CMDA 3634: mpiCircuitShell

February 19, 2016

1 Introduction

This document outlines the answers to the questions with written answers in homework 2 for CMDA 3634. This includes questions 6 and 7 in this assignment. Code for Q1-Q4 is listed at the end of the document.

2 Q6

Check that your MPI code reproduces the results of your serial code from HW01 to a reasonable tolerance. As you can see from the figure 1 and 2, results from both HW01 and HW02 came out very similar.

3 Q7

Figure 3, is the screen shot of the jumpshot using the given commend: mpiexec -n 4 ./main 100 25 1e-6 Total process takes few seconds whereas circuit size 400x400 takes about a minute to just finish compile mpicc. As you can see on the bottom of Figure 4 jumpshot, time lapse is much greater. Time frame between 90 to 100 seconds is where more complex data were being solved. So when I zoomed in for detail, each process time frames were very inconsistent. Some took less than a quarter of a second, some were about 2 seconds long.

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);
}

/*
Implement function to collectively sum up a variable from all MPI processes and
the result to all processes. This operation is called an   all   Reduce operat
You should adapt the barrier function in mpiBarrierTree.c that we developed in c
*Isend prmoises to do correct execution
*/
double allReduce(double data){

    // INPUT: data value from this MPI process to be summed up
  // RETURN: INPUT data values summed up over all MPI processes using tree reduc
   double reduceData;
   MPI_Status status;
   int rank, size, alive, source;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   int active = size/2;
/*
  while(active){
    if(rank<active)
      MPI_Recv(&reduceData, 1, MPI_DOUBLE, source=rank+active, 999, MPI_COMM_WOR
    else{
      MPI_Send(&data, 1, MPI_DOUBLE, dest=rank-active, 999, MPI_COMM_WORLD);
    }
    data += reduceData;
    active /= 2;
  }*/

  double message = rank;
  int messageLength = 1;
  int messageTag = 999;
  alive = size;

  // keep looping until there are only 2 processes left
  while(alive>1){

    // bottom 1/2 of alive threads receive from top 1/2 of alive threads
    if(rank+(alive+1)/2<alive && rank+(alive+1)/2<size){

      MPI_Recv(&message, messageLength, MPI_DOUBLE,
               rank+(alive+1)/2, messageTag, MPI_COMM_WORLD, &status);
```

```
      data = data + message;
    }

    // top 1/2 of alive threads receive from bottom 1/2 of alive threads
    if(rank>=(alive+1)/2 && rank<alive){

      MPI_Send(&data, messageLength, MPI_DOUBLE,
               rank-(alive+1)/2, messageTag, MPI_COMM_WORLD);
    }

    // kill half the processes
    alive /= 2;
  }
  // keep looping until all the threads are alive
  alive = 1;
  while(alive<size){
    // send to rank + alive;
    if(rank<alive && rank+alive<size){

      MPI_Send(&data, messageLength, MPI_DOUBLE,
               rank+alive, messageTag, MPI_COMM_WORLD);
    }

    // receive from rank - alive
    if(rank>=alive && rank<2*alive && rank<size){
      MPI_Recv(&data, messageLength, MPI_DOUBLE,
               rank-alive, messageTag, MPI_COMM_WORLD, &status);
    }
    alive *= 2;
  }

  // pass summed data back to calling function
  return data;
}


/* 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);
```

```
    }
  }

  // add up this quantity from all processes
  epsilon = allReduce(epsilon);

  epsilon = sqrt(epsilon);

  return epsilon;
}

/* halo exchange
* Implement haloExchange function to exchange data prior update step in iterate
*
* The iterate function requires one extra step where each process needs to share
    The iterate calls the haloExchange function that exchanges its top and botton
*/
void haloExchange(int M, int N, double *I){

  int rank, size, dataTag=999;
  MPI_Status status;
  int destination = rank+1;
  int source = rank-1;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  int offsetTop, offsetBottom;
  int circuitSize = (M*size) * N;

