CMDA 3634: mpiNonBlockingCircuitShell

May 29, 2018

1 Introduction

This document outlines the answers to the questions with written answers and Jumpshot images in homework 3 for CMDA 3634. This includes questions 3 and 4 in this assignment. Code for the homework is listed at the end of the document.

2 Q3

Check that your non-blocking MPI code reproduces the results of your serial code from HW01 to a reasonable tolerance. Use a convergence tolerance of 1e 9 to guarantee that the circuit currents have adequately converged for this test. Use the LATEX verbatim environment to report the loop currents output by your code.

```
From using the HW1 with a convergence tolerance of 1e-9, I was able
to get the following result.
epsilon = 1.00533e-09
epsilon = 1.00436e-09
epsilon = 1.00338e-09
epsilon = 1.00241e-09
epsilon = 1.00144e-09
epsilon = 1.00048e-09
epsilon = 9.99509e-10
I_{11} = 0.302347
I_{10 10} = 0.00317066

Using the same convergence tolerance, I was able to get almost identical result.
epsilon = 1.00482e-09
epsilon = 1.00385e-09
epsilon = 1.00288e-09
epsilon = 1.00191e-09
epsilon = 1.00094e-09
epsilon = 9.99973e-10
```

```
I_{11} = 0.302347
I_{10 10} = 0.00317066
```

3 Q4

Use jumpshot to visualize the time history of your code executing with four MPI processes on a circuit with total size 400x400. Describe any patterns you observe in the jumpshot traces. Remember to zoom in so that you can see individual MPI operations on the process timelines.

```
Figure 1, is the general overview of the jumpshot using the following executable command:
   mpiexec -n 4 ./main 400 400 1e-6
As you look further in detail, figure 2, there are lots of MPI_Allreduce and MPI_Wait
time spent among all 4 processors. Hard to tell whether MPI_Isend or MPI_Irecv are being
any work done. Figure 3, this shows the detail of the time history work done by the
process 2. As soon as the wait is finished, MPI_Isend to which ever process that is
designated to, then MPI_Irecv happens.
```

4 Code

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

/* function to convert from (i,j) cell index to linear storage index */
int idx(int N, int i, int j){

  return i + j*(N+2);
}


double allReduce(double data){
  // INPUT: data value from this MPI process to be summed up
  // OUTPUT: sum of the data variables from all MPI processes

//int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
//                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

//int MPI_Iallreduce(const void *sendbuf, void *recvbuf, int count,
//                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Reques
  double someValue;
  //MPI_Comm comm = MPI_COMM_WORLD;
```

```
 // int rank, size, i = 0;
  //MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 // MPI_Comm_size(MPI_COMM_WORLD, &size);
 // double *someData = (double*) calloc(size, sizeof(double));
  //for (i = 0; i < )

//It is not strictly necessary to compute e at every iteration. Add a command li
//change the number of iterations between computation of e.

  MPI_Allreduce(&data, &someValue, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

  return someValue;

}


/* function to compute l2 norm (Euclidean norm of difference between Inew and Io
double calculateEpsilon(int M, int N, double *Iold, double *Inew){

  double epsilon = 0;

  int i,j;
  for(j=1;j<=M;++j){
    for(i=1;i<=N;++i){
      epsilon += pow(Inew[idx(N,i,j)]-Iold[idx(N,i,j)],2);
    }
  }
  epsilon = allReduce(epsilon);

  // add up this quantity from all processeszdx
  epsilon = sqrt(epsilon);

  return epsilon;
}

/* reference halo exchange from HW02 */
void haloExchange(int M, int N, double *I){

  int rank, size, tag=999;
  MPI_Status status;

