|epcc|

EPCC-SS-98-04

Java Simulation of MPI Collective Communications

Tom Doel

Abstract

MPI (Message Passing Interface) is an application programming interface for Fortran and C that allow communication between a number of processors on a parallel (multiprocessor) computer. This project is designed to complement MPI training courses at Edinburgh Parallel Computing Centre by providing a simulation of MPI collective communications, to help teach the concepts involved.

Contents

1 Introduction

1.1 Parallel Computing

A common programming model for parallel computation is SPMD (Single Program, Multiple Data), where each processor runs the same program, but operates on different memory and need not be executing the same command at the same time. A combination of a processor, its local memory, and the hardward support for it is called a *processing element* (PE). Often each PE will only run a single process, so here we use the terms interchangeably.

Often a process will require some data from another process' memory. The hardware and operating systems of the parallel machine will generally provide some method of transferring data between processes. Unfortunately, these routines will be specific to the particular machine being used, so code written for one parallel machine is not generally portable if it uses such communication routines.

1.2 MPI

To solve this portability problem, MPI (Message Passing Interface [2]) was developed. This is a standard set of routines for C and Fortran to pass data between processes. On each parallel machine, MPI will be implemented differently, but the actual commands used in the programming language remain the same. The MPI message-passing routines - those that pass data between processes - can be divided into two categories. *Point-to-point* communications concern data transfer (message passing) from one individual process to another. *Collective* communications are those that involve a group of processes exchanging data.

1.3 MPI Simulation

EPCC [1] provide training courses in MPI usage. In order to help trainees understand the underlying concepts, a simulation of Point to Point communications was developed by SELLIC at the University of Edinburgh ([3]). This was in the form of a Java applet that could be run from a web browser. The purpose of this project is to create an applet to complement the existing program, by simulating MPI collective commands.

The applet must be reasonably straightforward to use, and be intuitively close to the way MPI is actually used. This means that it should be flexible, but not so much so that it becomes too complicated to understand.

2 The Point-to-Point Java Simulation

The project is based on an existing Java applet package, Mpidemo, which illustrates point-to-point communications. This consists of a display of three processes, each containing a list of MPI instructions. These instructions can be added to or cleared using a Controller panel which contains buttons and menus for performing these tasks. There are run and stop buttons that control the execution of the programs on each process, which run in parallel. Messages are represented by graphical envelopes which move from process to process. The buffered, standard, synchronous, ready send and receive commands are implemented.

The Mpidemo program uses Java Beans. These are independent programming components which conform to certain set standards which allow them to be reused in other programs without code alteration. The principle advantages of using Beans in this simulation is that each runs as an independent thread through time sharing, and any number of Beans of each type can be created without problems with resource sharing, since each Bean is an independent entity.

Each Process Bean acts on signals from a Controller Bean, and creates a thread under which the program is executed. The graphical envelopes are displayed in Beans that implement a class called EnvelopeBox, which is basically a canvas on which the moving envelopes are drawn. There is an overall initialisation program called MPI.java; otherwise all the beans work independently.

