User Manual for ISP (In-Situ Partial Order)

A Formal Verification Tool for
Message Passing Interface (MPI) 1.0.x C Programs

Gauss Research Group
School of Computing
University of Utah
`http://www.cs.utah.edu/formal_verification`
Salt Lake City, UT 84112

Contents

1 OVERVIEW

ISP is a tool for formally verifying MPI programs. It can be used by anyone who can write simple MPI C programs, and requires *no special training* (if you can run an MPI C program using Unix tools for MPI, you can use ISP). ISP allows you to run your MPI C programs automatically without any extra efforts on your part (apart from compiling and making your examples) and flags deadlocks and MPI object leaks (see Section 4 and 9 for details; here we provide a brief summary). In addition, it helps you understand as well as step through *all relevant* process interleavings (schedules). Notice our use of the word 'relevant': even a short MPI program may have too many (an "exponential number") of interleavings. For example, an MPI program with five processes, each containing five MPI calls, can have well in excess of 10^{10} interleavings. However, ISP generates a new interleaving only when it represents a truly new (as yet unexamined) behavior of your program. As examples, (i) if an MPI program consisting of two processes issues point-to-point sends and non-wildcard receives to each other, then there is no observable difference (apart from performance or resource consumption) whether the sends are posted before each matching receive or vice versa; in this case, ISP will generate only *one* interleaving; (ii) if an MPI program consisting of three processes is such that the second process issues a sequence of wildcard receives, and the first and third process issue point-to-point sends to process two, then the exact non-deterministic choices of matches made at each step may affect the ensuing computation (including the conditional outcomes). In such programs, it is possible to force an *exponential* number of interleavings. In practice, here is how the results look (these examples come with the ISP distribution).

- For Parmetis [1], a 14,000+ line MPI program that makes 56,990 MPI calls for four ranks and 171,956 calls for eight ranks, ISP needed to examine only *one* interleaving!
- For many tests in the Umpire test suite [2], conventional MPI testing tools missed deadlocks despite examining many schedules. ISP determined a minimal list of interleavings to examine, finding deadlocks whenever they existed (see our table of results at the URL given under [3]).

- For Madre [4], a naïve algorithm – present in the *current release* of ISP – can result in $n!$ interleavings. An improved algorithm called POE_B in the upcoming release of ISP reduces this to *one* interleaving.

Much like with existing partial order [5, Chapter 10], ISP does not guarantee a minimal number of interleavings, although it comes pretty close to it.

2 OBTAINING ISP

ISP can be downloaded from `http://www.cs.utah.edu/formal_verification/ISP-release`. What you will receive are the full sources as well as the following examples in the directory `isp-tests`:

- 12 examples pertaining to the Game of Life (in directory `life`, courtesy of Bill Gropp of UIIC and Rusty Lusk of Argonne from their EuroPVM/MPI 2007 tutorials)
- 74 examples of the Umpire test suite (in directory `Umpire`), courtesy of Bronis de Supinski from LLNL.
- 12 examples under `others` developed by various authors.
- 16 examples under `madre-0.3` courtesy of Stephen F. Siegel (U. of Delaware) from his Madre release (see our PPoPP 2009 paper).
- The `Parmetis 3.1` hypergraph partitioner (see our PPoPP 2009 paper).

You can go to the directory `isp-tests` and simply type `make` to get all the tests compiled. Here are some of the requirements to run ISP:

Machine: ISP can be run on machines with one or more CPU cores. With more cores, ISP's OpenMP parallelization can help speed up model checking (see [6] for details).

Sockets: ISP can be run with TCP sockets or Unix-domain sockets. The latter is much faster, but requires all ISP runs to occur within one machine. A *distributed* ISP checker is in our future plans.

Operating System: Currently, only a Unix version is being released. Work is underway to have a Microsoft release, including a Visual Studio plug-in (the Microsoft release without the Visual Studio plug-in exists, and can be given out if you are interested).

MPI Library: The current ISP release is for MPICH 2.1. We have successfully tested ISP with OpenMPI as well as Microsoft's MPI (sometimes called CCS). An enhanced `configure` and `make` will be released corresponding to these MPI libraries also. If you are interested, kindly let us know and we can help tailor the current distribution with these other MPI libraries.

3 INSTALLING ISP

ISP can be installed by untarring the single tar.gz file in the above distribution, running `configure` and then `make`. You may also carry out other `make` options supported (e.g. `make install`).

- Be sure to install MPICH (or another tested MPI library) in the standard place. It is also possible to give information on non-standard paths when running `configure` (try and let us know if this works).
- Be sure that you can run `mpd`. Also, you must start `mpd` before you run ISP.
- Unpack the tarball, then within the directory `isp-0.1.0`, type `./configure`, then `make`, and then `make install` (prefix your commands with `sudo` as necessary). If all goes well, you will have the scripts `ispcc`, `isp`, and `ispUI` in your path. Check the contents of the `scripts` directory to know what all scripts are supported.

4 WHAT CAN YOU DO WITH ISP?

4.1 MPI commands supported

ISP works by "hijacking" the MPI scheduler, using the `PMPI` mechanism. The list of MPI commands that are explicitly trapped may be found out by visiting `profiler/isp.h`. Here is a summary of what you can use in your MPI program:

Trapped Commands: If your MPI C program contains a subset of the explicitly trapped MPI commands, then ISP tries to guarantee to do a sound backtracking search over the executions of your program, detecting bugs as explained earlier (details in Section 9). Clearly, there are disclaimers to this warranty (e.g., if the C code surrounding the MPI calls is designed to carry out strange side-effects, such as turning on the sprinkler system right above your heads, then a re-execution based model checking of your code isn't such a hot idea). *We will appreciate your notifying us of any strange effects you may observe, and also email us suggestions for improvement.*

The Use of Untrapped Commands: If your MPI C program uses more MPI functions than we explicitly trap, then please make sure that those MPI commands do not interact with the trapped ones in strange ways. For many MPI commands, one does not need to trap them: they may be directly sent into the MPI runtime. Examples of MPI commands that may need special handling are those relating to MPI I/O.