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if(rank<size-1)
    MPI_Send(I+idx(N,1,M), N, MPI_DOUBLE, rank+1, tag, MPI_COMM_WORLD);
  if(rank>0)
    MPI_Send(I+idx(N,1,1), N, MPI_DOUBLE, rank-1, tag, MPI_COMM_WORLD);

  if(rank>0)
    MPI_Recv(I+idx(N,1,0), N, MPI_DOUBLE, rank-1, tag, MPI_COMM_WORLD, &status);

  if(rank<size-1)
    MPI_Recv(I+idx(N,1,M+1), N, MPI_DOUBLE, rank+1, tag, MPI_COMM_WORLD, &status
}




void startHaloExchange(int M, int N, double *I, MPI_Request *IsendRequests, MPI_

  // initiate halo exchange using MPI_Isend and MPI_Irecv
  int rank, size, tag=999;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Status status;

  // Your Q2a code starts here
  if(rank<size-1)
        MPI_Isend(I+idx(N,1,M),   N, MPI_DOUBLE, rank+1,tag, MPI_COMM_WORLD, Ise
  if(rank>0)
        MPI_Isend(I+idx(N,1,1),   N, MPI_DOUBLE, rank-1,tag, MPI_COMM_WORLD, Ise
  if(rank>0)
        MPI_Irecv(I+idx(N,1,0), N, MPI_DOUBLE, rank-1, tag, MPI_COMM_WORLD, Irec
  if(rank<size-1)
        MPI_Irecv(I+idx(N,1,M+1), N, MPI_DOUBLE, rank+1, tag, MPI_COMM_WORLD, Ir



  // Your Q2a code ends here
}

void endHaloRecv(MPI_Request *IrecvRequests){

  // wait for halo data recv to complete using MPI_Wait
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // Your Q2b code starts here
  MPI_Status status;

 // MPI_Wait(IrecvRequests, &status);
  // Your Q2b code ends here

  if(rank<size-1)
        MPI_Wait(IrecvRequests , &status);

  if(rank>0)
        MPI_Wait(IrecvRequests +1, &status);
}

void endHaloSend(MPI_Request *IsendRequests){

  // wait for outgoing halo data send to leave the buffer using MPI_Wait
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // Your Q2c code starts here
  MPI_Status status;

  //MPI_Wait(IsendRequests, &status);
  // Your Q2c code ends here

  if(rank<size-1)
        MPI_Wait(IsendRequests , &status);

  if(rank>0)
        MPI_Wait(IsendRequests +1, &status);
}

/* function to update Inew from Iold */
void iterate(int M, int N, double *Iold, double *Inew){

  MPI_Request IsendRequests[2], IrecvRequests[2];
  int i,j;

//  int rank, size;
//  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
//  MPI_Comm_size(MPI_COMM_WORLD, &size);
//  MPI_Status status;


  // initializes the swap of the top/bottom rows needed by the iterate step
  startHaloExchange(M, N, Iold, IsendRequests, IrecvRequests);
  // process cell updates that do not require halo data from other processes
  for(j=2;j<=M-1;++j)
    for(i=1;i<=N;++i)
      Inew[idx(N,i,j)] = 0.25*(Iold[idx(N,i+1,j)] + Iold[idx(N,i-1,j)] +
                               Iold[idx(N,i,j+1)] + Iold[idx(N,i,j-1)]);

  // wait for the incoming halo data to arrive
  endHaloRecv(IrecvRequests);

  // finish update for bottom cells
  j = 1;
  for(i=1;i<=N;++i)
    Inew[idx(N,i,j)] = 0.25*(Iold[idx(N,i+1,j)] + Iold[idx(N,i-1,j)] +
                             Iold[idx(N,i,j+1)] + Iold[idx(N,i,j-1)]);

  // finish update for top cells
  j = M;
  for(i=1;i<=N;++i)
    Inew[idx(N,i,j)] = 0.25*(Iold[idx(N,i+1,j)] + Iold[idx(N,i-1,j)] +
                             Iold[idx(N,i,j+1)] + Iold[idx(N,i,j-1)]);