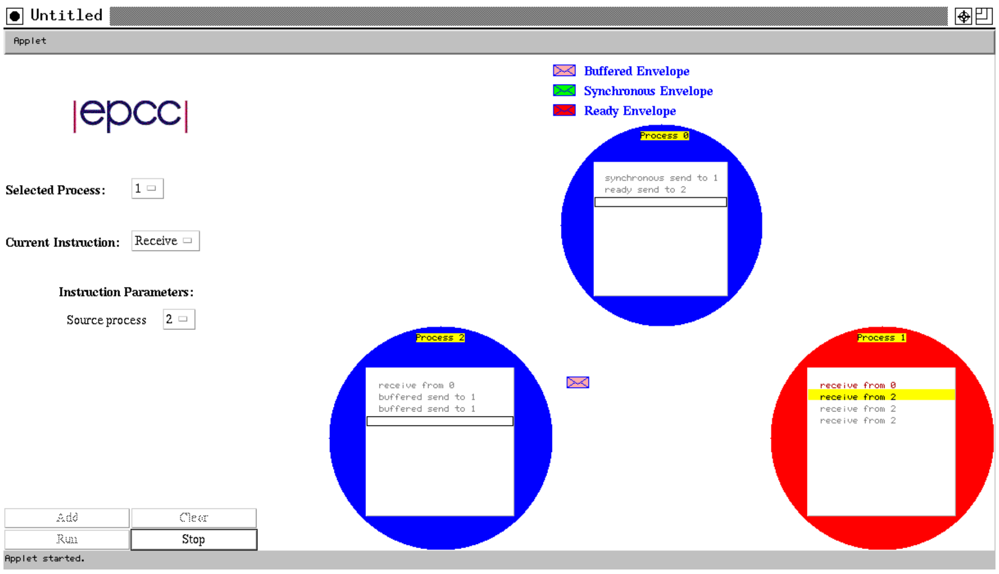

Figure 1: The Mpidemo point-to-point simulation

3 Project Specifications

The project is designed as a simulation of MPI collective communications which aims to assist in the teaching of MPI. In the original specification, each command would be simulated by a separate Java applet; however, it was agreed that a better approach would be to write one applet that simulated all the commands. The original specifications were as follows:

- The simulations should be developed as Java applets so that they can be run from the online MPI courses. Each collective command could be implemented as a separate applet, or they could all be incorporated into one program.
- Each simulation consists of three processes, each containing a list of instructions which are MPI collective commands. The user can add instructions to the list by using a user interface, where they can select from different typeins/options which depend on the command being added.
- Each process will contain integer data elements, which will be modified appropriately by the collective command. The alteration of these data elements will provide the main visualisation of how the MPI collective commands actually work.
- The principal commands to be implemented are: Barrier, Broadcast, Scatter, Reduction.
- There should be error checking to detect if the user has entered any parameters incorrectly or has otherwise done something that could cause MPI not to function as expected.

3.1 MPI Collective Communications

The following MPI commands were simulated in the project.

Barrier This is a synchronisation command; it halts execution of the process until all processes have executed this instruction.

Broadcast Data elements from a specified process (the *root process*) are copied to all other processes.

Gather Data elements are taken from all the processes and stored sequentially on the root process, in the order of the ranks of each process.

All Gather The same as Gather, but the resulting data is stored on all processes.

Scatter Data from the root process is taken sequentially, and different elements sent to each process in rank order. It is effectively the reverse of Gather.

Reduce Data elements are combined from all processes using a reduction operation, and the result is stored on the root process.

All Reduce The same as reduce, but the result is stored on all processes.

Scan This is similar to All Reduce, but the data from each process is combined one-by-one using the reduction operation in rank order, and the result of each reduction stored on the respective process.

4 Design

The modular Java Bean structure of the original MPIdemo program invites the possibility of adapting it to use Collective communications, whilst retaining the Beans which handle the point-to-point code.

Because the original Mpidemo program is structured to use Java Beans, it possible to add the collective communication instructions whilst retaining the Point-to-Point code. While the design of the original code meant that a few fundamental changes were necessary to the structure and implementation of the applet, most of the new Collective code was added in the form of new Java Beans and additional classes.

It was possible to extend the project beyond the original design specifications, to produce a simulation that was considerable more flexible, adaptable and comprehensive. The following design was implemented:

- One single applet was developed to simulate all the collective commands. This means that commands can be intermixed, making the simulation more flexible and minimising the duplication of code.
- The point to point simulation was incorporated into the program, allowing the applet to be used as a single MPI simulation for both point-to-point and collective communications.
- A completely new user interface was designed. This included more functionality than the original program, including editing existing instructions and copy/paste from one process to another.