The Use of Threading: Currently, ISP is unaware of any threading that may be going on. Recall that MPI does not impart different ranks for different threads. ISP makes certain *completes-before* assumptions (IntraCB and InterCB) with respect to threading.

5 A COMPLETE EXAMPLE

Consider running the following example from the Umpire suite called `any_src-can-deadlock9.c`.

```
/* -*- Mode: C; -*- */
/* Creator: Bronis R. de Supinski (bronis@llnl.gov) Tue Aug 26 2003 */
/* any_src-can-deadlock9.c -- deadlock occurs if task 0 receives */
/*                            from task 2 first; sleeps generally */
/*                            make order 2 before 1 with all task */
/*                            0 ops being posted after both 1 and 2 */
/*                            same as any_src-can-deadlock5.c */
/*                            except tasks 1 and 2 are interchanged */

#include <stdio.h>
#include <isp.h>
#include <string.h>

#define buf_size 128

int
main (int argc, char **argv)
{
  int nprocs = -1;
  int rank = -1;
  char processor_name[128];
  int namelen = 128;
  int buf0[buf_size];
  int buf1[buf_size];
  MPI_Status status;
  MPI_Request req;

  /* init */
  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name (processor_name, &namelen);
  printf ("(%d) is alive on %s\n", rank, processor_name);
  fflush (stdout);

  MPI_Barrier (MPI_COMM_WORLD);

  if (nprocs < 3)
    {
      printf ("not enough tasks\n");
    }
  else if (rank == 0)
    {
      //sleep (60);

     MPI_Irecv (buf0, buf_size, MPI_INT,
                MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &req);

      MPI_Recv (buf1, buf_size, MPI_INT, 2, 0, MPI_COMM_WORLD, &status);
```

```
      MPI_Send (buf1, buf_size, MPI_INT, 2, 0, MPI_COMM_WORLD);

      MPI_Recv (buf1, buf_size, MPI_INT,
                MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);

      MPI_Wait (&req, &status);
    }
  else if (rank == 2)
    {
      memset (buf0, 0, buf_size);

      MPI_Send (buf0, buf_size, MPI_INT, 0, 0, MPI_COMM_WORLD);

      MPI_Recv (buf1, buf_size, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

      MPI_Send (buf1, buf_size, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
  else if (rank == 1)
    {
      memset (buf1, 1, buf_size);

     // sleep (30);

      MPI_Send (buf1, buf_size, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

  MPI_Barrier (MPI_COMM_WORLD);

  MPI_Finalize ();
  printf ("(%d) Finished normally\n", rank);
}

/* EOF */
```

5.1 Compiling programs for ISP

The set of test suites that accompanies the ISP distribution contains makefile to produce the binaries.The ISP installation script when installing ISP prepares the binary for the programs available in the test suites to run under ISP. To compile programs without a makefile `ispcc` script can be used.For example, the command to compile a program mpi-filename.c is

```
ispcc -o profiled_executable mpi-filename.c
```

profiled_executable is a normal executable file except PMPI has been incorporated.

5.2 ISP Command Line Options

After installing ISP and making the tests, if you type `isp`, you will see the following flag options:

```
usage:  isp.exe [ -<flag> [<val>] | --<name>[{=| }<val>] ]... \
profiled_executable [executable args]
  Flg Arg Option-Name    Description
   -n Num numprocs       Number of processes
   -p Num port           Listening port
   -h Str host           Hostname where ISP resides
   -x no  us             Use unix sockets
   -b no  block          Treat sends as blocking w/o buffering
   -g no  noblock        Sends use buffering of MPI subsystem (default)
   -l Str logfile        Location of interleaving log for UI
   -m no  mpicalls       Output # mpi calls trapped per rank
   -O no  verbose        Always output all transition lists
   -q no  quiet          Output no information save return code
   -r Num rpt-progress   Output progress every n MPI calls
   -d no  distributed    Use to launch profiled proc manually
   -f no  fibopt         Enables irrelevant barrier detection
   -e KWd exp-mode       Choose which interleavings to explore [all, random, left_most]
   -s no  env            Use envelope only
   -y Num exp-some       Explore random K choices at non-determ. points
   -z T/F stop-at-deadlock When enabled, ISP will stop at deadlock
   -v opt version        Output version information and exit
   -? no  help           Display usage information and exit
   -! no  more-help      Extended usage information passed thru pager

Options are specified by doubled hyphens and their name
or by a single hyphen and the flag character.

exit codes:
      0    Model checking complete with no errors detected.
    1    Model checking complete with deadlock detected.
    2    ISP help displayed.
    3    A process failed an assertion.
    4    A process exited with MPI_Abort.
    5    A process called an MPI function with an invalid rank.
    11   Unable to start profiled MPI program with mpiexec.
    12   Unable to use WINSOCK.
    13   Unable to open socket.
    14   Unable to bind to socket.
    15   Unable to listen on socket.
    16   Unable to accept connections on socket.
    17   Error reading from socket.
    20   Unable to open specified log file.
    21   Transitions different between interleavings.
    22   Received an empty buffer. (Profiled code might have crashed.)
```

We hope these flag options are self-explanatory. Please email us if any are unclear.

