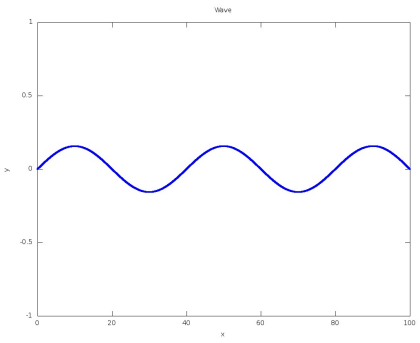
CS 789 Parallel Programming
Assignment 3
Parallel Wave Equation

Lucas Bang



Contents

1 Problem 1: Implementation

Implement a leader/worker version of the program using MPI.

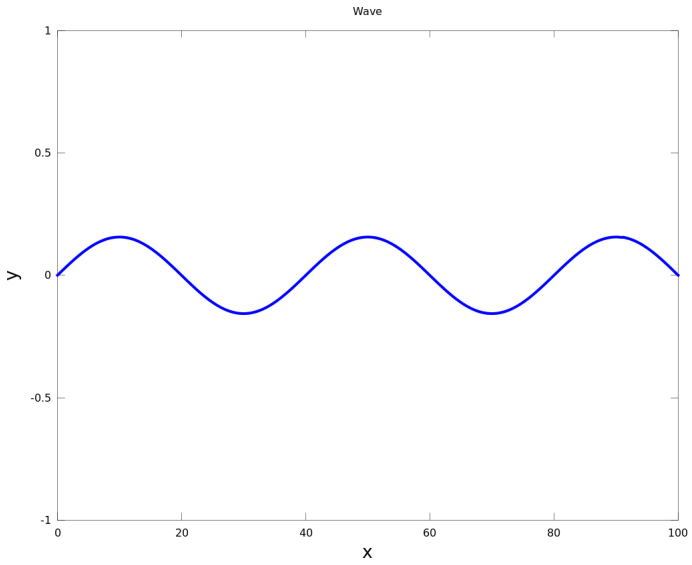
The parallel version was successfully implemented. See Appendix A for the code.

2 Problem 2: Verification

Perform a number of parallel runs to ensure that the parallel version computes the same as the sequential version.

Yes, the parallel version computes the same as the sequential version. The following figure shows the result of executing the command

```
mpirun -machinefile hosts -np 4 parWave 100 5 10000 4500 0.00005
```



3 Problem 3: Speedup

Measure and report speedups.

In order to measure speedup, I ran the program for constant values of $n = 32768$, $L = 100$, $s = 150000$, and $\Delta t = 0.0000005$, while varying the number of processors. I was able to run the program for several runs while I was the only person logged into `cortex`. Thus, the values I obtained for timing and speedup are very reliable. The times and speedups are shown in figures 1 through 4. The jump in each graph for 18 processes is an anomaly. I am not sure what caused this, but I believe it has something to do with the system running some other process, and has nothing to do with the time actually being worse for 18 processes.

In Figure 1, I have plotted the average time for the number of processes ranging from 3 to 19. You can see that the time decreases quickly at first and then levels off. This is as expected. I expected the time to slowly increase in an approximately linear fashion from then on and eventually top out, so I tested two more runs with 36 and 128 processes. These points are included in Figure 2, which indicates that the time does increase as the number of processors increases.

I then intended to measure times for a range of processes between 20 and 128 in multiples of 4, or perhaps 8, but it was at this time that the server became congested with other users. Hence any timing data collected showed high variance and was of course larger than the time I would have otherwise measured. Therefore, I did not collect any more of that unreliable data. It would have been interesting to see how the time varied for these larger ranges, but the upward trend is still visible in Figure 2.

Figure 1: Number of processes between 3 and 19.

