CS 181E: Fundamentals of Parallel Programming

Instructor: Vivek Sarkar
Co-Instructor: Ran Libeskind-Hadas

http://www.cs.hmc.edu/courses/2012/fall/cs181e/
Recap of Lecture 10

• Linearizability of Concurrent Executions and Concurrent Objects

• Liveness/progress guarantees

• Optimized Implementations of Isolated
**Worksheet #10 solution:**
Linearizability of method calls on a concurrent object

**Is this a linearizable execution?**

<table>
<thead>
<tr>
<th>Time</th>
<th>Task A</th>
<th>Task B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Invoke q.enq(x)</td>
<td>Invoke q.enq(y)</td>
</tr>
<tr>
<td>1</td>
<td>Return from q.enq(x)</td>
<td>Work on q.enq(y)</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Return from q.enq(y)</td>
</tr>
<tr>
<td>3</td>
<td>Invoke q.deq()</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Work on q.deq()</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Return y from q.deq()</td>
<td></td>
</tr>
</tbody>
</table>

No! q.enq(x) must precede q.enq(y) in all linear sequences of method calls invoked on q. It is illegal for the q.deq() operation to return y.
Outline

- Task Affinity with Places

- Introduction to the Message Passing Interface (MPI)
An example Memory Hierarchy --- what is the cost of a Memory Access?

- **L0**: Registers
- **L1**: L1 cache (Static RAM)
- **L2**: L2 cache (Static RAM)
- **L3**: Main memory (Dynamic RAM)
- **L4**: Local secondary storage (local disks)
- **L5**: Remote secondary storage (tapes, distributed file systems, Web servers)

**Sources:**
- CPU registers hold words retrieved from L1 cache
- L1 cache holds cache lines retrieved from L2 cache
- L2 cache holds cache lines retrieved from main memory
- Main memory holds disk blocks retrieved retrieved from local disks
- Local disks hold files retrieved from disks on remote network servers

# Storage Trends

## SRAM

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$/MB</td>
<td>19,200</td>
<td>2,900</td>
<td>320</td>
<td>256</td>
<td>100</td>
<td>75</td>
<td>60</td>
<td>320</td>
</tr>
<tr>
<td>access (ns)</td>
<td>300</td>
<td>150</td>
<td>35</td>
<td>15</td>
<td>3</td>
<td>2</td>
<td>1.5</td>
<td>200</td>
</tr>
</tbody>
</table>

## DRAM

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$/MB</td>
<td>8,000</td>
<td>880</td>
<td>100</td>
<td>30</td>
<td>1</td>
<td>0.1</td>
<td>0.06</td>
<td>130,000</td>
</tr>
<tr>
<td>access (ms)</td>
<td>375</td>
<td>200</td>
<td>100</td>
<td>70</td>
<td>60</td>
<td>50</td>
<td>40</td>
<td>9</td>
</tr>
<tr>
<td>typical size (MB)</td>
<td>0.064</td>
<td>0.256</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>2,000</td>
<td>8,000</td>
<td>125,000</td>
</tr>
</tbody>
</table>

## Disk

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$/MB</td>
<td>500</td>
<td>100</td>
<td>8</td>
<td>0.30</td>
<td>0.01</td>
<td>0.005</td>
<td>0.0003</td>
<td>1,600,000</td>
</tr>
<tr>
<td>access (ms)</td>
<td>87</td>
<td>75</td>
<td>28</td>
<td>10</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>29</td>
</tr>
<tr>
<td>typical size (MB)</td>
<td>1</td>
<td>10</td>
<td>160</td>
<td>1,000</td>
<td>20,000</td>
<td>160,000</td>
<td>1,500,000</td>
<td>1,500,000</td>
</tr>
</tbody>
</table>

Cache Memories

• **Cache memories** are small, fast SRAM-based memories managed automatically in hardware.
  — Hold frequently accessed blocks of main memory

• **CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory.**

• **Typical system structure:**

Examples of Caching in the Hierarchy

Ideally one would desire an indefinitely large memory capacity such that any particular word would be immediately available. ... We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

**A. W. Burks, H. H. Goldstine, and J. von Neumann**

*Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (1946)*

---

**Ultimate goal:** create a large pool of storage with average cost per byte that approaches that of the cheap storage near the bottom of the hierarchy, and average latency that approaches that of fast storage near the top of the hierarchy.
Locality