5.3 Running ISP

A terminal session is enclosed. `ispUI` reads the log file generated by `isp`. We describe the format of this log file in Section A.

```
isp -n 3 -f -l any_src-can-deadlock9.log ./any_src-can-deadlock9.exe
ISP - Insitu Partial Order
----------------------------------------
Command:        ./any_src-can-deadlock9.exe
Number Procs:   3
Server:         localhost:9999
Blocking Sends: Disabled
FIB:            Enabled
----------------------------------------
Started Process: 6634
(0) is alive on ganesh-desktop
(1) is alive on ganesh-desktop
INTERLEAVING :1
(2) is alive on ganesh-desktop
Started Process: 6641
(1) Finished normally
(0) Finished normally
(2) Finished normally
(0) is alive on ganesh-desktop
(1) is alive on ganesh-desktop
INTERLEAVING :2
(2) is alive on ganesh-desktop

Transition list for 0
0 1 0 0 Barrier any_src-can-deadlock9.c:36 1{[0, 1][0, 2]} {}
1 4 1 0 Irecv any_src-can-deadlock9.c:47 -1 0 1{[0, 2]} {} \
     Matched with  process :2 transition :1
2 7 2 0 Recv any_src-can-deadlock9.c:49 2 0{} {}


Transition list for 1
0 2 0 1 Barrier any_src-can-deadlock9.c:36 1{[1, 1][1, 2]} {}
1 5 1 1 Send any_src-can-deadlock9.c:74 0 0{} {}
2 8 2 1 Barrier any_src-can-deadlock9.c:77 2{} {}


Transition list for 2
0 3 0 2 Barrier any_src-can-deadlock9.c:36 1{[2, 1][2, 2]} {}
1 6 1 2 Send any_src-can-deadlock9.c:62 0 0{} {}         Matched with  process :0 transition :1
2 9 2 2 Recv any_src-can-deadlock9.c:64 0 0{} {}

No Matching found!!!
Deadlock Detected!!!
Killing program any_src-can-deadlock9.exe
application called MPI_Abort(MPI_COMM_WORLD, 1) - process 0[cli_0]: aborting job:
application called MPI_Abort(MPI_COMM_WORLD, 1) - process 0
ganesh@ganesh-desktop:~/ISP/isp-0.1.0/isp-tests/Umpire$ application \
     called MPI_Abort(MPI_COMM_WORLD, 1) - process 1[cli_1]: aborting job:
application called MPI_Abort(MPI_COMM_WORLD, 1) - process 1
application called MPI_Abort(MPI_COMM_WORLD, 1) - process 2[cli_2]: aborting job:
application called MPI_Abort(MPI_COMM_WORLD, 1) - process 2
rank 0 in job 3  ganesh-desktop_45900   caused collective abort of all ranks
  exit status of rank 0: killed by signal 9
```

5.4 Viewing the results using ispUI

The Java GUI for ISP can be fired using the following command

```
ispUI [file.log]
```

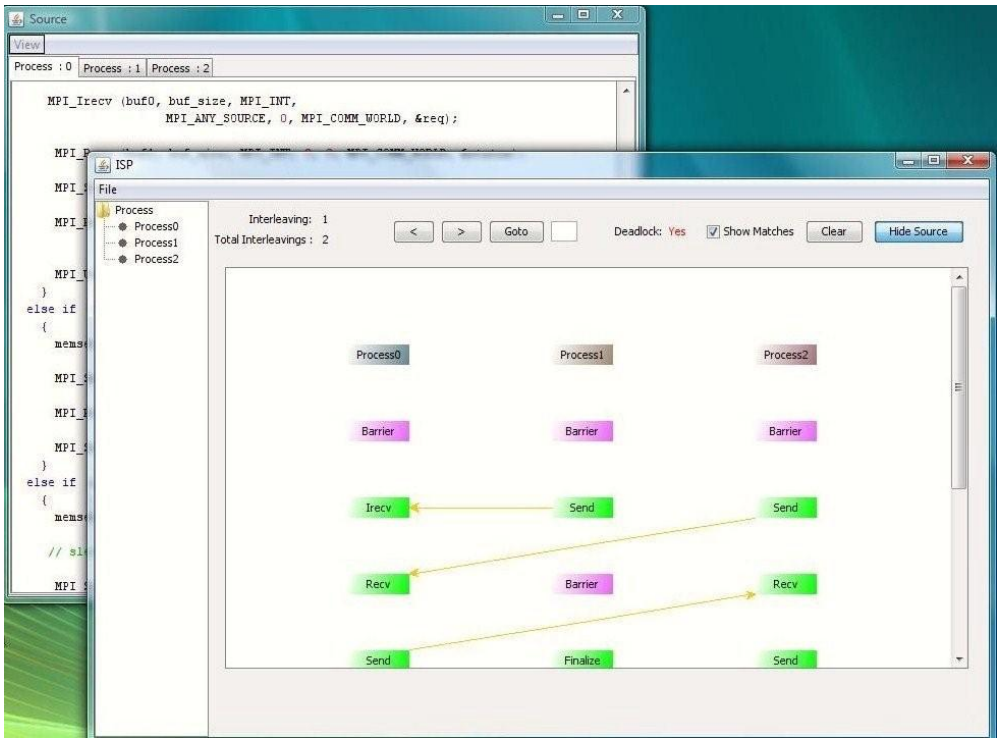


Figure 1: Screenshot of ispUI - showing non-deadlocking interleaving for the example `any_src-can-deadlock9.c`

The above example generated two interleavings, one of which results in a deadlock. The ispUI screen-shots are shown in Figure 1 and Figure 2.

Briefly, ispUI allows you to load a log file and examine the MPI calls made during the execution.

Some of the convenient features of the ispUI include

> and < buttons: These buttons on the top allows one to step through the different interleavings explored by ISP

Goto: Allows user to directly move to the specified interleaving

Show Source: Brings up the source code of the MPI program tested under ISP. The source code can be examined in two modes, tabbed and split mode. A *view* menu is present on top of the source window that lets one switch between these modes.

ToolTips: For a quick info about a MPI call, one can mouse over any rectangle and the UI displays info on the MPI call as a tool tip.