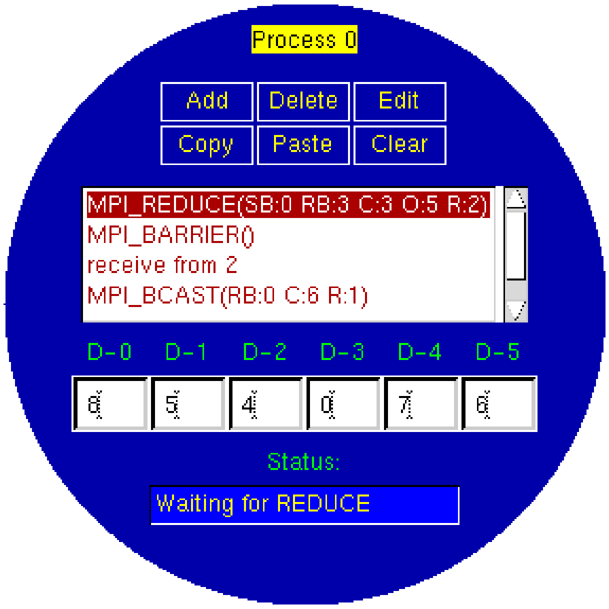

Figure 2: The Process Java Bean and user interface

- Data is visualised using editable text boxes where users can input data or click a Random Data button which inserts random values into all the data boxes on all processes. The data is modified as appropriate during the execution of collective communication commands.

- The collective communications are implemented as non-synchronous blocking commands. This means that control is returned to a calling process when the data on that process has been successfully retrieved or modified, but processes are not necessarily synchronised after calls to one instruction (with the obvious exception of Barrier).
- Error messages are displayed if the user enters invalid data into the data boxes.
- The simulation implements Barrier, Broadcast, Gather, Scatter, AllGather, Reduce, AllReduce and Scan, with the latter three commands implementing Sum, Product, Max, Min, and logical OR, AND and XOR operations.

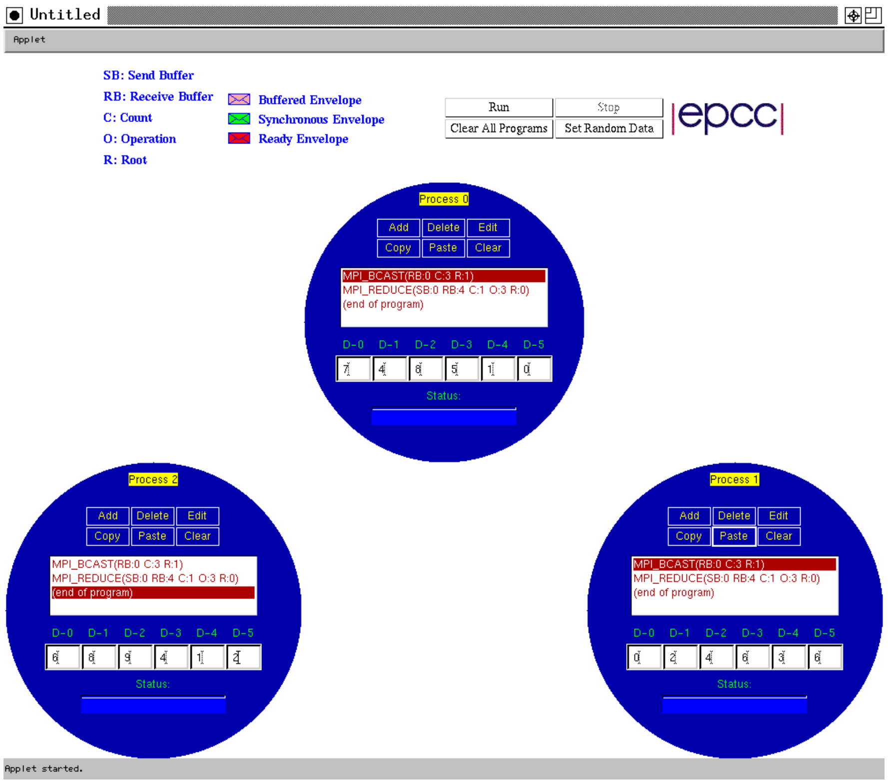

Figure 3: Screen display of the collective communications simulation

4.1 User Interface

The applet provided a considerably enhanced user interface with some useful features.