//int idx(int N, int i, int j)
// return i + j*(N+2);
  offsetTop = idx(N, 1, M*(rank+1));
  offsetBottom = idx(N, 1, M*(rank));
  int n = 0;

  if(rank>0){
 // double *newi=
//       (double*)calloc(N, sizeof(double));

  // receive from p+1 and p-1
  MPI_Recv(I + offsetBottom, N, MPI_DOUBLE, source, dataTag, MPI_COMM_WORLD, &st

  MPI_Send(I + offsetBottom, N, MPI_DOUBLE, source, dataTag, MPI_COMM_WORLD);
  }

  if(rank<size-1){
  // send from p+1 and p-1
```

```
  MPI_Recv(I + offsetTop, N, MPI_DOUBLE, destination, dataTag, MPI_COMM_WORLD, &
  MPI_Send(I + offsetTop, N, MPI_DOUBLE, destination, dataTag, MPI_COMM_WORLD);
  }
}

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

  int i,j;

  // this function sends/recvs the top/bottom rows
  // between processes as needed before iterating
  haloExchange(M, N, Iold);

  for(j=1;j<=M;++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)]);
    }
  }

}
// M = 25
// N = 100
/* 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);

  // Set the ghost cells for the two batteries
  // based on MPI process rank
  if(rank==0){ // bottom left cell
```

```
    Iold[idx(N,0,1)] = 1;  //
    Inew[idx(N,0,1)] = 1;  //Inew[2]
  }

  if(rank==size-1){ // top right cell
    Iold[idx(N,N+1,M)] = 1;
    Inew[idx(N,N+1,M)] = 1;  // Inew[idx(100, 100+1, 25)] = Inew[25 + 101*(100+2
  }
        //int idx(int N, int i, int j)
        // return i + j*(N+2);

  /* 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 on MPI process 0 */
    if(rank==0)
      printf("epsilon = %g\n", epsilon);

  }while(epsilon>tol);

  /* print out the loop currents
     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:
   mpiexec -n 4 ./main 100 25 1e-6
   To solve for a network of 100x100 to tolerance 1e-6 using four processes with
*/
int main(int argc, char **argv){
```

```
  // Your Q2 code to call MPI initialization starts here
    MPI_Init(&argc, &argv);
  // Your Q2 code to call MPI initialization 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 convergence tolearnce 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 Q3 code to call MPI finalization starts here
  MPI_Finalize();
  // Your Q3 code to call MPI finalization ends here

  return 0;