• Principle of Locality:

  — Empirical observation: Programs tend to use data and instructions with addresses near or equal to those they have used recently

• Temporal locality:

  — Recently referenced items are likely to be referenced again in the near future

• Spatial locality:

  — Items with nearby addresses tend to be referenced close together in time

  — A Java programmer can only influence spatial locality at the intra-object level
    
    - The garbage collector and memory management system determines inter-object placement

Source: http://www.cs.cmu.edu/afs/cs/academic/class/15213-f10/www/lectures/09-memory-hierarchy.pptx
Locality Example

• Data references
  — Reference array elements in succession (stride-1 reference pattern).  Spatial locality
  — Reference variable sum each iteration.  Temporal locality

• Instruction references
  — Reference instructions in sequence.  Spatial locality
  — Cycle through loop repeatedly.  Temporal locality

sum = 0;
for (i = 0; i < n; i++)
  sum += a[i];
return sum;

Source: http://www.cs.cmu.edu/afs/cs/academic/class/15213-f10/www/lectures/09-memory-hierarchy.pptx
Memory Hierarchy in a Multicore Processor

- Memory hierarchy for a single Intel Xeon Quad-core E5440 HarperTown processor chip
Programmer Control of Task Assignment to Processors

• The parallel programming constructs that we’ve studied thus far result in tasks that are assigned to processors dynamically by the HJ runtime system
  — Programmer does not worry about task assignment details

• Sometimes, programmer control of task assignment can lead to significant performance advantages due to improved locality

• Motivation for HJ “places”
  — Provide the programmer a mechanism to map each task to a set of processors when the task is created
HJ programmer defines mapping from HJ tasks to set of places

HJ runtime defines mapping from places to one or more worker Java threads per place

The option “-places p:w” when executing an HJ program can be used to specify
- \( p \), the number of places
- \( w \), the number of worker threads per place
Example of places 4:2 option on an 8-core node (4 places w/ 2 workers per place)
Places in HJ

\textbf{here} = place at which current task is executing

\textbf{place.MAX\_PLACES} = total number of places (runtime constant)

\hspace{1cm} Specified by value of \texttt{p} in runtime option, \texttt{-places \ p:w}

\textbf{place.factory.place(i)} = place corresponding to index \texttt{i}

\texttt{<place-expr>.toString()} returns a string of the form “place(id=0)”

\texttt{<place-expr>.id} returns the id of the place as an \texttt{int}

\texttt{async at(P) S}

\hspace{1cm} Creates new task to execute statement \texttt{S} at place \texttt{P}

\hspace{1cm} \texttt{async S} is equivalent to \texttt{async at(here) S}

\hspace{1cm} Main program task starts at \texttt{place.factory.place(0)}

Note that \texttt{here} in a child task refers to the place \texttt{P} at which the child task is executing, not the place where the parent task is executing
Example of –places 4:2 option on an 8-core node (4 places w/ 2 workers per place)

// Main program starts at place 0
async at(place.factory.place(0)) S1;
async at(place.factory.place(0)) S2;
async at(place.factory.place(1)) S3;
async at(place.factory.place(1)) S4;
async at(place.factory.place(1)) S5;
async at(place.factory.place(2)) S6;
async at(place.factory.place(2)) S7;
async at(place.factory.place(2)) S8;
async at(place.factory.place(3)) S9;
async at(place.factory.place(3)) S10;
Example of –places 1:8 option (1 place w/ 8 workers per place)

All async’s run at place 0 when there’s only one place!
Example HJ program with places

```java
class T1 {
    final place affinity;
    . . .
    // T1's constructor sets affinity to place where instance was created
    T1() { affinity = here; . . . }
    . . .
}
. . .
finish { // Inter-place parallelism
    System.out.println("Parent.place.==", here); // Parent task's place
    for (T1 a = . . .) {
        async at (a.affinity) { // Execute async at place with affinity to a
            a.foo();
            System.out.println("Child.place.==", here); // Child task's place
        } // async
    } // for
} // finish
. . .
```
Distributions --- hj.lang.dist