- A clickable program listing with a scrollbar.
- A scrollable status window on each process.
- Add, Edit and Delete buttons to insert, modify or erase an instruction at the currently selected position.
- A pop-up input dialog window for selecting an instruction and its parameters. This made it easier to use and avoids unnecessary screen clutter.
- A Clear button to delete all instructions on a process, and a Clear All button to clear programs on all processes.
- Editable data boxes representing the memory locations that are used by the collective communications.
- A Set Random Data button to insert random numbers into all data boxes.
- Pop-up error boxes for reporting when invalid input has been entered into the data boxes.
- Start and Stop buttons for starting and halting execution of the programs on each process.
- Retention of the point-to-point visualisation using graphical envelopes.
- Buttons which create a link to a web page to provide help on MPI commands.

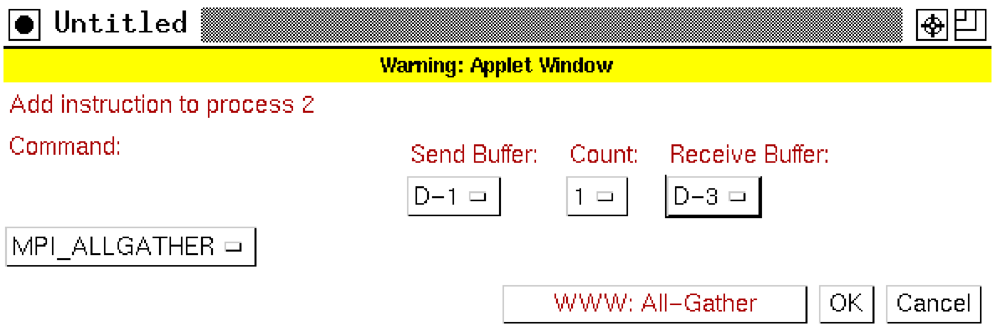

Figure 4: The pop-up instruction input menu

5 Implementation

The applet consists of a number of Java Beans. Each process is simulated by a Process Bean. Between any two processes there is a Envelope Box Bean, which handles the simulated point-to-point communications between those two processes. There is also one Communicator Bean which handles all collective communication commands, and one Controller Bean which is used to communicate with all the Beans, principally to start and stop the simulation running.

Java Beans are in principle independent modules and do not communicate directly with external programs. Instead, they use a concept which could loosely be described as a message ring, where messages are passed from the sender to a number of listeners. Java Beans are generally set up to listen to each other by an initialisation program, and each Bean will examine the messages passed to the ring and act on them where appropriate.

This program uses two independent message rings; one for the point-to-point communications, which formed part of the original simulation, and one for both the collective communications and the overall control of the Beans. Since all messages are individually distinguished, these rings could be combined into one, but having two independent rings has certain coding advantages and improves the clarity of the program structure.

