COMP 322: Fundamentals of Parallel Programming

Lecture 33: Introduction to the Message Passing Interface (MPI)

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Announcements

- **Homework 4**
    - —Deadline extended to 11:59pm on Sunday, April 12th
    - —You are allowed to modify any class that you choose, but you must clearly state in the report which classes you modified.
    - — There will be no deductions for poor performance or poor actual speedup in this homework. You still need to run your solution on STIC and include the results in your report. If you have a submission that demonstrates parallel speedup on STIC, that's great. If not, the staff will evaluate the algorithmic approach in the parallel version of your code, instead of evaluating the actual parallel speedup.

- **Homework 5**
    - —Will be assigned on April 13th and due on April 24th (with an automatic 1-week extension to May 1st)
    - —Main homework will consist of written questions only
    - —There will be an optional programming assignment for which you can get extra credit

Worksheet #32 solution: Spark and Map-Reduce

```
val points = spark.textFile(…).map(parsePoint).cache()

var w = Vector.random(D) // current separating plane

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(doWork(1)).reduce(_ + _)

  w -= gradient
}

println("Final separating plane: " + w)
```

There are ITERATIONS sequential iterations, each mapping doWork in parallel over every value in points, which is immediately forced by a reduce. So,
work = ITERATIONS * |points|
CPL = ITERATIONS.

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
  - http://www.cs.utexas.edu/users/lin/cs380p/schedule.html

- Slides accompanying Chapter 6 of "Introduction to Parallel Computing", 2nd Edition, Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar, Addison-Wesley, 2003
  - http://www-users.cs.umn.edu/~karypis/parbook/Lectures/AG/chap6_slides.pdf

- 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

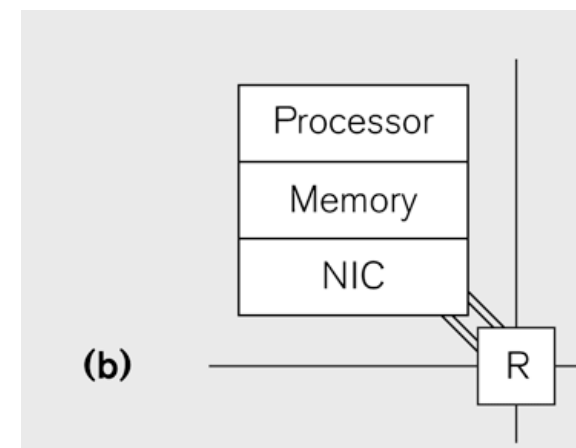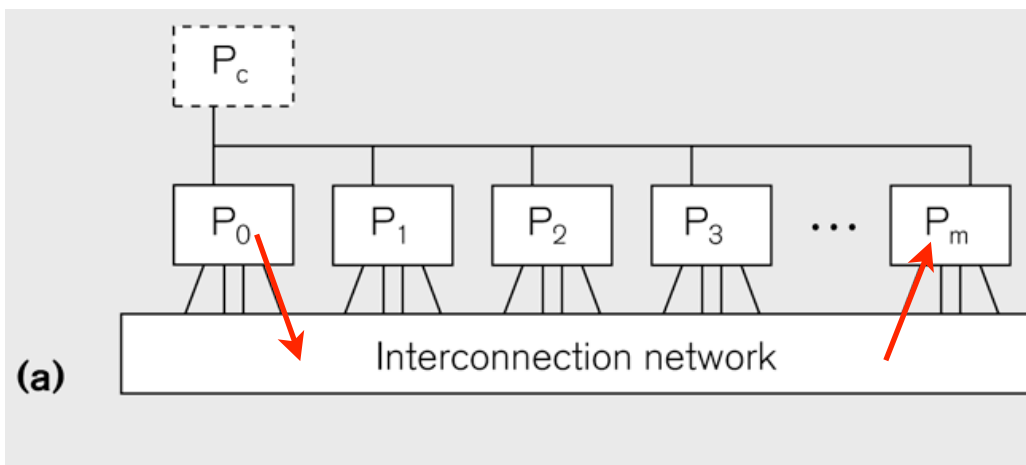# Organization of a Distributed-Memory Multiprocessor