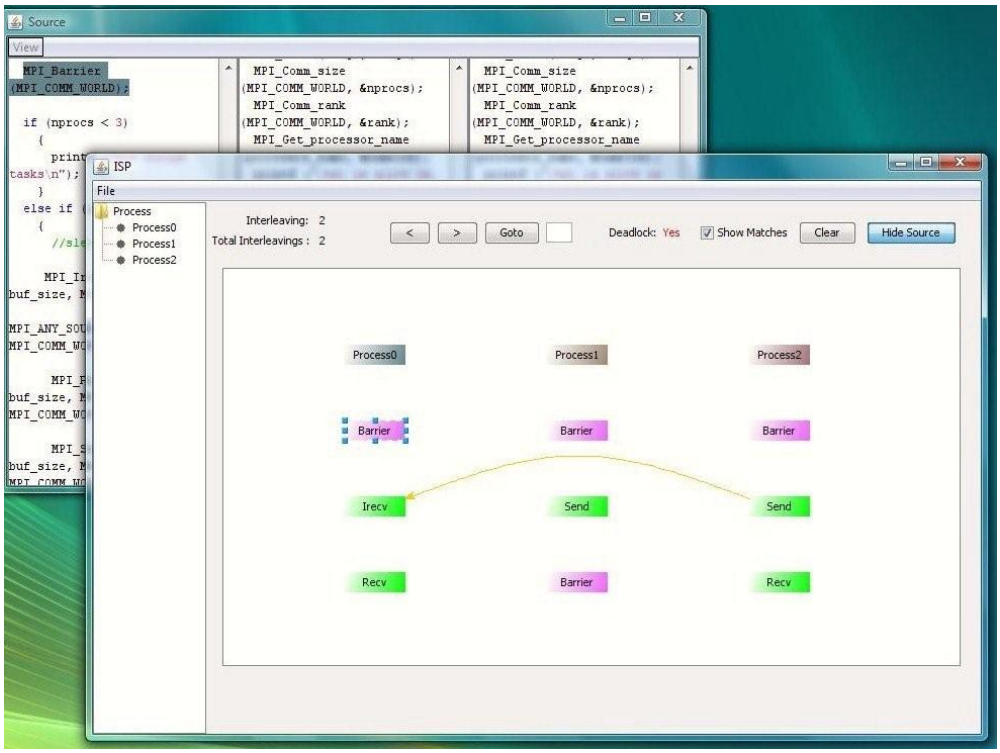


Figure 2: Screenshot of ispUI - showing deadlocking interleaving for the example `any_src-can-deadlock9.c`

CB Edges: ispUI allows one to view intraCB edges or interCB edges for one or more MPI calls. One can choose multiple rectangles (MPI calls) and right click for viewing options. Also, to view intraCB edges of all the MPI calls in a process one can choose the desired process displayed as tree on the left and right click for options.Please see our papers for details on these edges and what they signify. Briefly, these edges signify the *completes-before* relation followed by the MPI calls in the given MPI program

Source Highlight: ispUI automatically highlights the source code corresponding to each MPI call on the source window as the user selects one or more rectangles on the main UI window.

6 Running ISP with Eclipse

The Eclipse plugin, The Graphical Explorer of MPI Programs (GEM), is the most up-to-date version of visualizing and interacting with ISP. Information on its use and installation can be found on our website:

7 Running ISP with Microsoft Visual Studio 2008

7.1 Download and Installation

Required software:

- Microsoft Visual Studio 2008 (Standard or Professional)
- Microsoft Compute Cluster Pack SDK from here. Another MPI implementation will most likely work; however, the plug-in has only been tested against the mentioned MS MPI implementation.
- The Visual Studio ISP plug-in.

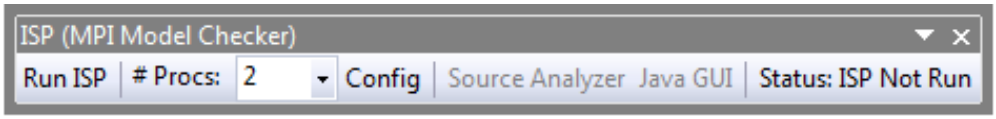
Installation instructions:

- Install Microsoft Visual Studio 2008.
- Install Microsoft Compute Cluster Pack SDK.
- Install the ISP plug-in MSI file. In the installation, "ISP Visual Studio plug-in" is all that is needed. The "MPI template for Visual Studio" is a Visual Studio wizard that sets up a project with the correct MPI include and libraries, so starting an MPI project is easier. Installation of the MPI template is recommended.

7.2 Creating a New MPI Project

First, open Visual Studio 2008. Select *File*, *New*, and click *Project*. If you select the *Visual C++* option on the left, the newly installed MPI template (*MPI Project*) should show up on the right. Name the project and press *OK*. It is okay to accept all of the default options, and then press *Finish*. A sample MPI program has now been created, ready for you to start programming. (If you did not install the MPI template, you must set the correct MPI *INCLUDE* and *LIBRARY* paths in the Project Properties.

The new ISP toolbar should have also popped up. This is the toolbar that will allow you to interact with ISP in the Visual Studio environment.



Run ISP: Runs the ISP model checker with the specified number of processes, on the program specified via the *Config* option.

Number Processes Specifies how many processes ISP should model check the program with.

Config: Allows configuration of which projects should be recompiled to link against the ISP libraries, and what the StartUp project should be. The StartUp projects and projects that depend on other checked projects are automatically selected. You can only select VC++ projects, since ISP does not support other languages.

Source Analyzer: This is a Visual Studio exclusive user interface which visually displays the output that ISP generated by highlighting lines in the source file. It shows both the current MPI call, and any matching point-to-point or collective operation.

Java GUI: This launches the ispUI Java GUI discussed in Section 5.4.

Status: This is the status of current ISP operations. Clicking on the button will give you more information, such as ISP run time, number of interleavings, etc.