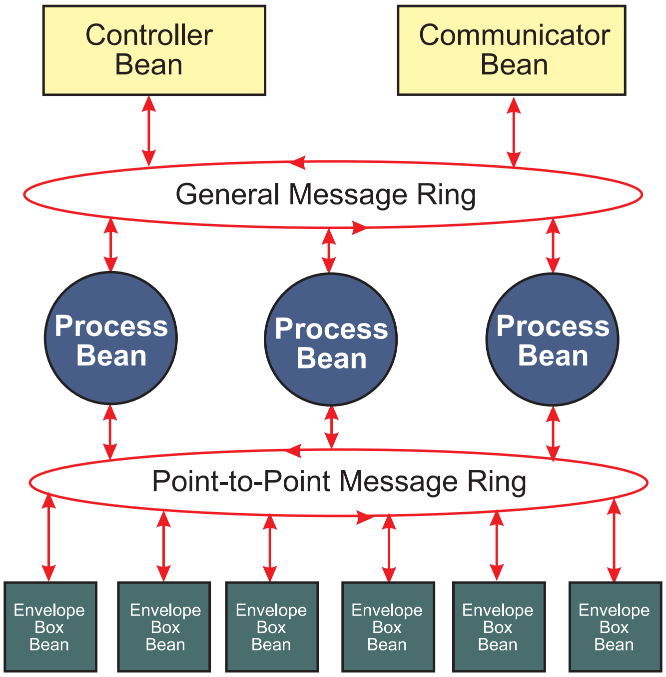

Figure 5: Illustration of communications between the Java Beans

5.1 Initialisation

The initialiser program, MPI.java, creates the Beans and sets them up to listen to the appropriate message rings. It is this program which determines how many processes are present in the simulation; a direct advantage of the Java Bean approach is that any number of processes can be simulated without any recoding of the Java Beans. Each Java Bean runs as an independent thread; when the initialisation program has finished, it simple exits and leaves the Beans to run by themselves.

The MPI.java file sets up three Processes, with six Envelope Box Beans (one corresponding to each one-way point-to-point communication). The communication between the Beans is illustrated in the diagram, where each Bean and each line of communication between a Bean and a message ring has been set up by the initialiser program. In this program, all communication is two-way, ie. all Beans need to listen as well as add to message rings.

5.2 The Process Java Bean

The Process Bean contains the principle user interface components, and creates the pop-up input and error boxes. Command instructions are stored in an array and executed by sending signals to either the Envelope Box beans or to the Communicator, depending on the type of instruction being executed.

5.3 The Controller Java Bean

The Controller oversees the other processes, and contains the Run, Stop, Clear All Programs and Set Random Data buttons. When one of these buttons is pressed, the Controller sends an appropriate signal to be received by all the processes, which then perform the necessary tasks. The Controller also performs actions based on signals from the processes; for example, sending Copy data to all the processes, or stopping all the processes running when invalid data has been found in one process. This is a much simpler approach than having all the processes listening to each other.

5.4 The Communicator Java Bean

The Communicator handles all collective communication instructions, which are sent to the message ring by each process. Since collective communications are blocking, each process then waits for a signal from the Communicator before proceeding. A new thread is started to deal with each instruction, and messages are returned to each process containing the result data and informing them to resume operation as and when it is appropriate for them to do so.

5.5 The Envelope Box Java Beans

These Java Beans have been retained from the point-to-point simulation without any recoding. They handle the point-to-point communications, and visually consist of a canvas on which envelopes are drawn to simulate messages moving from one process to another.

5.6 User Interface

The user interface components are contained in the Controller and the Process Java Beans as appropriate. Each process has a pop-up window to display an instruction input dialog and error messages; this window is hidden when not in use rather than destroyed, as to create new windows is noticeably slow. Note that at present the send/receive instructions do not use their buffer/count parameters as the original point-to-point code incorporated into this program does not include the functionality to actually send data.

5.7 Support Classes

Additional classes have been created to provide support for the code, but which do not directly form part of it. These include classes for each type of instruction, and a Message class used to package instructions and data sent on the main message ring to the Communicator. Additionally, a GotoURL class was created to provide simple-to-use but powerful functionality for creating buttons that link to web pages.

6 Limitations and Further Improvements

While this project has achieved considerably more than that which formed the original specifications, there are some further desirable enhancements to the program which time did not permit. Furthermore, there are a number of other additions which, although beyond the scope of this project, could be implemented as a future update.

- The point-to-point simulation has been retained unmodified from the original simulation. This means that there is no functionality to transfer data from one process to another through point-to-point communications. This could easily be implemented with some modifications to the design of the point-to-point code.
- More MPI commands could be added, such as All-to-All, and other functions could be added to the Scan and Reduce instructions, such as MINLOC and MAXLOC.
- This applet simulates a non-synchronous implementation of MPI, but it would be straightforward to include a synchronous version, and allow the user to choose which to use, perhaps using buttons on the Controller. This would aid in teaching about the pitfalls of writing programs that depend on MPI commands being synchronous or non-synchronous, as the MPI standard allows either to be implemented.
- The user interface could be enhanced to emphasize more the Fortran and C style of MPI calls. The differences in calls between the two languages could be implemented and the user could select between the two via buttons on the controller.
- The visual design of the applet and the user interface could be improved; for example, a background graphic could replace the current process circles. The Swing package could be used to provide much more pleasing user interface components than those provided by the standard Java packages.
- A computation instruction could be added; the purpose of this is to simulate the execution of normal (ie. non-MPI) program code by creating a delay. The instruction class to support this has already been added, but has not been implemented. This would require simple modifications to the Process Bean, and the addition of the computation instruction to the input dialog.