Figure (a)

- Host node (Pc) connected to a cluster of processor nodes ($P_0 \ldots P_m$)

- Processors $P_0 \ldots P_m$ communicate via an interconnection network which could be standard TCP/IP (e.g., for Map-Reduce) or specialized for high performance communication (e.g., for scientific computing)
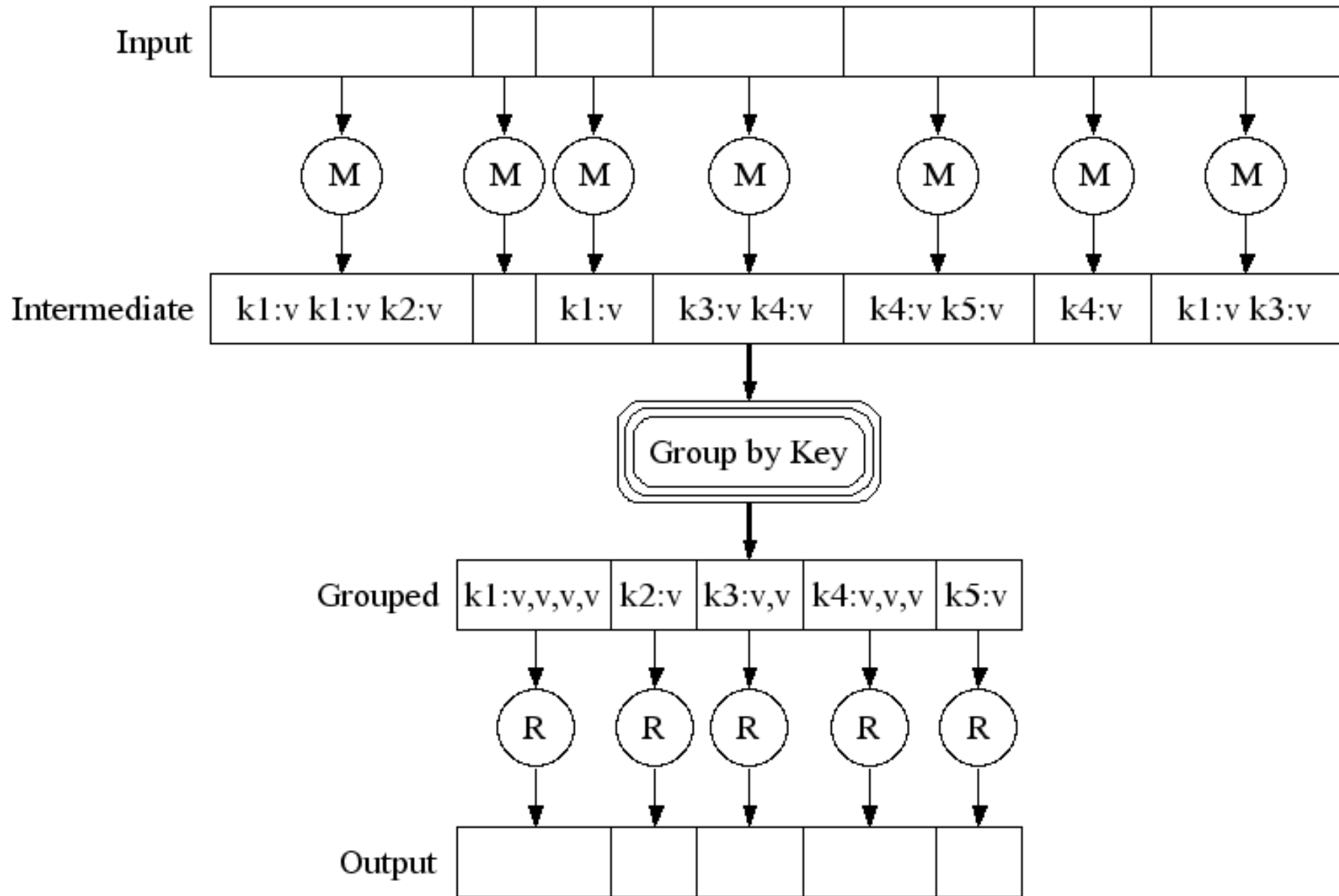
Figure (b)