7.3 Running ISP on an Example Program

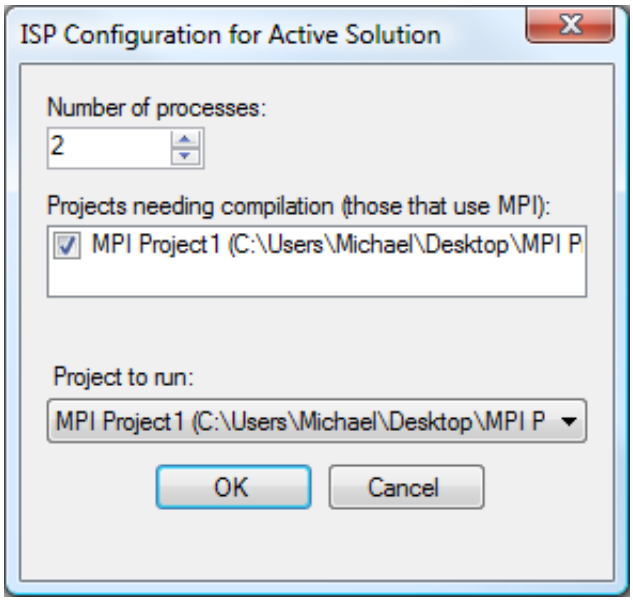
For the rest of this tutorial, I will use the following MPI program (which is the any_src-can-deadlock9.c example from the Umpire test suite):

```
/* -*- Mode: C; -*- */
/* Creator: Bronis R. de Supinski (bronis@llnl.gov) Tue Aug 26 2003 */
/* any_src-can-deadlock9.c -- deadlock occurs if task 0 receives */
/*                            from task 2 first; sleeps generally */
/*                            make order 2 before 1 with all task */
/*                            0 ops being posted after both 1 and 2 */
/*                            same as any_src-can-deadlock5.c */
/*                            except tasks 1 and 2 are interchanged */

#include <stdio.h>
#include <string.h>
#include <mpi.h>

#define buf_size 128

int
main (int argc, char **argv)
{
  int nprocs = -1;
  int rank = -1;
  char processor_name[128];
  int namelen = 128;
  int buf0[buf_size];
  int buf1[buf_size];
  MPI_Status status;
  MPI_Request req;

  /* init */
  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name (processor_name, &namelen);
```

```
  printf ("(%d) is alive on %s\n", rank, processor_name);
  fflush (stdout);

  MPI_Barrier (MPI_COMM_WORLD);

  if (nprocs < 3)
    {
      printf ("not enough tasks\n");
    }
  else if (rank == 0)
    {
      //sleep (60);

     MPI_Irecv (buf0, buf_size, MPI_INT,
         MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &req);

      MPI_Recv (buf1, buf_size, MPI_INT, 2, 0, MPI_COMM_WORLD, &status);

      MPI_Send (buf1, buf_size, MPI_INT, 2, 0, MPI_COMM_WORLD);

      MPI_Recv (buf1, buf_size, MPI_INT,
        MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);

      MPI_Wait (&req, &status);
    }
  else if (rank == 2)
    {
      memset (buf0, 0, buf_size);

      MPI_Send (buf0, buf_size, MPI_INT, 0, 0, MPI_COMM_WORLD);

      MPI_Recv (buf1, buf_size, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

      MPI_Send (buf1, buf_size, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
  else if (rank == 1)
    {
      memset (buf1, 1, buf_size);

     // sleep (30);

      MPI_Send (buf1, buf_size, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

  MPI_Barrier (MPI_COMM_WORLD);

  MPI_Finalize ();
  printf ("(%d) Finished normally\n", rank);
}

/* EOF */
```

The output from the ISP model checking is shown in Visual Studio's *Output* window. If you don't have the *Output* window enabled, you can enable it by selecting *Debug*, *Windows*, and then *Output*.

You must now change the number of processes to run the program with to **3**, since that is what the program requires. (You can see the program will only print an error and do nothing interesting with less than 3 processes.) The process count can be changed with the *Config* button or by changing the number on the toolbar. I will use the *Config* button, so I can give a short overview.



The *Config* dialog pops up and lets you select from a number of options. ISP needs to insert itself in to the program by "hooking" the MPI calls, and in-turn, calling the corresponding PMPI call when the scheduler tells it to do so. Therefore, the project needs to be compiled against the ISP static library. This is done automatically, so all you must do is select which projects use MPI, so they can be re-compiled when ISP is run. The Visual Studio project dependencies for each project are automatically selected for you, along with the project selected in "Project to run". The "Project to run" is from which project ISP should launch the output executable with *mpiexec*.

After selecting **3** processes, click *OK* to close the dialog. The number of processes, along with the projects to compile and run, are stored in the project file, so these values will be restored if the project is closed and reopened.