7 Summary

- This program furfils the specifications laid out above and simulates the main MPI collective communication commands.
- The complete functionality of the existing point-to-point simulation is incorporated to provide a more comprehensive simulation.
- A versatile user interface has been created which is simple to learn.
- The modular, object-orientated approach of the coding allows future improvements, such as those suggested, without major alteration to the existing code.

References

[1] Edinburgh Parallel Computing Centre. EPCC. http://www.epcc.ed.ac.uk.

[2] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994. http://www.mcs.anl.gov/mpi.

[3] University of Edinburgh. SELLIC science and engineering library learning and information centre. http://www.sellic.ed.ac.uk.

About the Author

Tom Doel is a Mathematical Physics student at the University of Edinburgh where he is entering 4th year of a 5-year MPhys course. He completed this project whilst under the EPCC Summer Scholarship Programme, July-September 1998, and can be contacted at tom_doel@bigfoot.com

Supervisors: Elspeth Minty (MPI), Bobby Davey (Java)

8 Appendix: User Manual

8.1 Program Layout

The screen is divided into three processes and a top panel. Each process contains a list of instructions which can be added to or modified using the six buttons on the process circle. There are also six labelled data boxes on each process which represent memory locations.

The top panel contains four buttons; the Start and Stop buttons begin and end the running of the simulation. The Clear All button deletes all instructions from all the processes, while the Set Random Data button inserts random numbers into all the data boxes on all the processes.

8.2 Adding instructions

Click Add on a process to add an instruction to that process. Note that when clicking Add for the first time on a process, there may be a short delay. A window will pop-up, allowing you to select the type of instruction you want, and the parameters that go with it. To select a command or parameter, click on the appropriate box, and select the alternative you want from the menu that appears. The parameters that are available depend on the current instruction that is selected, and are designed to reflect actual MPI parameters- see MPI documentation to understand how the parameters work.

To edit or delete an instruction, click on that instruction and click the appropriate button. When the program is larger than the program window, you can use the scrollbar to find the command you want. You can insert a new instruction by selecting a position before clicking Add. You can use the Clear button to delete all instructions in that process.

8.3 Copy and Paste

Use the Copy button to copy the entire program into a buffer, and the Paste to replace the current program with the one in the buffer. These are useful with collective communications where the programs will often be the same in all processes.

8.4 Data Boxes

To change the value of a data box, simply click on the box and change the number as if it were text. You must only enter integer numbers; anything else will result in an error message being displayed when the program is run. Note that the data boxes scroll, so you may not see the full number shown at a given time.

8.5 Running the Simulation

Click start to run the simulation. The instructions will be executed one by one. the send/receive instructions will result in graphical envelopes moving from one process to another. Other instructions will cause data to be modified. The current status of each process is displayed at its base; you can scroll through the previous status messages using the scrollbar. If you have not entered the correct commands in all processes, the simulation will halt. Simply press Stop to end the simulation, and correct the error. Note that you must press stop at the end of the simulation before you can edit the instructions or data, even when all the processes have finished running.

8.6 Errors

If you enter a non-integer into a data box, the program will halt on running with an error message, and the appropriate box will show "***". Replace the "***" with an integer value and continue. Each process with a non-integer value will display an error message.

If the parameters you enter cause memory outside the data buffer to be accessed, an error message will be displayed during running, but the programs will not halt, as in real life MPI would not have detected this error. The simulation will not be harmed, but doing this in real MPI could cause data corruption or a segmentation error.