}
```

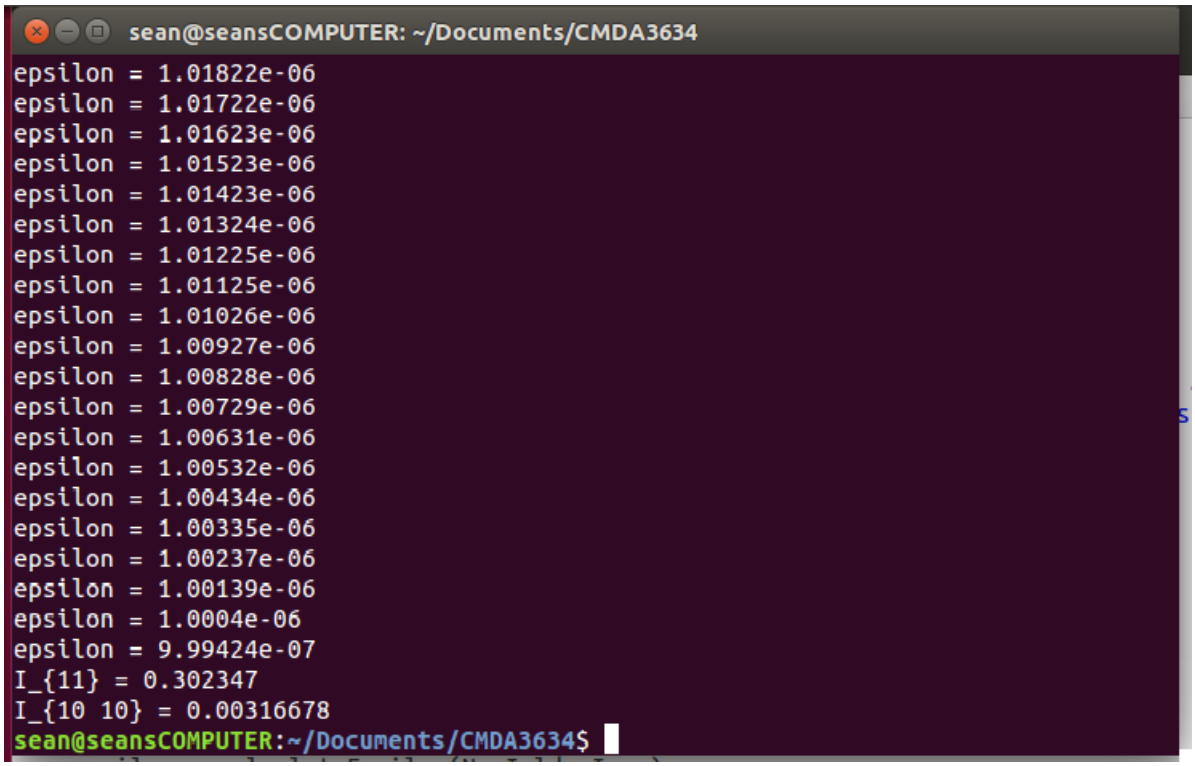

Figure 1: HW1 output