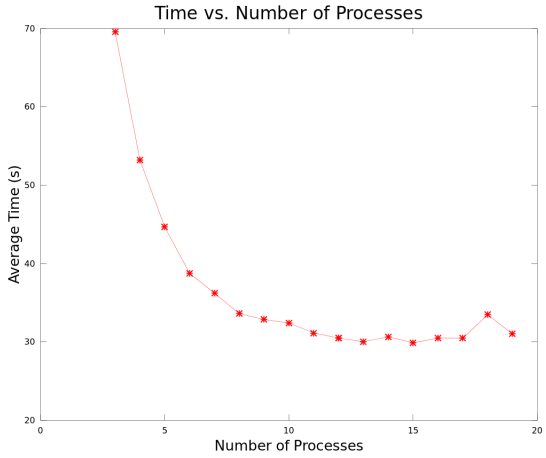
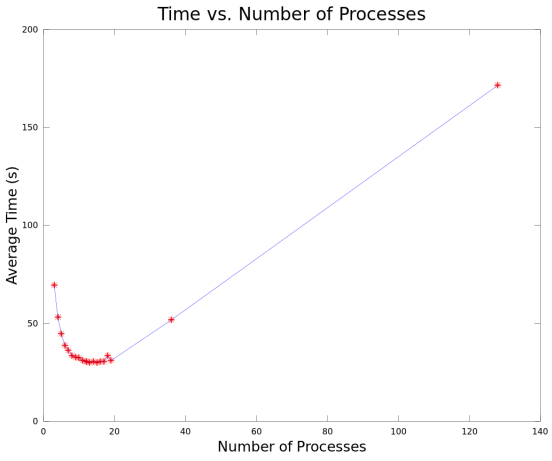


Figure 2: Number of processes between 3 and 19, one run for 36 processes, and one for 128 processes.



Figures 3 and 4 depict the speedup that was achieved. The sequential code run with the same parameters had an average running time of 128.134 seconds. The speedup graphs are then simply graphs of average sequential time

divided by average parallel time. The maximum speedup I achieved was 4.516 times faster than that sequential implementation.

Figure 3: Number of processes between 3 and 19.

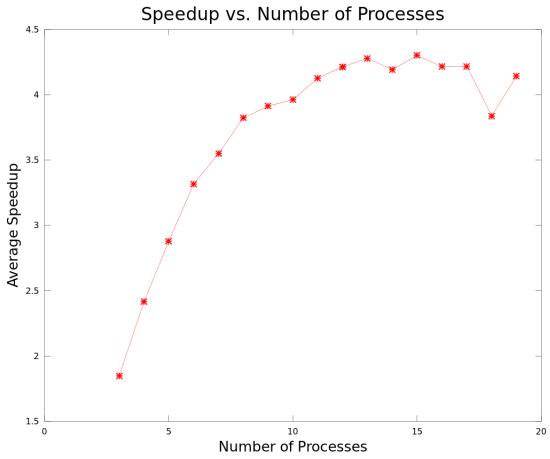
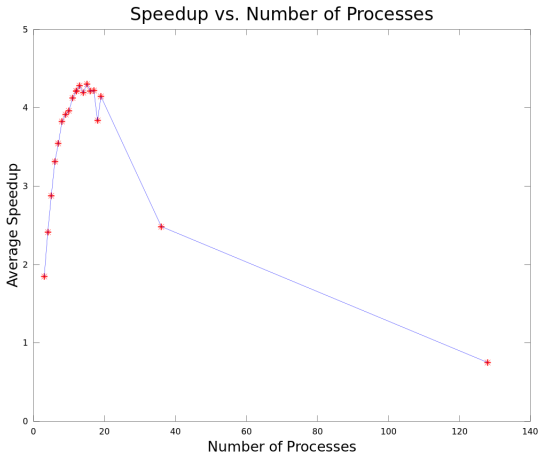


Figure 4: Number of processes between 3 and 19, one run for 36 processes, and one for 128 processes.



4 Problem 4: Load Sensitivity

Briefly explain why this program is more sensitive to load difference than the processor farm implementation for Mandelbrot.

This problem is more sensitive to load difference due to the the dependency of one process on the computation of another. Because process i must communicate with processes $i-1$ and $i+1$ between every round of computation, a single tardy process can slow down the entire chain of communication and computation. This is not the case in the parallel computation of the Mandelbrot set, where each process can simply do all of its communication independently and will never be forced to wait for another process (except for the leader).

