|epcc|

THE UNIVERSITY OF EDINBURGH

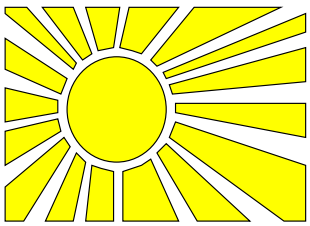
SS-2001-04

The Java MPI Simulator 2001

Rok Preskar

Abstract

MPI (Message Passing Interface) [3] is a standard for message passing. It has C, C++ and Fortran bindings that allows independent processes with separate memory spaces to exchange data while working on different parts of a problem. The Java MPI simulator was developed as an applet in order to demonstrate these communications and present them in a comprehensive manner. The goal of this project was to make the applet more consistent and open for improvements as well as more intuitive to the user.



Contents

1 Introduction

Message Passing Interface (MPI) is a standard for communication between processes. Before this standard each parallel computing vendor developed its own message passing libraries. This made source portability between different parallel machines difficult until MPI was created.

The Java MPI simulator was designed to help people learning MPI understand communications between processes and learn different MPI commands by visualising the message passing process. The applet was first developed in 1997 as a simulator of blocking point-to-point communications and then, in 1998, extended by another SSP project to include collective communications [2]. Up to this stage two programmers had worked separately on the simulator which had led to a few inconsistencies within it as well as a lot of undocumented code and some bugs. This was, in part, corrected by another SSP project in 2000 [4], which dealt mostly with understandability, documentation and refactoring of the existing code.

Though much work had been done to improve the applet there was still room to further extend its functionality and make it more intuitive.

Thus the goals of this project were:

- remove inconsistencies between collective and point-to-point communications,
- remove dependencies among different Beans the applet is comprised of,
- implement Java Swing instead of AWT to provide consistent cross-platform appearance,
- provide functionality for saving and restoring programmes,
- enable the user to set the number of processes within the applet,
- make the applet more intuitive for the user,
- find and remove some of the known bugs.

2 Background

As mentioned in the introduction the applet was first developed to simulate point-to-point communications between processes. The general idea was to try to visualise the data exchange between three processes as an exchange of envelopes. This has remained unaltered from that beginning. Also, the applet components were coded as Java Beans from the start, which made further improvements less difficult. Java Beans are independent programming components that must conform to certain standards (such as implementation of public set and get methods for relevant properties). The independency of the Beans allowed for modification of existing code to have a smaller impact on the functionality of other parts of the code.

In 1998 an SSP project took this forward by extending the applet to incorporate collective communications [2]. Considerable redesigning of the applet's user interface was done and collective communications were implemented. These allow global operations to be performed, e.g. a global summation. Two message rings were implemented; one to allow user to control the applet through the controller bean and to pass the collective communications and the other ring for point-to-point communications.

Another SSP project continued development of the applet in 2000 [4]. The results of this project lead to an increase in the understandability of the code, better documentation and refactored code that was more robust. This was necessary to make future development of the applet easier.

In general, while the code was promising and well designed there was still plenty of scope for improvement. Lots of small errors were found relatively quickly and removed. This led to a better foundation for continuing the development.

