Christopher Lewis

PA3: Bucket Sort

3/30/2017

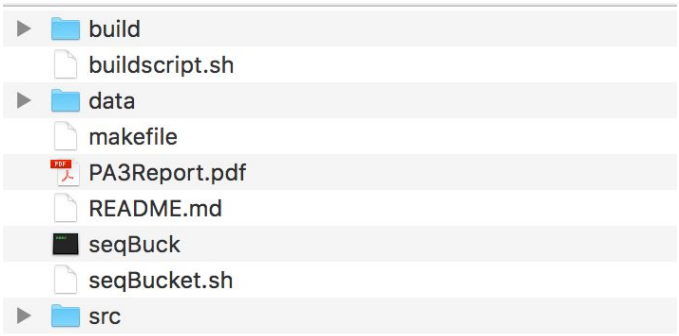
Table of Contents

Overview:

Project 3 requires that I take a set of numbers read in from a file and I get the size and all the numbers and sort them in an array. Part 1 of the assignment is implementing bucket sort sequentially and Part 2 requires that I implement bucket sort in parallel.

Project Layout:

Below is the structure of the project.



Inside **build** you will find a script that will be called using **sbatch** below is a sample of what will be found inside the script(s).

Sequential Script

```
#!/bin/bash
#SBATCH -N1
#SBATCH -n1
#SBATCH --time=01:00:00
#SBATCH --mail-user=christopherlewis@nevada.unr.edu
for (( a=100000000; a<=1500000000; a+=100000000 ))
do
    srun seqBucket $a seqBucketSortTimed.txt
done
```

Parallel Script

```
#!/bin/bash
#SBATCH -N1
#SBATCH -n4
#SBATCH --time=01:00:00
#SBATCH --mail-user=christopherlewis@nevada.unr.edu
for (( a=100000000; a<=1500000000; a+=100000000 ))
do
    srun parBucket $a fourCores.txt
done
```

Sequential Bucket-Sort

Sequential Bucket Sort is a sort that will divide up your set of numbers into buckets and will sort each bucket using an individual sort of the programmer's choice. Below in Table 1 is the file that hold the values for the chart below. Figure 1, Also below, is a diagram of the table plotted on a line.

	1
100000000	3.107935
200000000	6.241437
300000000	9.631196
400000000	12.430669
500000000	15.256826
600000000	19.32263
700000000	22.095468
800000000	24.886312
900000000	27.706288
1000000000	30.667014
1100000000	33.849104
1200000000	38.541852
1300000000	41.852368
1400000000	46.293893
1500000000	49.1953

Table 1: In the table above, right column is the time and the left is the size.