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

Processors communicate by sending messages via an interconnect

MapReduce Execution: One Approach to Programming Distributed-Memory Multiprocessors

Another approach: Message-Passing (extension of Actor model)

- The logical view of a machine supporting the message-passing paradigm consists of *p processes*, each with its own exclusive address space, that are capable of executing on different nodes in a distributed-memory multiprocessor

  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 ("bulk synchronous") model, processes synchronize infrequently to perform interactions. Between these interactions, they execute completely asynchronously.
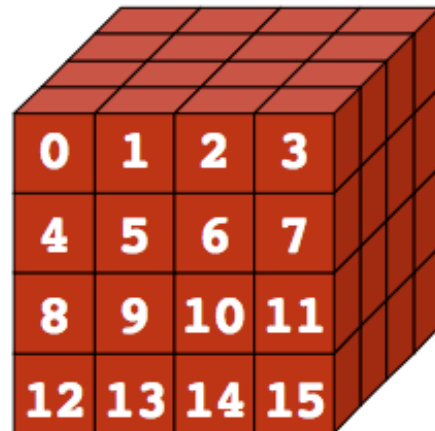
Data Distribution: Local View in Distributed-Memory Systems
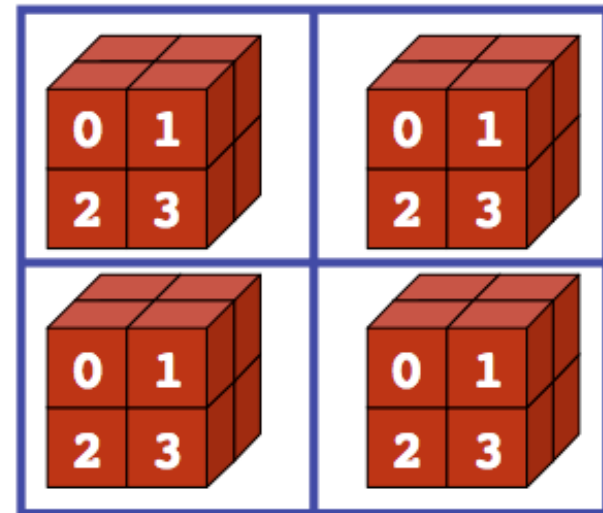
Distributed memory

– 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

Global View

Local View (4 processes)

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**
    - **http://www-unix.mcs.anl.gov/mpi**
  - **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**

- **Most MPI 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 Single Program Multiple Data (SPMD) model with a Local View

SPMD code

– Write one piece of code that executes on each processor

Local View (4 processes)
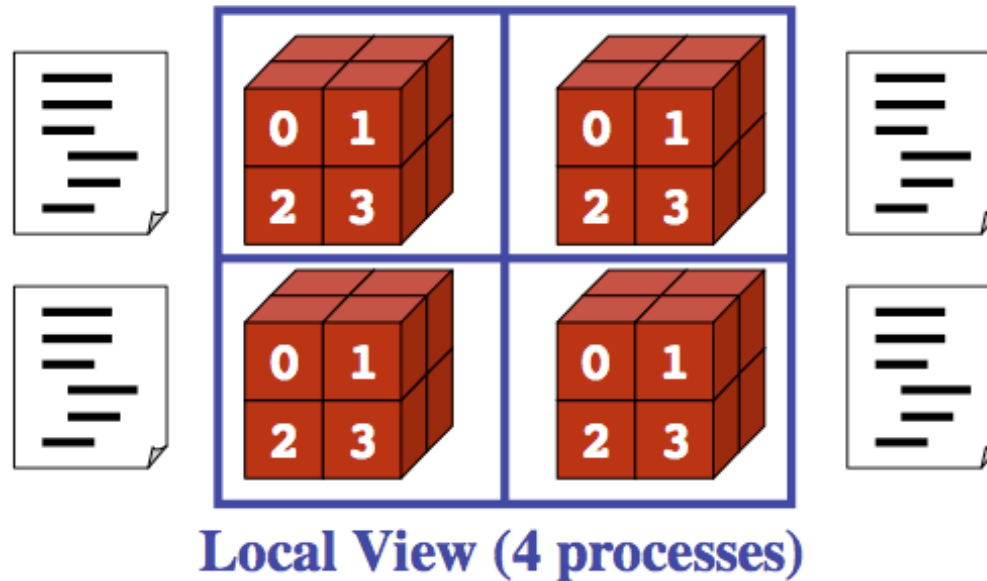
Processors must communicate via messages for non-local data accesses