3 The User interface

3.1 The applet before the project

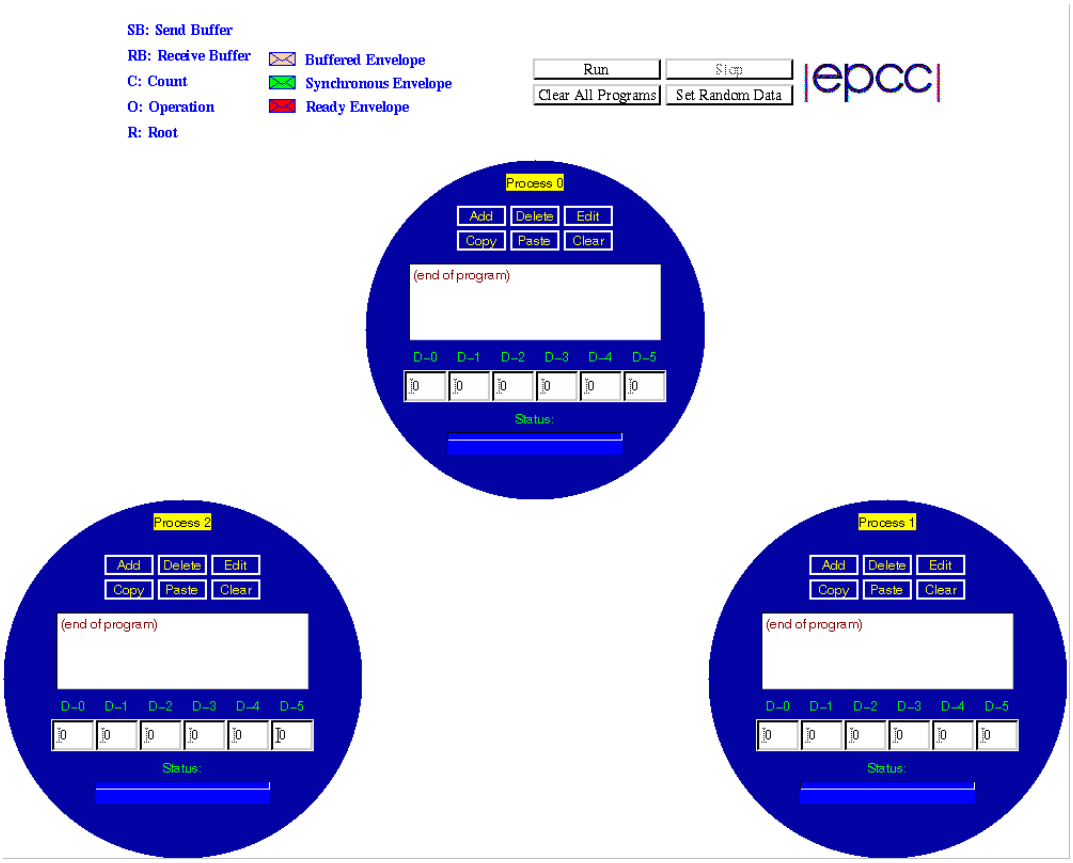


Figure 1: The applet at the start of this project.

The applet consisted of several modules (coded as Java Beans), which, contrary to Java Bean philosophy, were not completely independent. These were:

- **Miscutils** - Part of a different package, the utilities were scarcely used.
- **Envelope box** - A canvas capable of drawing the travelling envelopes.
- **Process** - A component representing one process involved in communications.
- **Controller** - A panel of buttons providing general applet commands.

- **Communicator** - Hidden component that supervised the collective communications.
- **MPI** - The runnable applet containing the other components listed above.

The applet consisted of three processes (the word ‘process’ is in this report used interchangeably with the word ‘processor’ as each processor supposedly, to have efficient parallel code, runs only one process) that exchanged envelopes between each other. The processes were illustrated as circular entities as seen on the figure 1. Each process had a list of commands to be executed and its own data. The commands were displayed as a list in the centre of the circle. The data was represented by a row of six integer values directly beneath the command list. The control buttons were set up to edit the commands and to add them.

Between any two processes there were two envelope box canvases that carried envelopes to their destination. Each canvas could only draw one path, which made the layout of the applet difficult and the envelopes started their way quite far from the originating process because different canvases cannot draw in the same area. This made the representation of sending and receiving data a bit confusing.

A control panel was set up that allowed the user to start or stop the programme. It also allowed the user to set random data on every process and to clear all process instruction lists.

The communicator handled all the collective communications by gathering the data, performing the operation and then sending the data back. This was not visualised.

3.2 Current status

In figure 2 the current appearance of the applet is shown in the process of sending and receiving data from the other processes. The data exchange is represented by envelopes that, when they arrive at their destination, pass the data to the receiving process. This did not happen before.