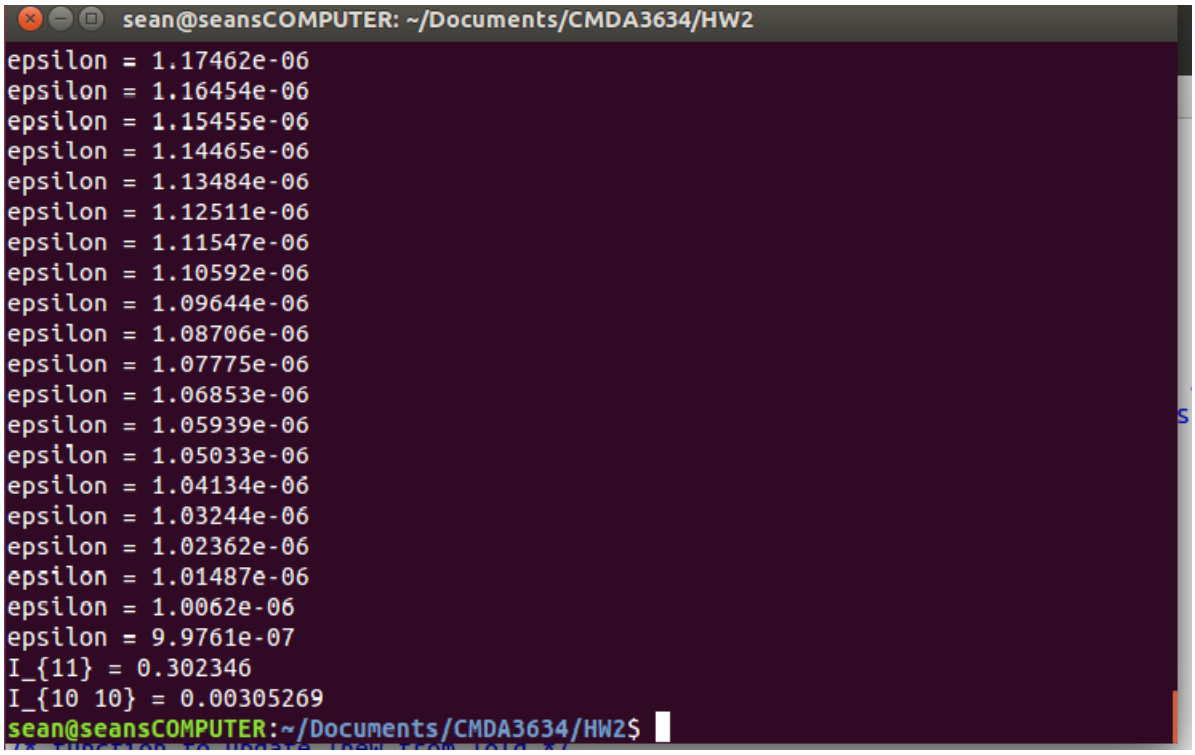

Figure 2: HW2 output

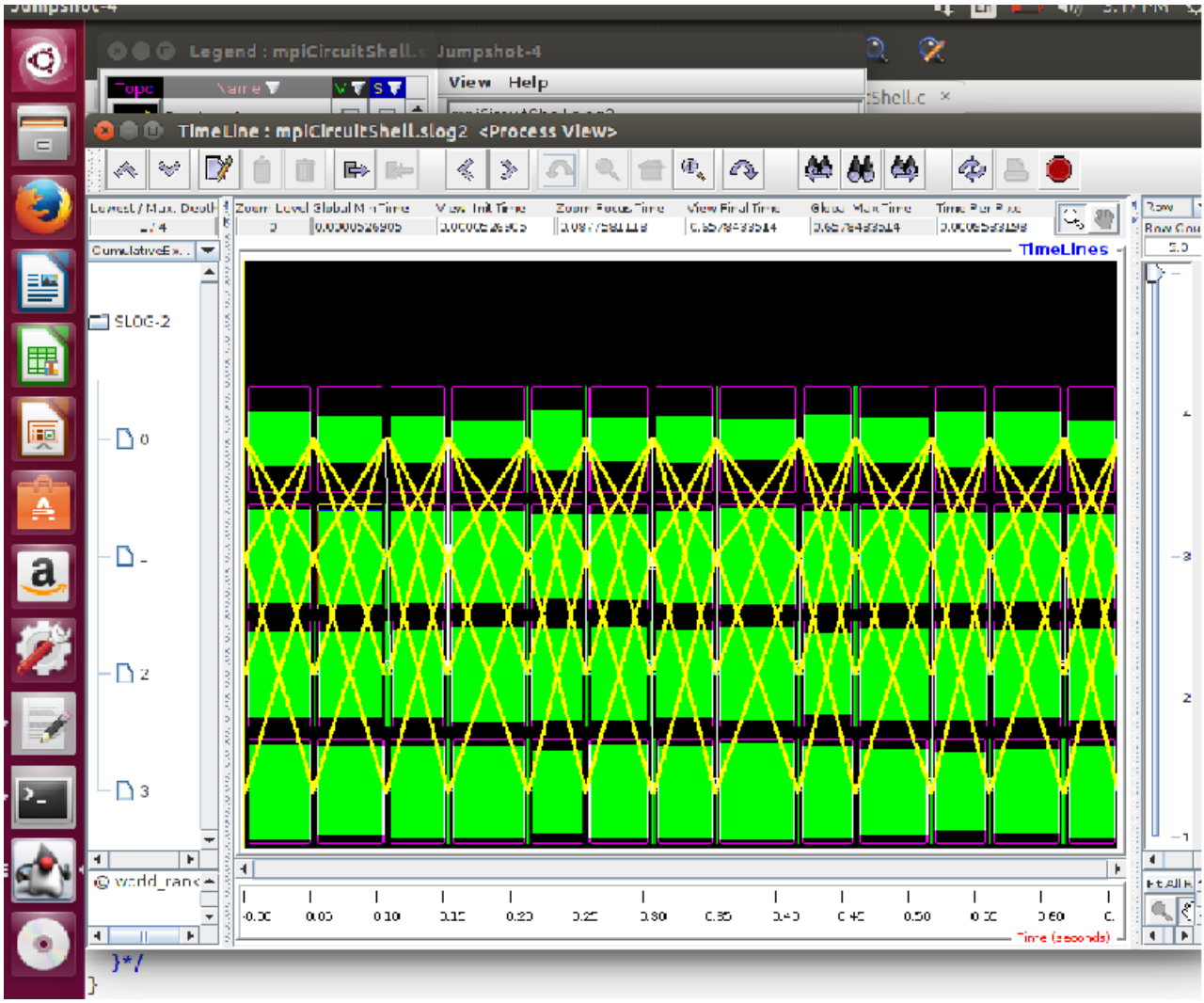