- **Similar to communication constraint for actors (except that we allow hybrid combinations of task parallelism and actor parallelism in HJ)**

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

- **<u>Note:</u>**
  —In this subset, processes act independently with no information communicated among the processes.
  —COMM_WORLD is the default communicator that includes all N processes, and numbers them with ranks from 0 to N-1

Our First MPI Program (mpiJava version)

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.}
```

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**

MPI.COMM_WORLD

0 5 6 3 2 1 4 7
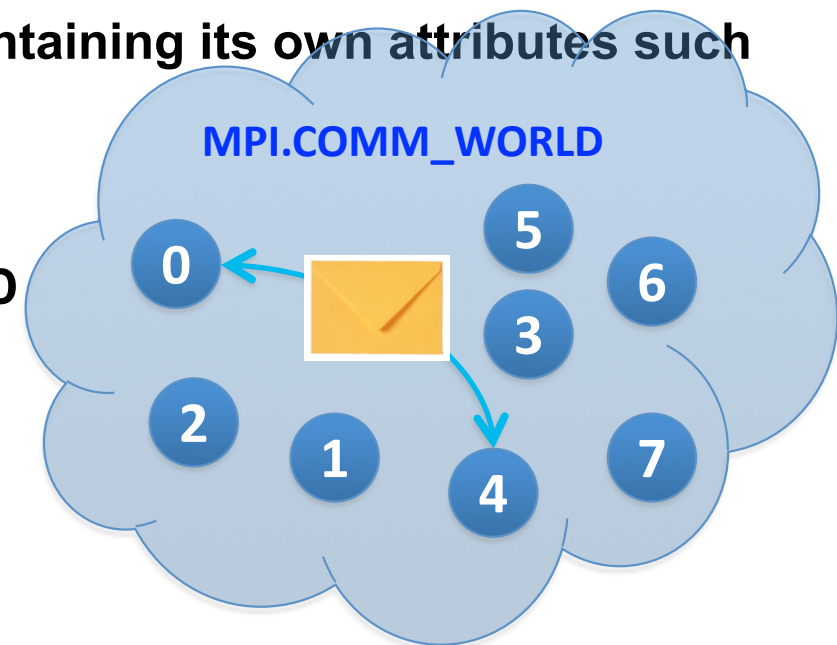
Adding Send() and Recv() to the

- **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

Point-to-point commn

MPI Blocking Point to Point Communication: Basic Idea

- **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 send and receive

- Send and receive members of Comm:

  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 <u>blocking</u> operations by default

  — Analogous to a phaser next operation

Example of Send and Recv

```
1. import mpi.*;
2.  class myProg {
3.   public static void main( String[] args ) {
4.     int tag0 = 0; int tag1 = 1;
5.    MPI.Init( args );                    // Start MPI computation
6.    if ( MPI.COMM_WORLD.rank() == 0 ) { // rank 0 = sender
7.      int loop[] = new int[1]; loop[0] = 3;
8.      MPI.COMM_WORLD.Send( "Hello World!", 0, 12, MPI.CHAR, 1, tag0 );
9.      MPI.COMM_WORLD.Send( loop, 0, 1, MPI.INT, 1, tag1 );
10.    } else {                            // rank 1 = receiver
11.      int loop[] = new int[1]; char msg[] = new char[12];
12.      MPI.COMM_WORLD.Recv( msg, 0, 12, MPI.CHAR, 0, tag0 );
13.      MPI.COMM_WORLD.Recv( loop, 0, 1, MPI.INT, 0, tag1 );
14.      for ( int i = 0; i < loop[0]; i++ ) System.out.println( msg );
15.    }
16.    MPI.Finalize( );                    // Finish MPI computation
17.  }
18. }
```

Send() and Recv() calls are blocking operations by default