The original appearance of the applet was kept unchanged as much as possible as it was considered to be quite effective if not exactly user-friendly. However, there were many changes made during the project as there were ideas on how to improve the applet. The major changes are listed in the implementation section but here are some of the smaller ones:

- The appearance of the process was changed to remove scaling problems (see figure 3),
- The numbers beside the programme list indicate consecutive numbers of commands the user programmes in,
- Improved diagnostic and error messages, e.g. overflow of the buffer, etc.

These modifications were necessary to make the applet more portable and easier to use but do not need more explanation than this.

4 Implementation

4.1 Inconsistencies between point-to-point and collective communications

The main objective of this part of the project was to remove the inconsistencies between collective and point-to-point communications. The former was represented as envelopes passing from

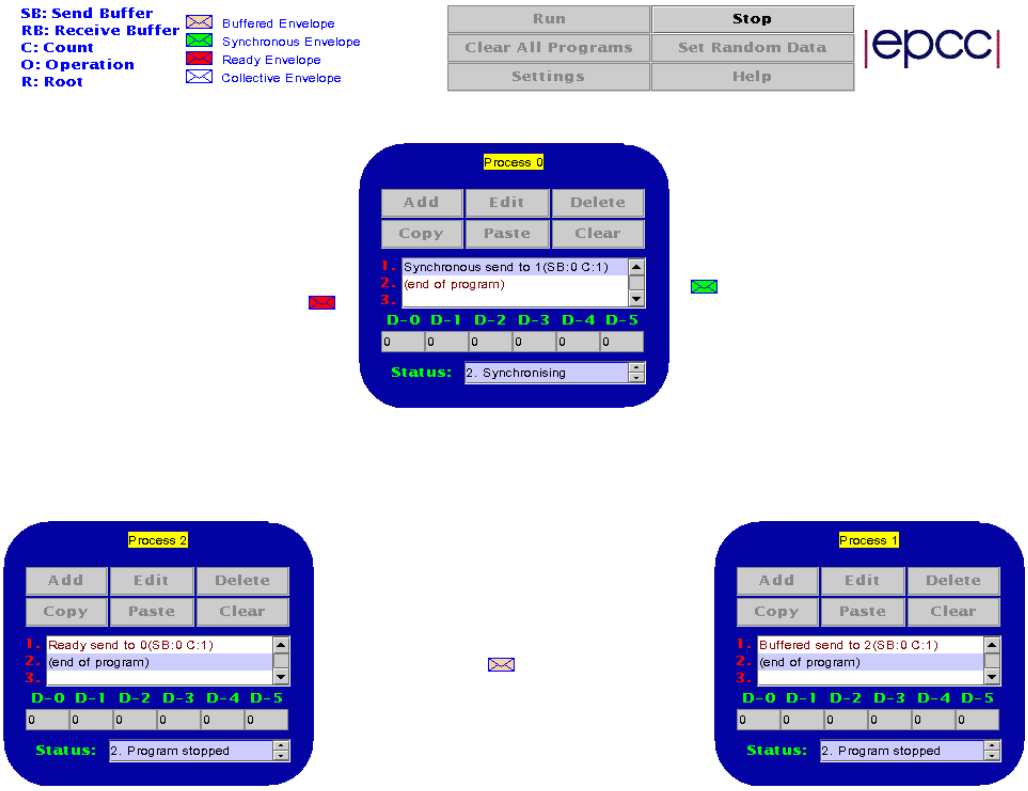


Figure 2: The current version of the applet

Figure 3: The new appearance of a process

process to process and not transferring any data and the latter was represented with just the data transfer and was done mostly by Communicator. This Bean collected all information it needed for the collective communication, performed the operation and then sent the modified data back to processes. Since Communicator was not seen on the screen the result could only be seen in the data of each process.

At the beginning of this project there were two message passing rings, which functioned independently of each other. One was communicating with processes via the `EnvelopeEvents`, which were part of the Envelope Box module, and the other was communicating through property change events. The former handled messages coming from envelope boxes and was in charge of point-to-point message passing, while the latter was in charge of receiving messages from Controller and Communicator Java Beans.

It was observed that in order to make communications more consistent these two rings should not be separated so much. Through property change events these two could still be separate enough for the code to remain comprehensive and close enough not to be the cause of inconsistencies.