• A distribution maps points in a rectangular index space (region) to places e.g.,
  — \( i \rightarrow \text{place.factory.place}(i \mod \text{place.MAX PLACES}-1) \)

• Programmers are free to create any data structure they choose to store and compute these mappings

• For convenience, the HJ language provides a predefined type, hj.lang.dist, to simplify working with distributions

• Some public members available in an instance \( d \) of hj.lang.dist are:
  — \( d.\text{rank} \) = number of dimensions in the input region for distribution \( d \)
  — \( d.\text{get}(p) \) = place for point \( p \) mapped by distribution \( d \). It is an error to call \( d.\text{get}(p) \) if \( p.\text{rank} \neq d.\text{rank} \).
  — \( d.\text{places()} \) = set of places in the range of distribution \( d \)
  — \( d.\text{restrictToRegion}(pl) \) = region of points mapped to place \( pl \) by distribution \( d \)
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.

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Block Distribution (contd)

- If the input region is multidimensional, then a block distribution is computed over the linearized one-dimensional version of the multidimensional region.

- Example in Table 2: `dist.factory.block([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>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>
Distributed Parallel Loops

• Listing 2 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.

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

Chunk jj is always executed in the same place for each iteration,

Let's try another example of a distributed parallel loop in Worksheet 11!
Analyzing Locality of Fork-Join Iterative Averaging Example with Places

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>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>
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>
Outline

• Task Affinity with Places

• Introduction to the Message Passing Interface (MPI)
Organization of a Distributed-Memory Multiprocessor

Figure (a)
- Host node (Pc) connected to a cluster of processor nodes ($P_0 ... P_m$)
- Processors $P_0 ... P_m$ 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?
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
   double[] myVal = gVal; double[] myNew = gNew; // Local copy
   for (point [iter] : [0:numIters-1]) {
      // Compute MyNew as function of input array MyVal
      for (point [j]:getChunk([1:n],[Cj],[jj]))
         myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
      next; // Barrier before executing next iteration of iter loop
   }
   // Swap myVal and myNew (replicated computation)
   double[] temp=myVal; myVal=myNew; myNew=temp;
   // myNew becomes input array for next iter
} // forall
} // for
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 showing global view and local view of data distribution in distributed-memory systems.](image)
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.
main() is enclosed in an implicit “forall” --- each process runs a separate instance of main() with “index variable” = myrank

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. }

Our First MPI Program (mpiJava version)
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)**
  - 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

- **MPI.COMM_WORLD.Send()**
  - send message using COMM_WORLD communicator

- **MPI.COMM_WORLD.Recv()**
  - receive message using COMM_WORLD communicator
Example with `Send()` and `Recv()` calls

```java
import mpi.*;

class myProg {
    public static void main( String[] args ) {
        int tag0 = 0;
        MPI.Init( args );  // Start MPI computation
        if ( MPI.COMM_WORLD.rank() == 0 ) {  // rank 0 = sender
            int loop[] = new int[1]; loop[0] = 3;
            MPI.COMM_WORLD.Send( "Hello World!", 0, 12, MPI.CHAR, 1, tag0 );
            MPI.COMM_WORLD.Send( loop, 0, 1, MPI.INT, 1, tag0 );
        } else {  // rank 1 = receiver
            int loop[] = new int[1]; char msg[] = new char[12];
            MPI.COMM_WORLD.Recv( msg, 0, 12, MPI.CHAR, 0, tag0 );
            MPI.COMM_WORLD.Recv( loop, 0, 1, MPI.INT, 0, tag0 );
            for ( int i = 0; i < loop[0]; i++ ) System.out.println( msg );
        }
        MPI.Finalize( );  // Finish MPI computation
    }
}
```

Send() and Recv() calls are blocking operations by default
Worksheet #11 (to be done individually or in pairs): impact of distribution on parallel completion time

1. public void sampleKernel(int iterations, int numChunks, dist d) {
2.     for (int iter = 0; iter < iterations; iter++) {
3.         finish for (point [jj] : [0:numChunks-1])
4.             async at(d.get(jj)) {
5.                 perf.addLocalOps(jj);
6.                 // Assume that time to process chunk jj is O(jj)
7.             } // finish-for-async
8.         } // for iter
9.     } // 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 parallel completion time, assuming that all tasks on the same place are serialized?