Abstract

In this report will be presented and analysis of the Midterm Exam by Stephen F. Siegel [1] through the use of ISP (In-Situ Partial Order) Tool. In this exam there are some C/MPI-based programs that point out some particular situations of the MPI routines like MPI_Send, MPI_Recv, MPI_reduce, etc., that bring about deadlock, etc.. Besides, there are other examples that show other behaviors of MPI_Wait and non-blocking operations.

1 Analysis

In this report we present an analysis of running some MPI example programs on ISP. They are simple programs that show how ISP handles deadlocks and the number of interleavings. I will present an analysis of the example Code and ISP’s output for it. These examples are from Steve’s MPI Midterm Exam (Appendix A).

2 Steve’s MPI Exam Examples

2.1 Example 19 - p. 6

In the example 19 there are 3 processes, each one performs a blocking operations (MPI_Send or MPI_Recv) or both. Running ISP, it points out only 2 interleavings and nothing deadlock.

```c
#include <stdio.h>
#include <stdlib.h>
#include <isp.h>

int main(int argc, char **argv) {
    int rank, x, y;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        x = 10; y = 11;
        MPI_Send(&x, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
        MPI_Send(&y, 1, MPI_INT, 2, 9, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&y, 1, MPI_INT, MPI_ANY_SOURCE, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf(“%d %d
”, x, y);fflush(stdout);
    } else if (rank == 2) {
        x = 20;
        MPI_Recv(&y, 1, MPI_INT, 0, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(&x, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```
Looking the pictures below we see that in the first interleaving the 2 \textit{MPI\_Recv} of process 1 matches respectively with the first \textit{MPI\_Send} of process 0 and with the \textit{MPI\_Send} of the process 2.

On the other hand, the second interleaving showed that (for some reason, maybe delay) the first \textit{MPI\_Recv} of the process 1 can match before with the \textit{MPI\_Recv} of the process 2 and then the second \textit{MPI\_Recv} matches with the \textit{MPI\_Send} of process 0.
Neither of these interleavings deadlock so we can rest assured that this will never happen. The results will however vary slightly. The output will be 10 20 with the first interleaving and 20 10 with the second interleaving.

### 2.2 Example 20 - p. 7

In this example there are 3 processes and blocking operations.

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <isp.h>

int main(int argc, char **argv) {
    int rank, x, y;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("Process %d
", rank);
    if (rank == 0) {
        x = 10; y = 11;
        MPI_Send(&x, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
        MPI_Send(&y, 1, MPI_INT, 2, 9, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, 11, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&y, 1, MPI_INT, MPI_ANY_SOURCE, 10, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("%d %d\n", x, y);
        fflush(stdout);
    } else if (rank == 2) {
        x = 20;
        MPI_Recv(&y, 1, MPI_INT, 0, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(&x, 1, MPI_INT, 1, 11, MPI_COMM_WORLD);
    }
}
```
Running ISP in blocking mode (which has no buffering) the program always deadlock and ISP points this out. This is because the first \texttt{MPI\_Send} of the process 0 would matches with the second \texttt{MPI\_Recv} of process 1, but the first \texttt{MPI\_Recv} of process 1 would match with \texttt{MPI\_Send} of process 2 that is called after a \texttt{MPI\_Recv} would match with second \texttt{MPI\_Send} of process 0. All these \texttt{MPI\_Send} and \texttt{MPI\_Recv} are blocking so the first \texttt{MPI\_Send} of process 0, the first \texttt{MPI\_Recv} of process 1 and the \texttt{MPI\_Recv} of process 2 are all waiting for a match that will never arrive. So it’s a deadlock, and ISP detects it only in blocking mode. Below we can see ISP’s output and the graphic which also confirms that there is always a deadlock.

\begin{verbatim}
ISP - In situ Partial Order
 Command: ./a.out
 Number Procs: 3
 Server: localhost:9999
 Blocking Sends: Enabled
 FIB: Enabled

 Started Process: 12508
 Process 1 Process 0 Process 2
 rank 2 in job 13 Simone-MacBook_34805 caused collective abort of all ranks
   exit status of rank 2: return code 1
 rank 1 in job 13 Simone-MacBook_34805 caused collective abort of all ranks
   exit status of rank 1: return code 1
 rank 0 in job 13 Simone-MacBook_34805 caused collective abort of all ranks
   exit status of rank 0: return code 1

 INTERLEAVING :1

 Transition list for 0
 0 1 0 0 Send ex20.c:17 1 10{} {} 

 Transition list for 1
 0 2 0 1Recv ex20.c:21 -1 11{} {} 

 Transition list for 2
 0 3 0 2Recv ex20.c:27 0 9{} {} 

 No matching MPI call found!
 Detected a DEADLOCK!
 Killing program a.out
\end{verbatim}
So the program will always deadlock.