4.1.1 Point-to-point communication

The first change to communications was made by adding data to envelopes. Previously, the envelopes only travelled back and forth without actually transferring data to the receiving process. When the envelope arrives to its destination, an event, which contains data, is fired. The receiving process then copies the data and resumes its execution.

4.1.2 Collective communication

Collective communications were substantially altered as the execution of collective communication was transferred from communicator to the processes themselves. This was necessary because the collective communications were being represented by data-carrying envelopes and that was easier to implement on the processes themselves. It was also easier to simulate exact execution of MPI within the code by finding the description of how communications look from process's view in the MPI Standard [3].

Also, the Communicator is not the main carrier of collective communication anymore, so it was considered it would be better to join the Communicator and Controller together, thus making the Controller the only Bean that was controlling other Beans. There are two things a process can ask to be notified of:

1. That root process of the instruction has arrived.
2. That all processes have reached the same instruction.

This is the only part that is done by the controller as everything else is done through arrival of collective envelopes and modification of local data in the process. This way is much more convenient and clearer but it means that more functionality rests within the Process bean.

4.2 Bean and module independency

This applet has been constructed using Beans from the very beginning. As mentioned before, a Java Bean is more of a class that conforms to a set of standards so that even if we want to comply with them, deviation is possible simply because the compiler does not point out the irregularities. If a component is a Java Bean it should be a reusable component with methods that allow a programmer to customise it to better fit into the current programme. If the result is to be a reusable component it should be independent of its surrounding classes. During the previous stages of the development of the applet several dependencies appeared, some of which were even circular.

4.2.1 The dependencies

The most basic dependencies existed and still exist within the MPI module. This module was set up to initialise, display and set up event listening between the independent Beans. These dependencies are probably not removable, since the Beans communicate with each other using these events.

There were also several circular dependencies, however since they were noted by the previous SSP project and some were corrected then, only trivial dependencies, such as calling methods of a bean that was higher in the dependency hierarchy, remained. These were fortunately rather quickly dealt with by using events to call these methods instead of calling them directly.

Disposing of such smaller dependencies left the applet with just one. The dependency existed between the Process and Envelope Box module. This dependency was in the shared data structures, e. g. envelope events. When an envelope arrives at its target it has to give information to the receiver and that information has to be in some recognizable form. To ensure understandability of the code this form has to be a structure or class, the definition of which has to be shared by both Beans.

This remaining dependency would not qualify as really being a serious one. For this reason it was decided to put a three-level architecture model within the project. As shown on figure 4 the Miscutils module is the lowest and contains the classes that are shared by the other Beans simply as structures of data and so cannot be cleanly removed as the Beans have to pass each other information using these. The highest is MPI module, which depends on all the other Beans. It would be possible to make MPI bean independent of other Beans by putting all the basic structures in the Miscutils module but that was considered to be too independent as the MPI module is quite specific to this project and making it independent would overly complicate the programme, where this was not necessary. All the other modules are on the same level and are independently compilable.

4.2.2 Why the complete structural redesign of the envelope box module?

There was also a functionality problem with the Envelope Box as the events to add the envelopes were passed without the signature of the sending process. The original implementation relied on each canvas to draw only one envelope path. For instance, if an envelope is travelling form process 0 to process 1 it would traverse a different canvas than an envelope from process 1 to process 0. Implementing envelope messages with property change events was a considerable

