
6.         int npes = MPI.COMM_WORLD.Size();
7.         int myrank = MPI.COMM_WORLD.Rank();
8.         System.out.println("My process number is " + myrank);
9.         MPI.Finalize(); // Shutdown and clean-up
10.     }
11. }

main() is enclosed in an implicit “forall” --- each process runs a separate instance of main() with “index variable” = myrank
MPI Communicators

- Communicator is an internal object
  - *Communicator registration is like phaser registration, except that MPI does not support dynamic parallelism*

- MPI programs are made up of communicating processes

- Each process has its own address space containing its own attributes such as rank, size (and argc, argv, etc.)

- MPI provides functions to interact with it

- Default communicator is MPI.COMM_WORLD
  - All processes are its members
  - It has a size (the number of processes)
  - Each process has a rank within it
  - Can think of it as an ordered list of processes

- Additional communicator(s) can co-exist

- A process can belong to more than one communicator

- Within a communicator, each process has a unique rank
Adding Send() and Recv() to the Minimal Set of MPI Routines (mpiJava)

- **MPI.Init(args)**
  - initialzie MPI in each process
- **MPI.Finalize()**
  - terminate MPI
- **MPI.COMM_WORLD.Size()**
  - number of processes in COMM_WORLD communicator
- **MPI.COMM_WORLD.Rank()**
  - rank of this process in COMM_WORLD communicator
- **MPI.COMM_WORLD.Send()**
  - send message using COMM_WORLD communicator
- **MPI.COMM_WORLD.Recv()**
  - receive message using COMM_WORLD communicator
A very simple communication between two processes is:
—process zero sends ten doubles to process one

In MPI this is a little more complicated than you might expect.

Process zero has to tell MPI:
—to send a message to process one
—that the message contains ten entries
—the entries of the message are of type double
—the message has to be tagged with a label (integer number)

Process one has to tell MPI:
—to receive a message from process zero
—that the message contains ten entries
—the entries of the message are of type double
—the label that process zero attached to the message
mpiJava Class hierarchy

package mpi

MPI
  Group
  Comm
  Datatype
  Status
  Request

Intracomm
  Intercomm
  Cartcomm
  Graphcomm

Prequest
mpiJava send and receive

- Send and receive members of Comm:
  
  ```java
  void Send(Object buf, int offset, int count, Datatype type, int dst, int tag) ;
  Status Recv(Object buf, int offset, int count, Datatype type, int src, int tag) ;
  ```

- The arguments buf, offset, count, type describe the data buffer—the storage of the data that is sent or received. They will be discussed on the next slide.

- dst is the rank of the destination process relative to this communicator. Similarly in Recv(), src is the rank of the source process.

- An arbitrarily chosen tag value can be used in Recv() to select between several incoming messages: the call will wait until a message sent with a matching tag value arrives.

- The Recv() method returns a Status value, discussed later.

- Both Send() and Recv() are **blocking** operations by default—Analogous to a phaser next operation
Example of Send andRecv

1. import mpi.*;

3. class myProg {
4.     public static void main( String[] args ) {
5.         int tag0 = 0;
6.         MPI.Init( args ); // Start MPI computation
7.         if ( MPI.COMM_WORLD.rank() == 0 ) { // rank 0 = sender
8.             int loop[] = new int[1]; loop[0] = 3;
9.             MPI.COMM_WORLD.Send( "Hello World!", 0, 12, MPI.CHAR, 1, tag0 );
10.            MPI.COMM_WORLD.Send( loop, 0, 1, MPI.INT, 1, tag0 );
11.         } else { // rank 1 = receiver
12.             int loop[] = new int[1]; char msg[] = new char[12];
13.             MPI.COMM_WORLD.Recv( msg, 0, 12, MPI.CHAR, 0, tag0 );
14.             MPI.COMM_WORLD.Recv( loop, 0, 1, MPI.INT, 0, tag0 );
15.             for ( int i = 0; i < loop[0]; i++ ) System.out.println( msg );
16.         }
17.         MPI.Finalize( ); // Finish MPI computation
18.     }
19. }

Send() andRecv() calls are blocking operations by default
Worksheet #32: MPI send and receive

1. int a[], b[];
2. ...
3. if (MPI.COMM_WORLD.rank() == 0) {
4.   MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, 1, 1);
5.   MPI.COMM_WORLD.Send(b, 0, 10, MPI.INT, 1, 2);
6. }
7. else {
8.   Status s2 = MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, 0, 2);
9.   Status s1 = MPI.COMM_WORLD.Recv(a, 0, 10, MPI.INT, 0, 1);
10.  System.out.println("a = " + a + " ; b = " + b);
11.}
12. ...

In the space below, indicate what values you expect the print statement in line 10 to output.

Name 1: ___________________          Name 2: ___________________