2.3 Example 21 - p. 8

```c
#include <stdio.h>
#include <stdlib.h>
#include <isp.h>

int main(int argc, char **argv) {
    int rank, x, y;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        x = 10; y = 11;
        MPI_Send(&x, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
        MPI_Send(&y, 1, MPI_INT, 2, 9, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, 10, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&y, 1, MPI_INT, MPI_ANY_SOURCE, 11, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("%d %d\n", x, y); fflush(stdout);
    } else if (rank == 2) {
        x = 20;
        MPI_Recv(&y, 1, MPI_INT, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(&x, 1, MPI_INT, 1, 11, MPI_COMM_WORLD);
    }
    MPI_Finalize();
```
This example is like the previous, but with different tags. So, this time the \texttt{MPI\_Send} and \texttt{MPI\_Recv} match in order.

The output then is \textbf{10 20} and the program will never deadlock.

2.4 Example 22 - p. 9

```c
#include <stdio.h>
#include <stdlib.h>
#include <isp.h>

int main(int argc, char **argv) {
  int rank, x, y;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    x = 10; y = 11;
    MPI_Send(&x, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
    MPI_Send(&y, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
  } else if (rank == 1) {
    MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&y, 1, MPI_INT, MPI_ANY_SOURCE, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("%d %d\n", x, y); fflush(stdout);
  }
  MPI_Finalize();
  return 0;
}
```
In this program only 2 processes participate. Tags and blocking operations force there to be only one inter-leaving. It will never deadlock and the output will always be $10 \ 11$.

2.5 Example 23 - p. 9

```c
#include <stdio.h>
#include <stdlib.h>
#include <isp.h>

int main(int argc, char **argv) {
    int rank, x, y;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        x = 10; y = 11;
        MPI_Send(&x, 1, MPI_INT, 1, 15, MPI_COMM_WORLD);
        MPI_Send(&y, 1, MPI_INT, 1, 16, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&y, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("%d %d\n", x, y); fflush(stdout);
    }
    MPI_Finalize();
    return 0;
}
```
Like the previous program tags and blocking \texttt{MPISend} and \texttt{MPIRecv} force there to be only one interleaving. ISP indicates that there are no deadlocks, so the program will never deadlock and the output will always be \texttt{10 11}.

\section*{2.6 Example 24 - p. 10}

In the example 24 there are 2 processes that perform blocking e non-blocking operations.

```c
#include <stdio.h>
#include <stdlib.h>
#include <isp.h>

int main(int argc, char **argv) {
    int rank, x=10, y=20;
    MPI_Request request;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        MPI_Isend(&y, 1, MPI_INT, 1, 1, MPI_COMM_WORLD, &request);
        MPI_Wait(&request, MPI_STATUS_IGNORE);
    } else if (rank == 1) {
        MPI_Irecv(&x, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
        MPI_Recv(&y, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Wait(&request, MPI_STATUS_IGNORE);
    }
}
```
Running ISP, points out that there are no deadlocks and only one interleaving, this is because the first 
\texttt{MPI\_Send} of process 0 can only match with the first \texttt{MPI\_Recv} since the tag force it and the \texttt{MPI\_Send} is a 
blocking operation, so the other \texttt{MPI\_I sending} can’t be performed until the first \texttt{MPI\_Send} matches with process 1. 
After that, the \texttt{MPI\_I sending} of process 0 can be performed and match with the second \texttt{MPI\_Recv} of process 1.

The program will never deadlock and the output will always be $10 \ 20$.

\section*{2.7 Example 25 - p. 10}
```c
else if (rank == 1) {
    MPI_Recv(&x, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &request);
    MPI_Recv(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Wait(&request, MPI_STATUS_IGNORE);
    printf("%d %d\n", x, y); fflush(stdout);
}
}
MPI_Finalize();
return 0;
}
```

This example is like the previous, there is only a interleaving, but in this case MPI functions don’t match in order (as shown by the ISP output).

Moreover, ISP does not point out any deadlock and the program’s output will always be 20 10.