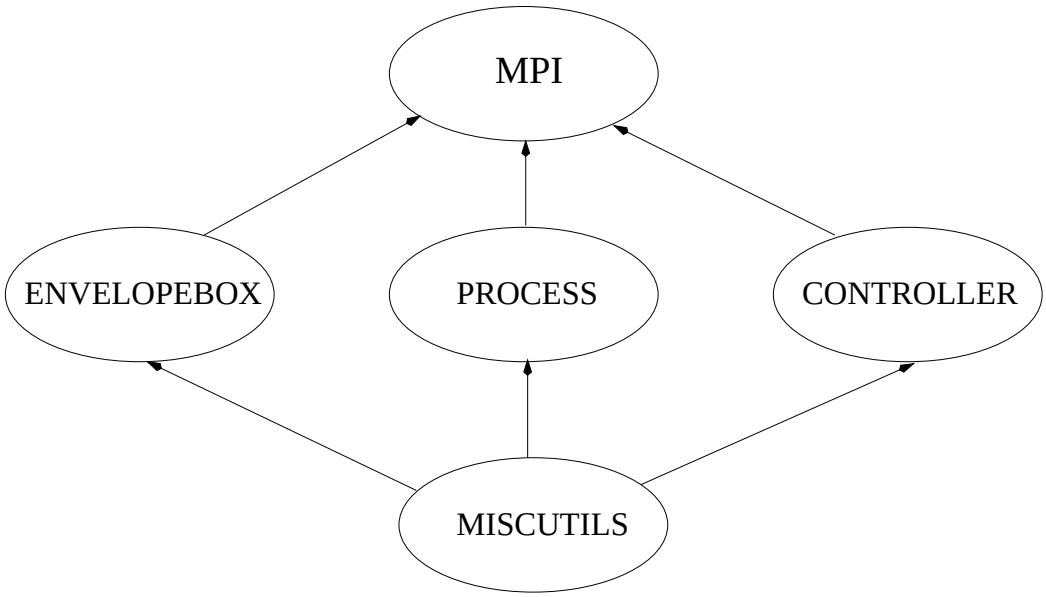


Figure 4: The dependencies between modules.

advantage as every canvas is already set up to listen to all processes' property changes and so, modification to display more or less envelope paths is made easier. For example, if a programmer would suddenly decide that an envelope should travel a different route to its destination, the only changes involved would be to call a method differently. This increases the usability of the Envelope Box and thus makes it conform more to the ideas of Java Beans.

For this major reason and consistencies when receiving and sending messages the previous event model was replaced by property change events so now the internal communication between the Beans is carried out completely by the property change model. It is debatable whether the original model of envelope events should have been kept as both models conform to Java Beans standards but the reason for choosing property change events was just to keep all similar actions as close together in the code as possible.

4.3 Conversion from AWT to Swing

The acronym AWT stands for *Advanced Windows Toolkit*, a toolkit for creating graphical user interfaces (GUIs). It was designed to make creating a GUI easier and it supported corrections after resizing. There were several objections to AWT as it was not as portable as other Java packages, mostly because it had *heavy-weight* components, which use system dependent appearance and resources.

The Java Swing is a package that was developed after AWT and its major advantage is that most of its components are light-weight. A *light-weight component* is a component that is system independent and differs from its heavy-weight peers by consulting only Java's resources. That has two advantages. First is that light-weight components load much quicker, are nicer in appearance and are better customisable. The second advantage is that, since they are system independent, they will appear the same on every system.

It was thought that converting the current applet from AWT to Swing was going to present

major benefits for the applet. The applet would become more generalisable as Swing offers more customisable functionality than AWT, and would become nicer for the user as there would be no unfamiliarity confusions (“It looks different on this system”).

There were, however, several difficulties with the conversion. It is usually said that converting from AWT to Swing is more or less a matter of putting the letter ‘J’ in front of the class name. That has appeared to be true for most components, however the functionality of JList class, for example, has changed considerably and had to be refactored completely. The advantages have already proven to outweigh these difficulties. More on this subject can be found in [5].

4.3.1 Problems with Swing

There were several strange problems encountered when working with the Java Swing classes. It seems that certain complex layout managers (such as the GridBagLayout) display differently in Swing than in AWT. This led to several changes in the layout part of the code (using the layout manager a bit differently) but were made as transparent as possible so that the user interface would not change too much.