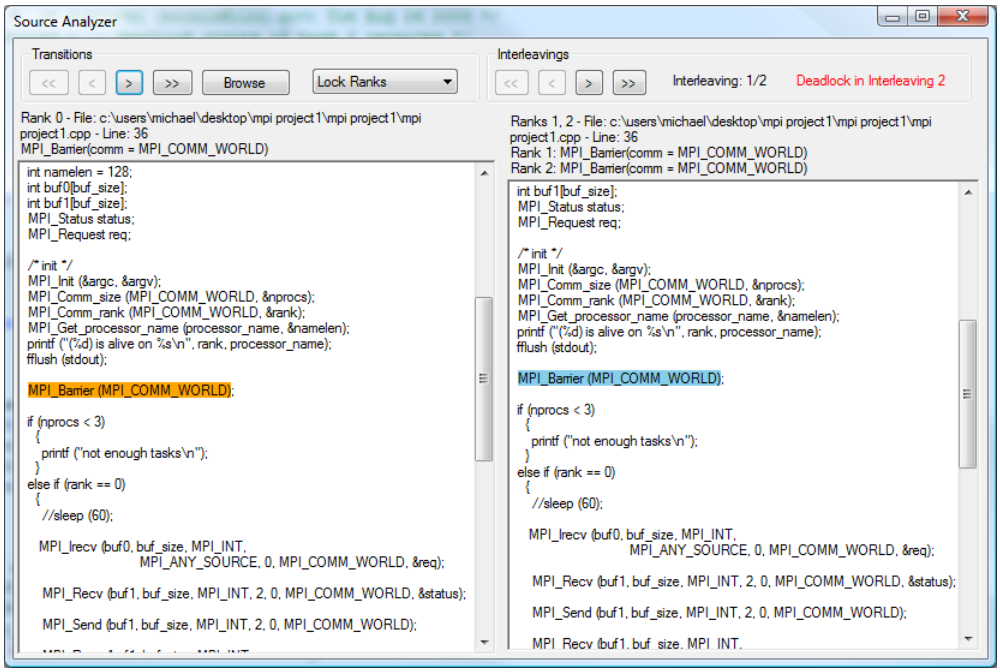
Finally, you can run ISP on the project. After clicking the *Run ISP* button, you are presented with a message stating that the project will be re-compiled and re-linked. If you wish, you are able to disable this dialog by selecting the *Tools* menu, *Options...*, select *ISP* and *General* on the left. Finally, set "Show Compile Dialog" to *False*. It should be noted that this dialog shows the plug-in options that can be configured. So if you want to change some ISP preferences, it can be done here.

When you click the *Run ISP* button, you will notice that the project is recompiled, and the ISP output is displayed in the *Output* window. The status field on the toolbar has

changed to say that a deadlock was detected. The Visual Studio status bar also contains a message that indicates the same fact.

7.4 Visualizing ISP Output with the Source Analyzer

To visualize ISP's output, the *Source Analyzer* can be used. It can be launched from the button on the toolbar.

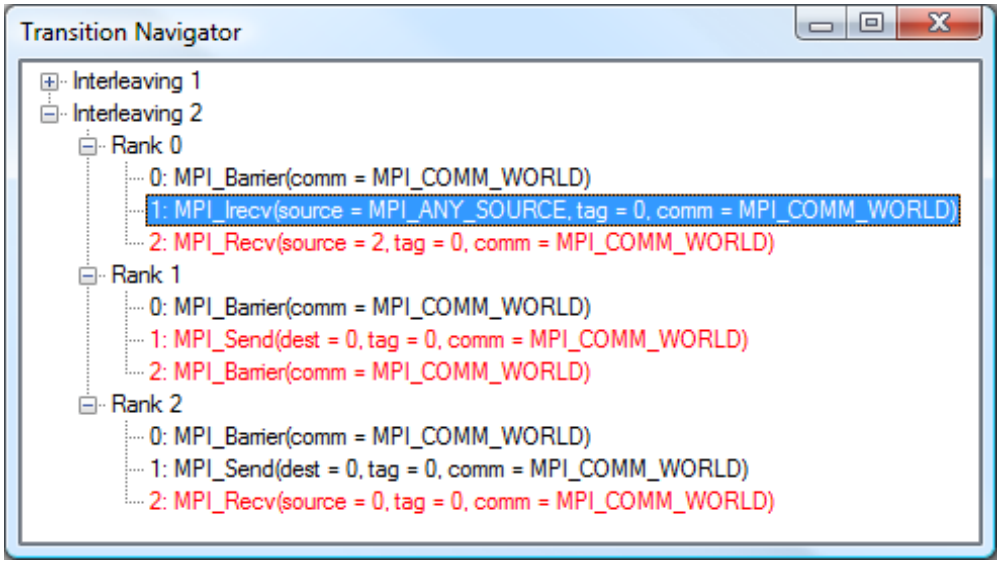


This user interface can be used to step through all of the transitions (MPI calls) that ISP executed, while highlighting the current transition in a display of the program's source code. The highlight on the left shows the current transition. The highlight on the right shows the matching point-to-point operations for point-to-point calls, or the line(s) that the other ranks in the communicator called for collective calls. There is a legend at the top of the window, which tell you what the highlight colors mean. (If you don't see that second, make the *Source Analyzer* window bigger, and it should show up.)

You can use the buttons at the top to step through the transitions and interleavings, or to step to the first / last transition / interleaving. The transitions in an interleaving can be ordered in one of two ways (program order or issue order). The program order is the order in which the profiled code sent the ISP scheduler information about the MPI calls. The issue order is the order in which MPI told the profiled code to actually execute the call, and send it in to the MPI runtime with the PMPI mechanism. Switching between these two modes can be done in the *Tools*, *Options...* dialog with the "Show MPI calls in issue order" property. (Note that the Source Analyzer needs to be restarted for changes to take effect.) In deadlock situations, some MPI calls that were sent to the scheduler might not have been issued, since ISP determined that there was a deadlock, and terminated the program before

these instructions were issued. These calls are highlighted in *red*, and no matching calls are shown on the right half of the screen.

When stepping through the transitions, it will switch between the multiple ranks of the program. You can set the *Lock Ranks* option to only step to the previous and next call of a specific rank, instead of all the ranks.



Clicking on *Browse* will bring up a list of all MPI calls for each of ISP's interleavings. Unissued instructions are highlighted in *red* in this window, so it is easy to see which instructions might have been involved in the deadlock. Double clicking on any MPI call will display that MPI call in the *Source Analyzer* window.