  // wait for the outgoing halo data buffer to be available for use
  endHaloSend(IsendRequests);

}




/* function to solve for loop currents using Jacobi iterative method */
void solve(int M, int N, double tol){

  /* use for computed epsilon */
  double epsilon;

  double *Inew = (double*) calloc((M+2)*(N+2), sizeof(double));
  double *Iold = (double*) calloc((M+2)*(N+2), sizeof(double));

  /* set batteries based on MPI process rank*/
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  // Your Q1 code to set the ghost cells for the two batteries
  // based on MPI process rank starts here
  if(rank==0){ // bottom left cell
    Iold[idx(N,0,1)] = 1;
    Inew[idx(N,0,1)] = 1;
  }

  if(rank==size-1){ // top right cell
    printf("cell\n");
    Iold[idx(N,N+1,M)] = 1;
    Inew[idx(N,N+1,M)] = 1;
  }
  // Your Q1 code to set the ghost cells ends here


  /* iterate using the Jacobi method here */
  do{

    /* iterate from Iold to Inew */
    iterate(M, N, Iold, Inew);

    /* iterate from Inew to Iold */
    iterate(M, N, Inew, Iold);

    /* compute epsilon (change in current) */
    epsilon = calculateEpsilon(M, N, Iold, Inew);

    /* print current residual */
    if(rank==0)
      printf("epsilon = %g\n", epsilon);

  }while(epsilon>tol);

  /* print out the loop current in cell (1 1) and (10 10) */
  if(rank==0)
    printf("I_{11} = %g\n", Iold[idx(N,1,1)]);
  if(rank*M <= 10 && 10 <= ((rank+1)*M))
    printf("I_{10 10} = %g\n", Iold[idx(N,10, 10-M*rank)]);

}

/* usage: ./main 100 1e-6
   mpiexec -n 4 ./main 100 25 1e-6
   to solve for a network of 100x100 to tolerance 1e-6 */

int main(int argc, char **argv){
  // Your Q1 code to call MPI_Init starts here
  MPI_Init(&argc, &argv);
  // Your Q1 code to call MPI_Init ends here

  {
    /* read N from the command line arguments */
    int N = atoi(argv[1]);

    /* read N from the command line arguments */
    int M = atoi(argv[2]);

    /* read the user supplied convergence from the command line arguments */
    double tol = atof(argv[3]);

    /* perform Jacobi iteration to solve for loop currents in resistor network *
    solve(M, N, tol);
  }

  // Your Q1 code to call MPI_Finalize starts here
  MPI_Finalize();
  // Your Q1 code to call MPI_Finalize ends here

  return 0;