More to the point when using JOptionPane class several problems were encountered. In that class there are several static methods that allow the programmer to construct question, error and other message windows with just one line of code. This had seemed to be a useful thing to do but it seems that these methods are not as reliable as was first assumed. Sometimes the window that appears is not displayed correctly (not all parts of the window are correctly painted) or is displayed in such way that it cannot receive any input from the user. Since the window is modal (it does not allow input to any other window of the application while it is displayed) it means that the entire applet has to be destroyed and restarted to regain functionality. However, it was suggested that it might be a window manager problem and not the problem of the applet. Please note that should the *load, save or delete dialogs and error messages* display incorrectly, they are fully created and maintained by Java installed on the machine the applet is being run on. More on JOptionPane class and how to create these windows can be found in [5]. Unfortunately, the book does not list any such problems and does not offer any solutions to it.

4.4 Saving programmes

Saving programmes and loading them later is considered to be a very useful aspect of functionality because, if the applet is to be used in a teaching environment, it would be much easier for the lecturer to simply load a previously constructed example and then demonstrate them to the people learning MPI. It was decided that the applet should save the instruction sets on each process as well as the data residing there. The only difficulty remaining was how to access the disk from an applet.

4.4.1 How does it work?

The process of saving is very simple at its core. When the user presses the settings button available in the controller bean a frame appears with several options. Among these are *save*, *load* and *delete* programmes. When the user chooses an action an event is fired, telling the other Beans to send the information to save. In response each process fires property changes carrying

first instruction list and then the actual data. These are gathered together in the controller class and then simply serialised to file. There is, actually, only one file, which carries all the different programmes, and stores them as a vector into a file.

Note that the file is stored locally on disk and, if the path is not specified, in working directory of the application running the applet. As such, creating the programmes is up to the user and does not depend on any other facts. This may not always be desired, but then the programmes can be distributed separately from the applet itself and so different tutorials can be set up in different files on the web.

4.4.2 Why does it not work everywhere?

Normally, the applet is granted no access to any system resources. However, there are several possibilities for an applet to gain access to the disk, all of them involving set up by the user. It is possible to sign the applet as it was done during the test phase (self-signing was used) but this resulted in only very limited access. The access included creating and writing to a file, however subsequent reads were not granted. The file had the same kind of permissions as normally created files and thus was not guaranteed to be allowed to reaccess it. Also, getting the applet signed properly (to view it through netscape) might present a problem since there are only a few vendors that netscape accepts certificates from.

Another way to allow the applet read/write access to the disk is by means of creating and installing a Java policy file. The Java policy system is an extension to the former Sandbox security model, which created a playground for applets and they could not reach out from it. Through setting up a policy system you can specify which applets to trust and which not to trust. You also specify exactly which parts of the disk the applet can access.

Here is an example of contents of a Java policy file:

```
grant codeBase "http://www.epcc.ed.ac.uk/~rok/-" {
  permission java.io.FilePermission
    "/disk/home/epcc-ss/v1rpresk/MPISIMsaves", "write, read";
};
```

The string following the keyword `codeBase` is the location of the classes that are allowed to access the disk. This means that the user or system administrator should be careful what location he or she is granting disk access to. The following important keyword is `permission` that is followed by what kind of permission is being granted, which part of local data is concerned and the specification of the permission. It is appropriate to grant as little permissions to the classes as it needs and this, in our case, is only one file called “MPISIMsaves”.

This can also be created automatically by running `policytool`. All you need to do is specify the important information to the programme and it will allow you to save it, including all these keywords, to a text file.

Having a Java policy file is not enough, however, as it must be consulted by JVM at startup for it to have an effect. There are two ways of doing this. One is by running the applet through appletviewer:

```
appletviewer -J-Djava.security.policy=<policy file> <html>,
```

where <policy file> is the file created for handling policies and <html> an html file containing the applet tag.

The other way is by running through Netscape or Internet Explorer, however, then this policy file has to be specified in a Java system file called “java.security”. It can be found at $JAVA-HOME/jre/lib/security/java.security. In this file it should be specified as:

```
policy.url.3=file:<policy file>
```

and should follow directly after other policy url fields (policy.url.1 and policy.url.2). As normal users do not have access to this file it is recommended to use the appletviewer. If saving and loading programmes is to be used within Netscape or Microsoft Explorer on the web then these steps should be followed by each user. This may be tedious but is a good way to prevent too much access being granted to unsafe applets.