8 ISP on Raven Server

To run ISP on raven-srv

- Be sure to have followed the instructions from `http://www.cs.utah.edu/formal_verification/mediawiki/index.php/Raven_Cluster_Tutorial` on using the cluster
- ssh into raven-srv using the command

  ```
  ssh -Y uname@raven-srv.cs.utah.edu
  ```

- Start mpd with the command

  ```
  mpd &
  ```

- Refer to Section 5.1 for compilation instructions
- Refer to Section 5.2 and Section 5.3 for running ISP
- Refer to Section 5.4 to bring up the Java GUI for ISP.

9 ALGORITHMS, LIMITATIONS, FUTURE PLANS

Here is a short summary:

- The algorithms behind ISP are only partly explained by our papers (the real details are often involved, and may be understood only by studying our code as well as talking to us). All our papers after the PPoPP 2008 papers describe the POE algorithm.
- The POE_B algorithm will intelligently handle buffer allocation, dealing with MPI's slack inelasticity. In general, one needs to study every MPI communication with/without buffering. The POE_B algorithm will avoid this exponential search as will be reported in a future publication.

10 ACKNOWLEDGEMENTS

10.1 Funding Acknowledgements

We gratefully acknowledge the following sources of funding:

- Microsoft, under the HPC Institutes program
- NSF, under grant CNS-0509379

10.2 Students and Fellow Researchers

Credits for ISP trace back to Rajeev Thakur's (Argonne) very astute suggestion "...but you can run an MPI program and trap its MPI calls using PMPI, and replay the executions" made in 2006 when he visited the University of Utah. Since then our group has done a lot with his suggestion, a true appreciation of which is obtained by consulting our webpage and going through our publications. Student credits for ISP are roughly as follows (apologies for any inaccuracies or omissions):

- Salman Pervez, with Robert Palmer's help, wrote the initial version of the ISP (EuroPVM/MPI 2007) for his MS thesis. This version of ISP supported mainly the MPI 2.1 one-sided operations, plus a handful of other MPI operations. It did not incorporate the POE algorithm and this version was discontinued after Salman's MS thesis.
- Sarvani Vakkalanka, for her PhD, developed, over several implementations (see her publications in PPoPP 2008, CAV 2008, EuroPVM/MPI 2008, PADTAD 2008) several versions of ISP. She developed both the POE algorithm (CAV 2008) and the POE_B algorithm (to be available in future).
- Anh Vo and Michael DeLisi have engineered ISP to its current form, working closely with Sarvani (see our PPoPP 2009 paper). Anh, for his PhD, continues to push ISP towards handling large examples.

- Anh Vo implemented the OpenMP parallelization of ISP, and did the Parmetis and Madre case studies (see our PPoPP 2009 paper).

- Michael helped in every aspect of our project, including developing our Windows version, creating the configure and Make scripts, developing automatic regression testing methods, etc. etc. He also converted from TCP sockets to Unix-domain sockets, speeding up ISP significantly (by several fold).

- Subodh Sharma made significant contributions in testing ISP in its early stages (co-authorship in EuroPVM/MPI 2008). He wrote the initial version of the FIB algorithm (see his EuroPVM/MPI 2008 paper).

- Sriram Aananthakrishnan, working under Sarvani's guidance, created the graphical user interface of ISP. Initially we were considering porting Basile Schlich's code from his EuroPVM/MPI 2008 paper (sincere thanks to Basile for sharing the code with us). However, after studying this code, Sriram and Sarvani adopted some of the ideas, but coded the present UI from scratch.

- Although not central to ISP so far, Guodong Li has worked on formalizing a significant proportion of MPI in TLA+. His work is available online on our webpages (see our PPoPP 2008 paper on the semantics of MPI, and the accompanying journal paper under construction).

- Geof Sawaya's contributions to our MPI effort are gratefully acknowledged. He has made impressive contributions, performing an objective study of the success of the DARPA/Intel tests on the Microsoft CCS platform which are available for viewing here.

We also acknowledge with thanks all the feedback and encouragement provided to us by Bill Gropp (UIUC), Rusty Lusk (Argonne), Erez Haba, Dennis Crain, Shahrokh Mortazavi, and Robert Palmer (all of Microsoft), Stephen Siegel (Delaware), and George Avrunin (U. Mass.).

A Log File Format

ISP generates a log file summarizing the results of a verification run (through the command `isp`. The log file contains information on the number of interleavings, the MPI calls that were trapped by the scheduler, and information on the send and receive MPI calls that were matched in an interleaving. The log file also contains data regarding interCB and intraCB edges corresponding to each MPI call. ISP also outputs some useful information such as the line number and the filename for every trapped MPI call to the log file. All these pieces of information are used by the `ispUI` command in portraying ISP's execution graphically. In this section we document the log file format for use by other analysis tools. We use the example code provided in MPI_TwoDeadlocks.c as given here:

```
#include <mpi.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <pthread.h>

int main(int argc, char** argv) {
int data, nprocs, rank;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
if (nprocs < 4) {
printf("This test needs more than 4 processes\n");
exit(1);
}

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 1) {
MPI_Recv(&data, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Recv(&data, 1, MPI_INT, 2, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Recv(&data, 1, MPI_INT, 3, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else if (rank == 0) {
MPI_Ssend(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
MPI_Ssend(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

} else if (rank == 2) {
MPI_Ssend(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (rank == 3) {
MPI_Ssend(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
}

MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
return 0;
}
```

A.1 Overall Contents

The very first line of the log file tells us the **number of processes in the MPI program**. If the last line of the log file contains the keyword **DEADLOCK**, then the MPI program deadlocked in the last interleaving. In the absence of the keyword DEADLOCK, the MPI program terminated normally without any deadlocks.

A.2 Specific Items in the log file

- An entry for a MPI call trapped by the scheduler always begins with the **interleaving number**. For example, the entry for very first MPI call in the log file begins with '1' indicating that the MPI call corresponds to first interleaving. All entries for MPI calls that has 1 as the first token belongs to the first interleaving and those entries having 2 as the first token correspond to second interleaving and so on.