### 2.8 Example 50 - p. 14

```c
#include <stdio.h>
#include <stdlib.h>
#include <isp.h>
#include <mpi.h>

int main(int argc, char **argv) {
    int rank, x=10, y=20, u=30, v;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Reduce(&u, &v, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

10
This example, is a little bit different from the others. There are 3 processes, that performs MPI_Send, MPI_Recv and a MPI_Reduce.

The program can evolves in 3 way:

**Output 10 20:** P1 performs MPI_Send, P0 performs MPI_Recv, all 3 processes perform MPI_Reduce, P2 does the MPI_Send and P0 performs the MPI_Recv

**Deadlock:** P2 performs the MPI_Reduce and the MPI_Send, P0 performs the MPI_Recv and P2 terminates. The program deadlock with P0 blocked at MPI_Reduce and P1 blocked at MPI_Send.

**Output 20 10:** P1 executes the MPI_Send, P2 MPI_Reduce and after performs MPI_Send, P0 receives the message from P2, P0 and P1 perform MPI_Reduce, finally P0 receives the message from P1.

Below, we can see the ISP’s output and the graphic.
The first interleaving show the first way which can evolve the program. So, how we can see in the ISP’s output, there is no deadlock and the output of the program is 10 20. Below we can see the graphic of the first interleaving.
In the second interleaving ISP detect a deadlock for the reasons explained previously. We can see in the next image P2 terminates correctly, P0 is blocked at $MPI\_Reduce$ and P1 blocked at $MPI\_Send$. 
ISP detects the deadlock, stop the execution and does not continue to check for next interleaving that would yield the output 20 10.

3 Additional Examples

3.1 Example 01

Process 3 does not participate in a MPI_Bcast (so if there is no Process 3 all is fine)

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

static const int root = 0;

int main(int argc, char *argv[]) {
    int sz, me;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);

    int size = 3;
    int *array = (int *)malloc(size * sizeof(int));

    if (me == root) {
        array[0] = 13;
        array[1] = 42;
        array[2] = 79;
    }

    if (me != 3)
```
3.2 Example 02

Same as previous except that I made the following change to test message blocking.

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

static const int root = 0;
int main(int argc, char *argv[]) {
    int sz, me;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);

    int size = 3;
    int *array = (int *)malloc(size * sizeof(int));
    if (me == root) {
        array[0] = 13;
        array[1] = 42;
        array[2] = 79;
    }
    if (me == 1)
        MPI_Send(array, size, MPI_INT, 3, MY_TAG, MPI_COMM_WORLD);
}```
MPI_Bcast(array, size, MPI_INT, root, MPI_COMM_WORLD);
if (me==3)
    MPI_Recv(array, size, MPI_INT, 1, MY_TAG, MPI_COMM_WORLD, &me);
    printf("At %d: %d %d %d\n", me, array[0], array[1], array[2]);
}
MPI_Finalize();

If there were less than four processes ISP notified me that it had encountered an MPI call with an invalid rank. With four or more processes it runs fine and reports that there are no deadlocks if message blocking is off. As expected if we turn message blocking on ISP finds a deadlock.

3.3 Example 03

Same as previous except that this time I’m checking barrier. I made the following change:

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

static const int root = 0;
int main(int argc, char *argv[]) {
    int sz, me;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    {
        int size = 3;
        int *array = (int *)malloc(size * sizeof(int));
```
ISP did not find the deadlock when “use blocking sends” was turned off, which is to be expected since the problem disappears when you don’t use blocking sends.

3.4 Example 04

This test involves MPI_Wait. Here is the class that works correctly:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
    int sz, me;
    int TAG = 8;
    MPI_Init(&argc, &argv);
```
MPI_Comm_size(MPI_COMM_WORLD, &sz);
MPI_Comm_rank(MPI_COMM_WORLD, &me);
{
    int size = 1; // number of elements being sent in each message
    int *array = (int*)malloc(size * sizeof(int));
    if (me == 0) {
        MPI_Request *reqs;
        MPI_Status s;
        int i, which;
        reqs = (MPI_Request*)malloc((sz - 1) * sizeof(MPI_Request));
        array[0] = 42;
        // Send a request to each process
        for (i = 1; i < sz; i++)
            MPI_Isend(array, size, MPI_INT, i, TAG, MPI_COMM_WORLD, &reqs[i-1]);
        // Wait for each request to finish
        for (i = 1; i < sz; i++)
            MPI_Waitany(sz - 1, reqs, &which, &s);
    } else {
        MPI_Request req;
        MPI_Status s;
        MPI_Irecv(array, size, MPI_INT, 0, TAG, MPI_COMM_WORLD, &req);
        printf("Process %d is receiving...
", me);
        MPI_Wait(&req, &s);
        printf("Process %d got: %d
", me, array[0]);
    }
    MPI_Finalize();
}

But when I move the MPI_Wait up a few lines (before MPI_Recv but after the declaration of MPI_Status s) then each process should wait for the MPI_Recv to happen before they perform the previously mentioned MPI_Recv. This is obviously wrong and creates a deadlock. Sadly though ISP does not catch this. A bug for this has been submitted.
References