It would still be a good idea to sign the applet if a certificate from a trusted company is obtainable. In the policy file a user can specify the certificate and so make sure that the applet on that location has not been tampered with.

More on certificates and policies can be found in [1].

4.4.3 Specifying the file to read or write

A single file can contain several different examples which means that one file should be enough for most users. The reason for saving the files this way is to allow the user to minimise applet’s permissions for accessing disk.

A file that contains the saved programmes can be specified in several ways. If no choice is made then, by default, the applet tries to use a file called “MPISIMsaves” in the working directory of the applet. The user can specify the file he or she wants to use in the settings frame. But the most useful way to specify the file is by putting it in the html as a parameter. This can be done like this:

```
<param name=''File'' value=x>,
```

where x is the name of the file intended for use. It can either be an URL or just a local name of the file on disk, relative to the working directory or full path. This tag has to be specified between the <applet> and </applet> tags in the html file. This makes it possible for a preprogrammed example to be put on the web. However, user can still choose a different file from the settings window but it is much more convenient for users if they only wish to use the file specified.

4.5 Changing the number of processes

The variability of the number of processes is important for the classroom demonstrations so that the functionality of a programme can be better expressed. It was decided that instead of constantly having three processes, the user could choose between two, three or four processes. It was considered that having even more processes would not add to functionality because the basic concepts could be explained with this much diversity anyway.

There are two ways in which we can choose the number of processes:

1. The first comes into play once the applet is loaded and starts executing its init method. The dialog window asks the user to choose the number of processes he or she wishes to use. This is done with

a combo box and that means that user is not allowed to choose any value different from two, three or four.

2. The other way is to specify it in the applet tag. This can be done with a param tag like this:

   ```
   <param name=''Number of processes'' value=x>,
   ```

   where x is the number of processes wanted. If the number of processes is specified in this way, no dialog window will appear. Note that should the value of the parameter not be acceptable the applet will ignore it and ask the user to input the number of processes at initialisation as described previously.

4.5.1 What was involved in the refactoring

The main refactoring to make the number of processes variable was done in the Envelope Box module. As was described previously, the Envelope Box module only displayed envelopes travelling in one direction of one fixed path from process to process at the beginning of this project. With the Bean changed to displaying envelopes travelling from and to the processes that were registered as neighbours of the envelope box canvases the only refactoring needed was to change the display of the applet. As the components are Beans that are as independent as possible only the layout needed to be changed.

However, the changes that were made through the entire duration of this project to work toward this end cannot be as easily listed. As there were a few suppositions that the number of processes would remain constant several parts of code needed changing.

4.6 Applet help

It was noted that the applet was not very intuitive to a user who is not familiar with MPI or does not know what it is. The problem was that the user was simply thrown into the applet and left to click on the buttons without there being any explanation as to how they are supposed to be used.

This was the last stage of the project and even though there have been interesting suggestions, such as creating a drag and drop interface for inserting commands, adding different help windows on all stages of the programming process, etc., they would involve quite a bit of refactoring and would perhaps take another week or two.

Due to the lack of time, only tool tip messages have been added and several windows describing the general purpose and concepts behind the applet. This may prove to be sufficient for better comprehensibility of the applet but there is still room for improvement.

5 Perceived problems and bugs

There were several problems and bugs within the project as it developed. Most of them were removed, however, some remain unresolved. These do not occur often and are not common, yet when they do appear the applet becomes unusable and has to be restarted. The following two are the ones that remain unsolved.

5.1 Thread problems

During the very first stage of development this problem became apparent. When designing the event model for the envelopes it might have been introduced or was there before but was not as common. The

effect is that at one trivial command or other all the threads suddenly stop for no obvious reason. Moving the mouse or giving some other input seems help and the threads continue execution normally.

As this problem seems to have gone away I suspect it might have been a synchronisation problem among the threads. Some conflicts may have arisen that had not been properly or quickly resolved. User input could therefore have resolved the conflict by raising the priority of a thread.