Figure 3: HW2 jumpshot

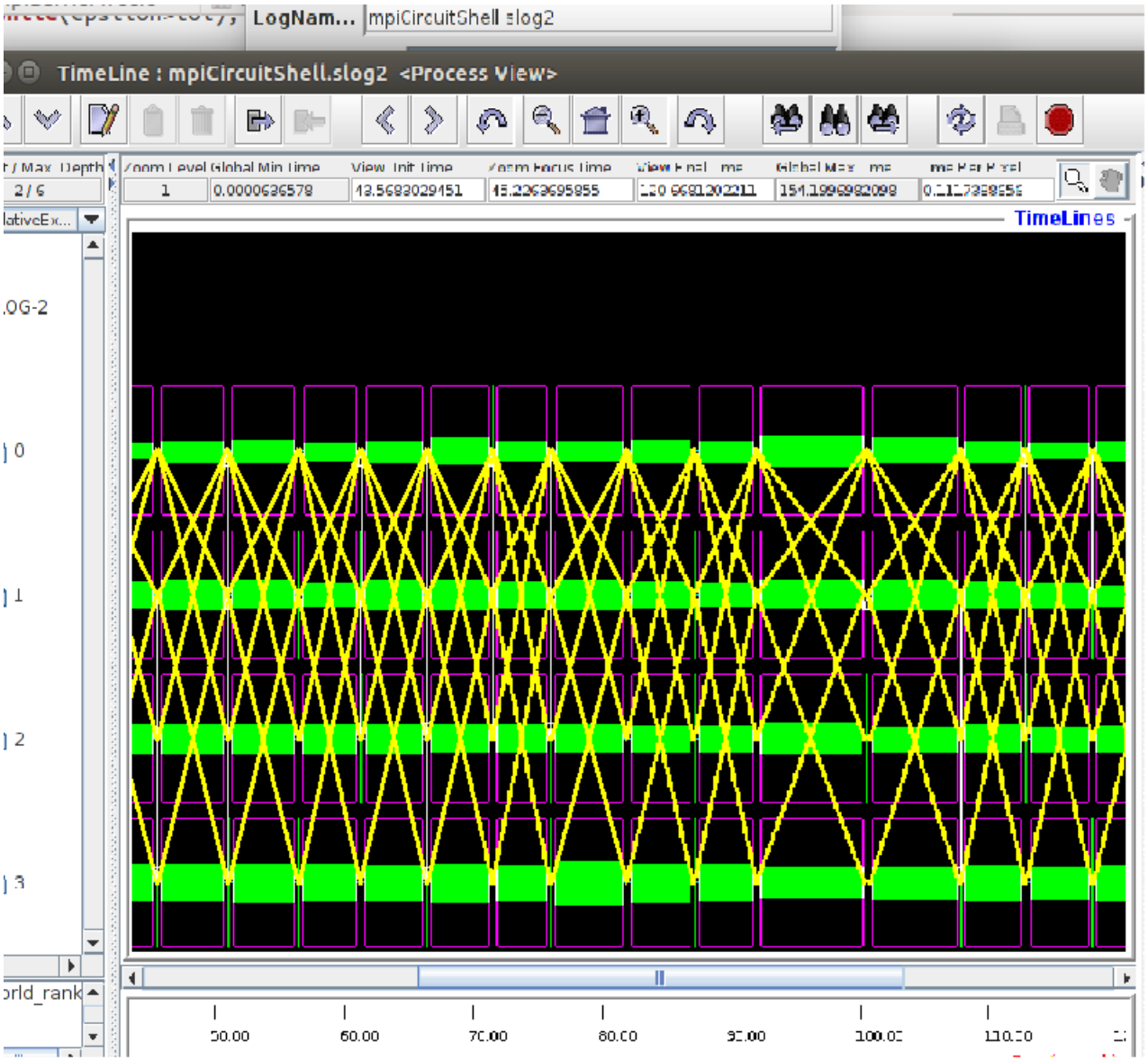

Figure 4: HW2 jumpshot