}
```

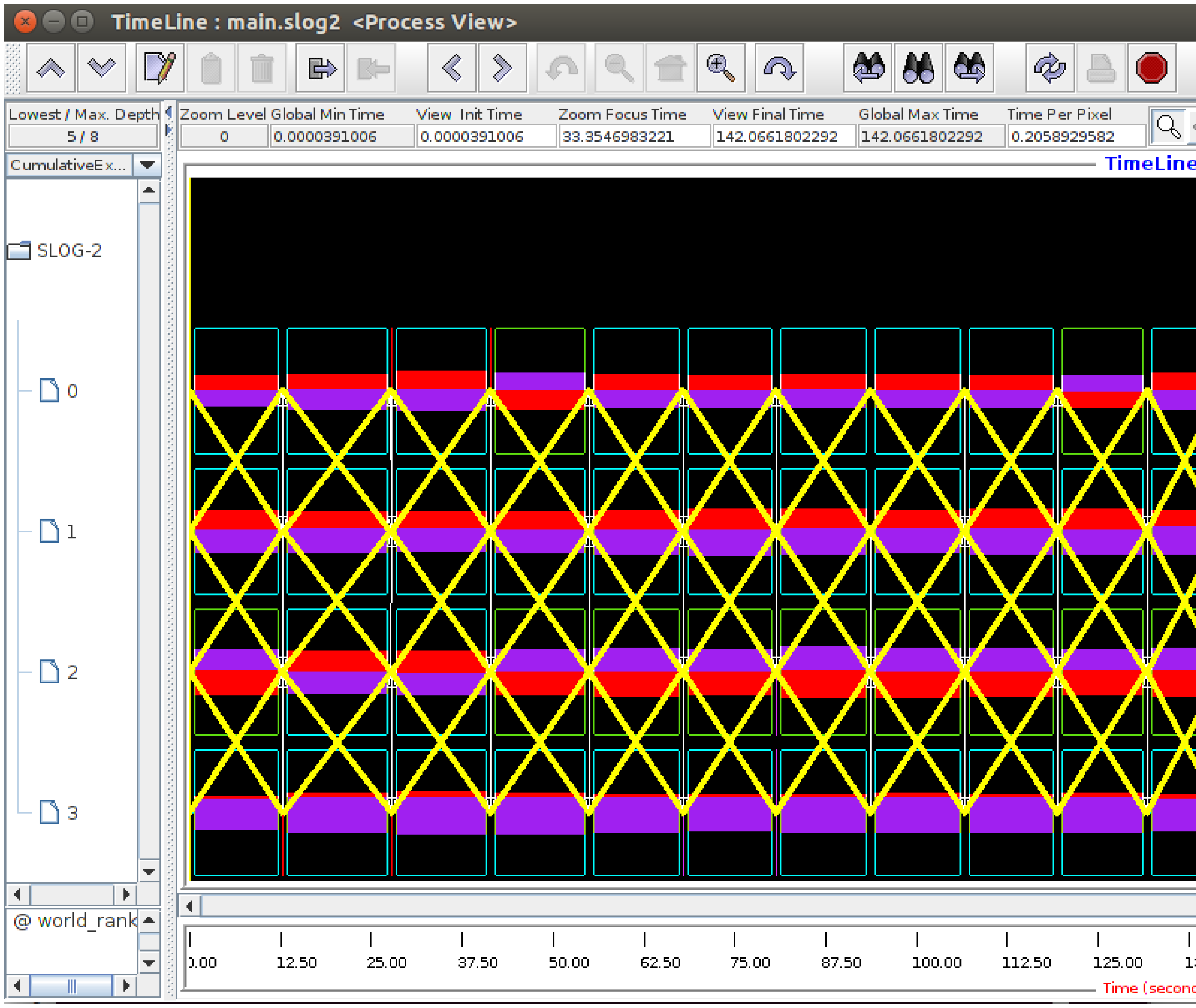
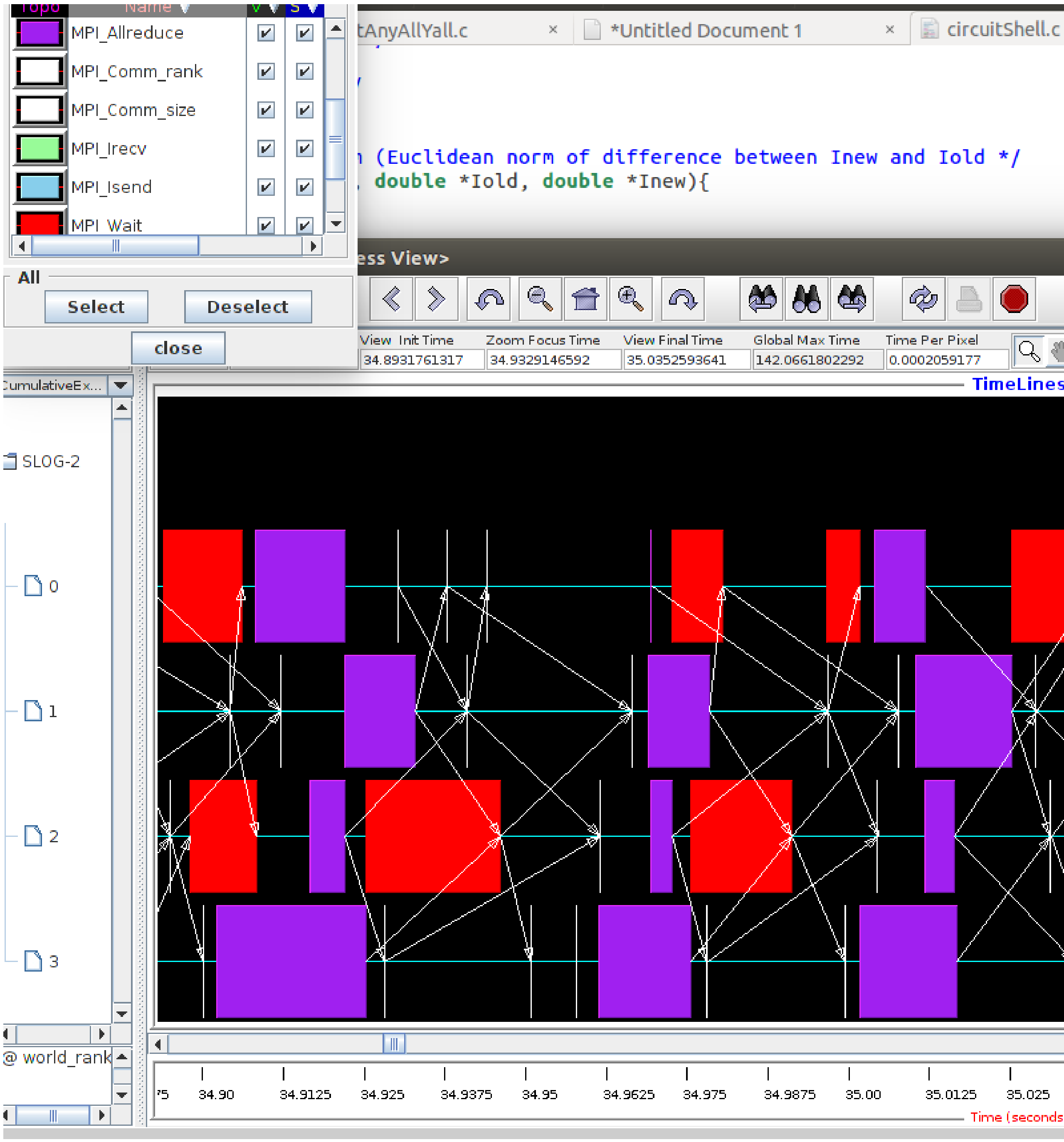


Figure 1: HW3 Jumpshot

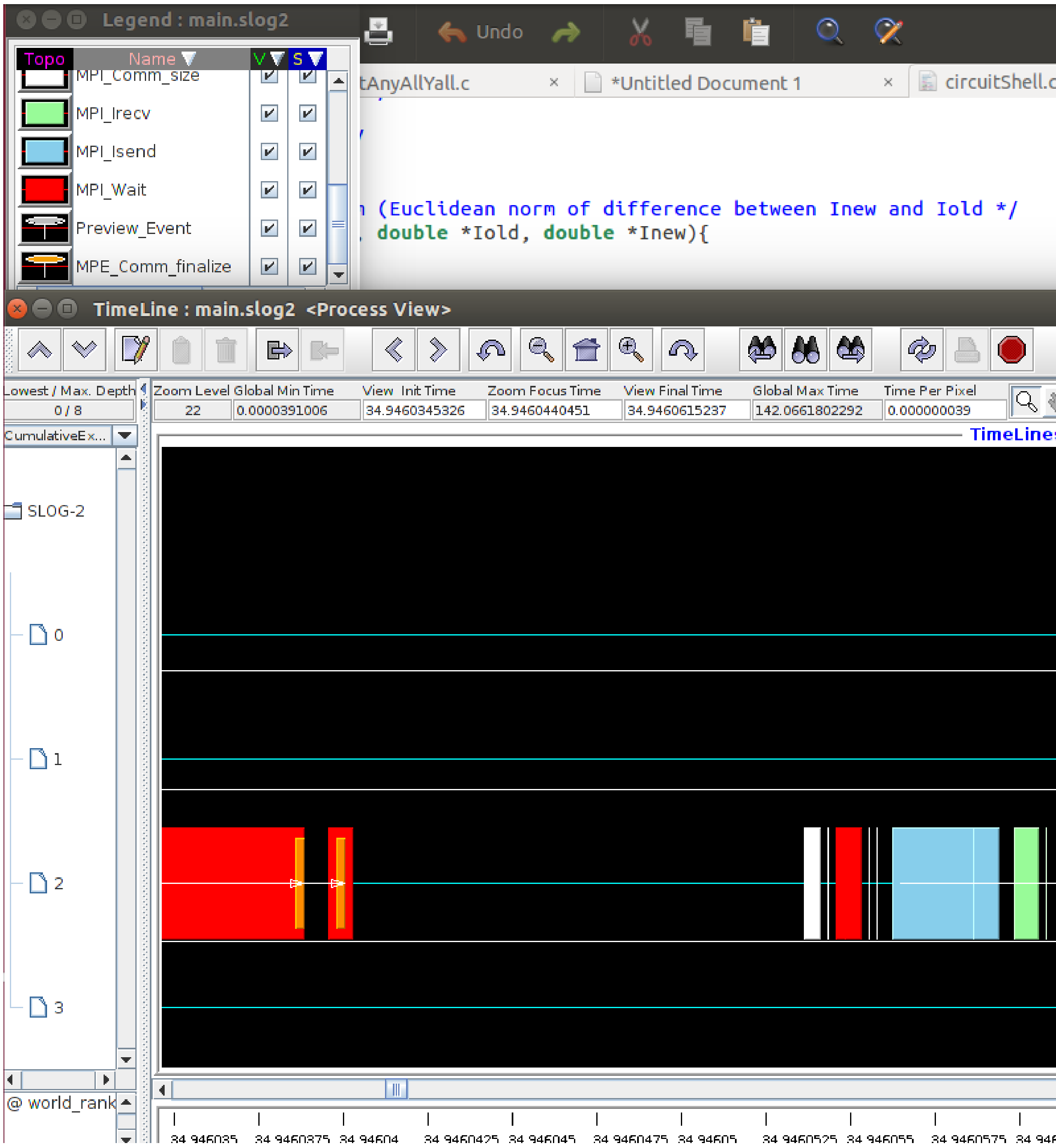


Topo	Name	V	S
🟪	MPI_Allreduce	✓	✓
⬜	MPI_Comm_rank	✓	✓
⬜	MPI_Comm_size	✓	✓
🟩	MPI_Irecv	✓	✓
🟦	MPI_Isend	✓	✓
🟥	MPI_Wait	✓	✓

```
n (Euclidean norm of difference between Inew and Iold */
, double *Iold, double *Inew){
```

View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
34.8931761317	34.9329146592	35.0352593641	142.0661802292	0.0002059177

Figure 2: HW3 Jumpshot



Topo	Name	V	S
⬜	MPI_Comm_size	✓	✓
🟩	MPI_Irecv	✓	✓
🟦	MPI_Isend	✓	✓
🟥	MPI_Wait	✓	✓
	Preview_Event	✓	✓
	MPE_Comm_finalize	✓	✓

```
n (Euclidean norm of difference between Inew and Iold */
, double *Iold, double *Inew){
```

Lowest / Max. Depth	Zoom Level	Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
0 / 8	22	0.0000391006	34.9460345326	34.9460440451	34.9460615237	142.0661802292	0.000000039

Figure 3: HW3 Jumpshot