5.2 Anomalies with running on different platforms

Even though Java is the most portable language and Swing the best way to decrease visual platform dependency anomalies were encountered when testing on different Java Virtual Machines (JVMs). Sometimes windows and combo boxes are not displayed correctly. It may either be a window manager problem or a Java Virtual Machine problem as it cannot be a problem of the code because the applet code does not really handle the displaying of combo boxes other than saying they are a part of this or that container. Even setting the opaque flag, see [5], does not help.

These problems can seriously hinder the usability of the applet but cannot be removed as they are too Java dependent. My recommendation would be to try it and use it on the computer and JVM on which it works best.

6 Future recommendations

At the beginning of this project there were some interesting recommendations on how to improve the applet and even though quite a few have been realised still a few ideas remain unimplemented. These would further extend the usability but have not been realised due to the lack of time.

6.1 Dead-lock detection

Currently there is no dead-lock detection. The idea here is to be able to tell whether a dead-lock has occurred in an MPI programme through a circular dependency scheme. The programme should check whether a process is waiting on another process and that on another process and so on. If such a chain was found to lead back to itself it would mean that the processes have dead-locked.

In the point-to-point communications this would easily be implemented, however for collective communications it is much different and would probably take some time. For reasons of having a more consistent applet none of these have been implemented.

6.2 Better comprehensibility of the applet

Even though the applet is now more general and has help windows to familiarise the user with it there are still things that would have to be explained by a tutor. The interface, though effective, is also a bit unintuitive and this should be altered to make it as simple as possible but not simpler. Several suggestions have been made, such as drag and drop interface, detailed walk-through on how to set up and run a programme, etc., but were not implemented, again, due to lack of time.

6.3 Differences between Fortran and C

In the previous stages of development of the applet (before this project) it was noted that the applet seems to not conform to any standard, neither C nor Fortran. This problem is a bit tedious but could be quickly

inserted within the existing code. As this was not in the original plan of this project and there was other functionality to look into, it was left for future development of the applet.

7 Conclusion

At the beginning of the project the code had some bugs, was inconsistent and has strayed from the original design. In the first part of this project the existing problems were solved and in the second part new functionality was introduced. This has produced better code for future development as well as usable applet that can be used for demonstration purposes without encountering any serious deficiency in functionality.

Several problems have been discovered while using the applet across different platforms and these remain unsolved as it is unclear whether the fault lies within the applet (which is doubtful) or in improper installation of JVM on tested computers. Also scalability, while much better than before, is still not perfect as low resolutions do not tend to decrease the size of components.

Aside from these smaller difficulties the applet is ready to be used by the users and make the MPI learning curve easier.

References

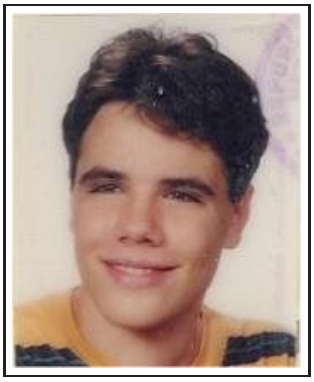
[1] Mary Dageforde. Security in Java 2 SDK 1.2. http://java.sun.com/docs/books/tutorial/security1.2/index.html.

[2] Tom Doel. Java Simulation of MPI Collective Communications. http://www.epcc.ed.ac.uk/ssp/1998/ProjectSummary/doel.html, September 1998.

[3] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, Jack Dongarra. MPI The Complete reference. The MIT Press, 1996.

[4] Derrick Pisani. Java MPI Simulator. http://www.epcc.ed.ac.uk/ssp/2000/ProjectSummary/pisani.html, September 2000.

[5] Robert Eckstein, Marc Loy, Dave Wood. Java Swing. O'Reilly and Associates, Inc, December 1998.



Rok Preskar is an undergraduate student at the Faculty of Computer and Information Science in Ljubljana and is entering fourth year of five year study course.
Email: rok.preskar@kiss.uni-lj.si.

Supervisors: Mario Antonioletti, Neil Chue Hong and Lindsay Pottage.