5 Problem 5: Load Balancing

Suggest a way to implement this program such that load balancing is taken into account.

In order to load balance this problem, a process that is consistently slow should be given less work to do. This requires a clever way of dynamically deciding how much of the array a given process should be responsible for computing. One possibility is for each process to keep track of the amount of time it spends computing on every step. Then, along with the values that it computed at its endpoints, it sends the amount of time it spent computing. It then compares its own time to the times of it's neighbors. If the difference between the two times is larger than some threshold, the process either increases or decreases the left and right endpoints of the interval over which it computes, while it's neighbors decrease or increase theirs respectively. The threshold should be chosen so that this resizing does not occur at every step, and may need to be tuned for specific instances of the problem. Alternatively, one could implement a counter, so that if the time difference is beyond the threshold a certain number of times in a row then the resizing occurs. Again, we want a process that is *consistently* slow to be punished and a process that is *consistently* fast to be rewarded, so the resizing should not happen for small differences in time that occur only once in a while.

6 Problem 6: Theoretical vs. Real Speedup

For a fixed size problem use timers to measure the parts of the parallel program that cannot be parallelized (i.e., typically file IO in the leader) and use this time to determine the maximum speedup that you should be able to achieve. Did you get the speed up you expected?

The average time for file IO in the sequential version was 0.087651 seconds, while the average total time for the sequential version was 128.134 seconds. We compute the portion of code that can be parallelized as

$$P = \frac{t_{SEQ} - t_{IO}}{t_{SEQ}}.$$

Plugging in the measured values gives $P = 0.99931594$. We compute the maximum possible speedup using Amdahl's Law

$$SU_{max} = \frac{1}{1 - P}.$$

Again, plugging in our measured value of P, we get a maximum possible speedup of 1461.86579803!

This seems like a far cry off from the speedup we actually achieved. Why is this? Well, increasing the number of processes has diminishing returns. Amdahl's law arises from taking a limit as n goes to infinity. However, we can't really let n got to infinity. In fact, it wouldn't even make sense in this problem to let the number of processes rise above the number of nodes simulated in the string, as this is the finest granularity we might hope to achieve for a given discretization of the spatial domain. Even so, as the number of processes rises the amount of communication required for each computation round goes up while the amount of computation done by a single processes goes down. Since the time for a single communication is (MUCH) larger than a single computation step, the communication time will eventually overtake the computation time. Furthermore, the portion of the problem that is "paralellizable" isn't completely so, due to the dependency on the computation of your neighbors. These are several factors that Amdahl's law does not account for, and so the actual observed speedup in practice is much lower than the predicted speedup.

A Parallel Wave Implementation

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

int main(int argc, char *argv[]) {
     MPI_Init(&argc, &argv);
     gettimeofday(&t3,NULL);
     int n;
  int nb;
  long double l;
  long double dt;
  int steps;
  FILE *y_file;
  int i,j,s;

  if (argc != 6) {
    printf("Usage: wave <l> <nb> <n> <steps> <dt>\n");
    printf("\t l \t = Total length of the string (x-axis).\n");
    printf("\t nb \t = Number of half sine waves.\n");
    printf("\t n \t = Number of nodes (number of discrete points on the x-axis between 0 and l.)\n");
    printf("\t steps \t = Number of steps.\n");
    printf("\t dt \t = Size of each step.\n%d args supplied\n",argc);
    exit(0);
  }

  l     = atof(argv[1]);
  nb    = atoi(argv[2]);
  n     = atoi(argv[3]);
  steps = atoi(argv[4]);
  dt    = atof(argv[5]);

  int num_workers, size, rank;
  MPI_Status status;
  char hostname[256];
  gethostname(hostname,255);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
     num_workers = size - 1;

     if(rank == 0){
          int worker_n, worker_rank;
          long double *x, *y;
          long double * y_piece;
          worker_n = n / num_workers;

          x = (long double *) calloc(n,sizeof(long double));
          y = (long double *) calloc(n,sizeof(long double));
          y_piece = (long double *) calloc(worker_n+2,sizeof(long double));