- The interleaving number of the last MPI call tells us the **total number of interleavings** explored by ISP for the MPI program.
- To identify an MPI call within an interleaving, **the process rank** and the **indices** are used. The four tokens following the interleaving number in the log file are the process rank and indices: order envelope sent by proc (0-based), order scheduler receives (1-based), and order scheduler sends (1-based). The process rank is the MPI rank of the process that issued this MPI call to the runtime and the index is the order in which this process made the MPI calls.

Example: *Consider the following entry in a log file:*

```
1 1 2 8 6 Recv 2 0 0_0:1:2:3: { 3 } { [ 2 0 ] [ 2 1 ] [ 1 2 ] } Match: 2 0 File: 18 MPI_TwoDeadlocks.c
```

This entry corresponds to a MPI function Recv that belongs to first interleaving issued by process with rank 1 and this is the third call issued by the process (as the first two calls would have been indicated by indexes 0 and 1 – i.e.*, counting begins at 0). It is the eighth one received by the scheduler (seven more must have been sent by other processes before this). It is the sixth one sent by the scheduler (five others must have been scheduled before this).*

The next token that follows the index number is the function name of the MPI call. For example, if MPI_Recv has been issued by the process then the name token for the MPI call is Recv. If the MPI call is MPI_Comm_create then the name token is Comm_create. In the example above, one can see the token `Recv` as the function name of the MPI call.

For all types of send and receive that MPI supports (such as blocking, non-blocking and synchronous, asynchronous), ISP outputs three tokens following the function name of the MPI call:

1. The first token being **src_or_dst**,
2. the next token being the **message tag**, and
3. the third token being the **communicator** followed by an underscore, then a colon separated list of com-number-members.

If the MPI call is a receive, then the src_or_dst corresponds to the source of the receive. (in other words, the process that issued the matching send). This token takes the value **-1** if the source id of the Recv is given by **MPI_ANY_SOURCE**. If the MPI call is a **send**, then **src_or_dst** corresponds to the **destination process**. If a MPI_Send() matched a **receive in process 2**, then the value for the pp src_or_dst token would have been **2**.

ISP differentiates the communicators used by the MPI calls with numbers. The communicator token will have the value **0** for **MPI_COMM_WORLD**. For all other communicators that were used, ISP identifies them by numbers starting with **1**.

For MPI calls like Barrier, Gather, AllReduce etc. (*i.e.*, calls other than send or receive), ISP outputs a single token following the function name of the MPI call, which is, then, the communicator token.

A.3 CB Edges

The communicator token is followed by the intraCB edge set enclosed within { }, where the values inside the parentheses are indices of function calls. For instance, consider again the following entry:

```
1 1 2 8 6 Recv 2 0 0_0:1:2:3: { 3 } { [ 2 0 ] [ 2 1 ] [ 1 2 ] } Match: 2 0 File: 18 MPI_TwoDeadlocks.c
```

The intraCB edge set for the above entry contains the index of 3. This means that the MPI call with index 3 has an intraCB edge from the above Recv. Please note that the intraCB is a completes before relation within a process. Therefore, the MPI call with index 3 belongs to the *same process as the above entry.*

The interCB edge set follows the intraCB edge set and it is enclosed within { } as with the intraCB edge set. For the interCBs, we need the both the process rank and the index. This pair is enclosed within [ ] and the interCB edge set is a collection of these pairs { }. For example, consider the entry,

```
1 0 1 5 3 Ssend 1 0 0_0:1:2:3: { 2 } { [ 1 1 ] [ 1 2 ] [ 0 1 ] } Match: 1 1 File: 18 MPI_TwoDeadlocks.c
```

This MPI_Ssend function call has three interCB given by the pairs [1 1], [1 2], and [0 1]. This first one means that the there is an interCB edge to the MPI function call with

- process rank 1 and
- function call index 1

from the above MPI_Ssend.

A.4 Match Tokens

The token **Match** follows the interCB edge set. If the MPI call is a send or receive, then two tokens follow **Match**, and these are the process rank and the index of the matching MPI call. For all other MPI calls which do not have the notion of a match, ISP outputs -1 and -1 following the Match token.

A.5 File and Line Numbers

The ISP also outputs some useful information about each MPI call such as the file name and the line number corresponding to that MPI call. The **File** token is followed by two tokens, the first one being the length of the file name and the actual file name. The token following the file name is the line number. Let us consider an example,

```
1 0 2 7 9 Barrier 0_0:1:2:3: { 3 } { [ 1 4 ] [ 2 1 ] [ 3 1 ] [ 1 5 ] [ 0 2 ] [ 2 2 ] [ 3 2 ] } Match: -
```

This MPI call corresponds to interleaving 1 issued by a process 0 and it is the third call that has been issued by process 0 (as its index is 2). The function call is MPI_Barrier() and the communicator is MPI_COMM_WORLD since the communicator token following Barrier is 0. The ranks in this communicator are all 4, so we see them listed. This MPI function has one intraCB edges to function call 3 of process 0. It also has many interCB edges, the first one being from process 1. This function calls does not have any matches and the hence the tokens following **Match** are -1. The file name contains 18 characters and the file name follows the token "18." The line number corresponding to this call is 34 in the file MPI_TwoDeadlocks.c.

A partial bibliography is provided below (for a full list of papers, kindly see our group webpage).

References

[1] ParMETIS - Parallel graph partitioning and fill-reducing matrix ordering. .

[2] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*. IEEE Computer Society, 2000. Article 51.

[3] Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*. Springer, 2008. Benchmarks are at .

[4] Stephen F. Siegel. The MADRE web page. , 2008.

[5] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[6] Anh Vo, Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby, , and Rajeev Thakur. Formal verification of practical mpi programs. In *Principles and Practices of Parallel Programming (PPoPP)*, 2009. Accepted.