          for (i=0; i<n; i++)
               x[i] = l*i/(n-1);

          int idx = 0;
          for(i=1; i<size; i++){
               MPI_Recv(y_piece, worker_n + 2, MPI_LONG_DOUBLE, i, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
               for (j=1; j<worker_n+1; j++){
                    y[idx] = y_piece[j];
                    idx++;
               }
          }
          y_file = fopen("resultSeq.txt", "w" );
          if (y_file == (FILE *)NULL) {printf("Could not open output file.\n"); exit(1);}
          for (i=0; i<n; i++)
               fprintf(y_file,"%Lf %15.15Lf\n",( l * i )/( n - 1 ), y[i]);
          fclose(y_file);
     }
     else{
          long double *x, *y, *yold, *ynew;
          long double pi, tau, dx;
          pi = 4.0 * atan(1);
          dx = l/(n-1);
          tau = 2.0*l*dt/nb/dx;
```

```
        int my_n;
        long double y_left_send, y_right_send, y_left_recv, y_right_recv;
        long double my_x_start;

        my_n = n/num_workers;
        x = (long double *) calloc(my_n+2,sizeof(long double));
        y = (long double *) calloc(my_n+2,sizeof(long double));
        yold = (long double *) calloc(my_n+2,sizeof(long double));
        ynew = (long double *) calloc(my_n+2,sizeof(long double));
        my_x_start = dx*(my_n)*(rank-1);

        for (i=0; i<my_n+2; i++)
          x[i] = i*dx+my_x_start;

        for (i=0; i<my_n+2; i++) {
            y[i] = sin(pi*nb*x[i]/l);
                  yold[i] = sin(pi*nb*x[i]/l);
        }

        for (s=0; s<steps; s++) {
             if(rank==1){
                  y_right_send = y[my_n];
                  MPI_Send(&y_right_send, 1, MPI_LONG_DOUBLE, rank+1,
                            1, MPI_COMM_WORLD);
                  MPI_Recv(&y_right_recv, 1, MPI_LONG_DOUBLE, rank+1,
                            MPI_ANY_TAG, MPI_COMM_WORLD, &status);
                  y[my_n+1] = y_right_recv;
             }
             else if(rank==size-1){
                  y_left_send = y[1];
                  MPI_Send(&y_left_send, 1, MPI_LONG_DOUBLE, rank-1,
                            1, MPI_COMM_WORLD);
                  MPI_Recv(&y_left_recv, 1, MPI_LONG_DOUBLE, rank-1,
                            MPI_ANY_TAG, MPI_COMM_WORLD, &status);
                  y[0] = y_left_recv;
             }
             else{
                  y_right_send = y[my_n];
                  y_left_send = y[1];

                  MPI_Send(&y_right_send, 1, MPI_LONG_DOUBLE, rank+1,
                            1, MPI_COMM_WORLD);
                  MPI_Recv(&y_right_recv, 1, MPI_LONG_DOUBLE, rank+1,
                            MPI_ANY_TAG, MPI_COMM_WORLD, &status);
                  y[my_n+1] = y_right_recv;

                  MPI_Send(&y_left_send, 1, MPI_LONG_DOUBLE, rank-1,
                            1, MPI_COMM_WORLD);
                  MPI_Recv(&y_left_recv, 1, MPI_LONG_DOUBLE, rank-1,
                            MPI_ANY_TAG, MPI_COMM_WORLD, &status);
                  y[0] = y_left_recv;
             }
            for (i=1; i<my_n+1; i++) {
              ynew[i] = 2.0*y[i]-yold[i]+tau*tau*(y[i-1]-2.0*y[i]+y[i+1]);
            }
            for (i=0; i<my_n+2; i++) {
              yold[i] = y[i];
              y[i] = ynew[i];
            }
        }
        MPI_Send(y, my_n+2, MPI_LONG_DOUBLE, 0, 1, MPI_COMM_WORLD);
    }
  MPI_Finalize();
  return 0;
}
```