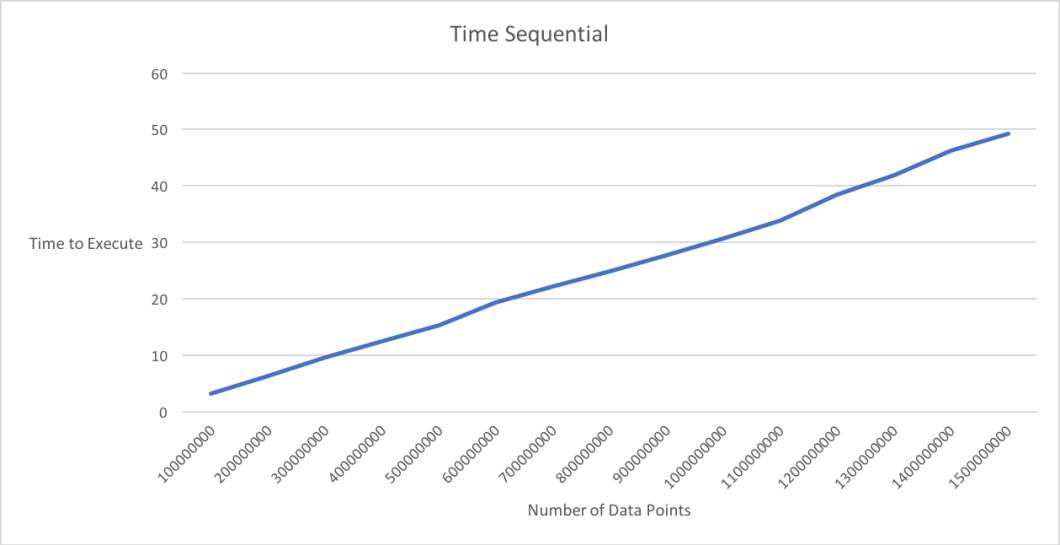


Figure 1: In the graph above, the X-Axis is the size of the data set and the Y-Axis is the time taken to execute. As the numbers get larger, so does the execution time.

Parallel Bucket-Sort

Parallel Bucket Sort is a little different from sequential. The parallel program will treat each processor as a bucket and will do message passing between processors until the buckets have their set of numbers to sort. Below in Figure 2 is a diagram showing how the message passing and putting the numbers in their buckets is done.

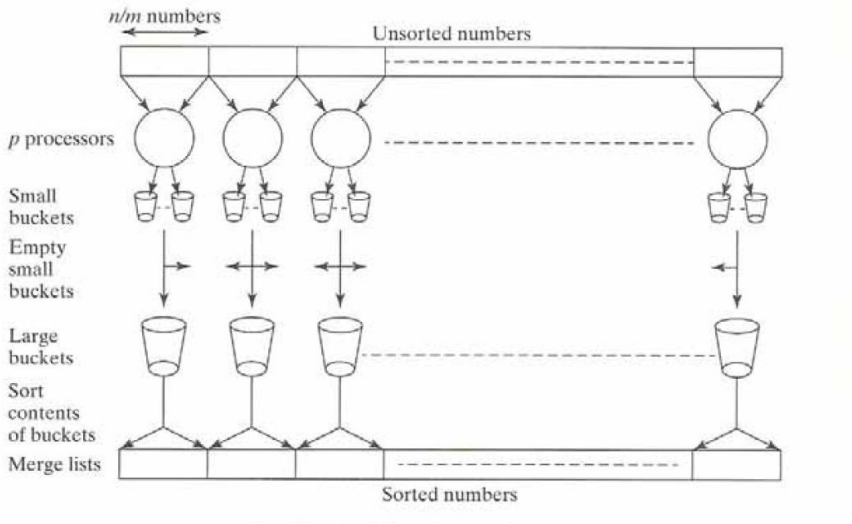


Figure 2: Diagram of how the parallel version will be constructed

When the runtime of the Parallel version of Bucket Sort was gathered, the data was placed in a table, below is Table 2.

	1	4	8	16	24	32
100000000	3.107935	3.079568	2.141458	3.107935	4.978505	7.916933
200000000	6.241437	6.171139	4.40149	6.241437	7.670317	9.676268
300000000	9.631196	9.142299	6.402113	9.631196	11.147209	12.190718
400000000	12.430669	12.495987	8.730779	12.430669	14.55248	16.029563
500000000	15.256826	15.322362	10.883965	15.256826	18.367688	19.516239
600000000	19.32263	18.667056	12.988076	19.32263	21.996029	23.559245
700000000	22.095468	21.196442	15.94541	22.095468	27.640341	26.711453
800000000	24.886312	25.5593	17.892476	24.886312	30.344151	30.24774
900000000	27.706288	28.518634	21.110137	27.706288	35.566851	33.563434
1000000000	30.667014	32.847546	26.41161	30.667014	38.182287	36.633687
1100000000	33.849104	77.930692	52.685423	33.849104	40.200281	40.163538
1200000000	38.541852	148.599888	121.061712	38.541852	43.820486	45.957184
1300000000	41.852368	187.801708	189.414046	41.852368	48.780931	51.010137
1400000000	46.293893	246.496012	226.36414	46.293893	50.990265	55.176352
1500000000	49.1953	289.301691	278.138197	49.1953	58.29016	58.265097

Table 2: In the table above, the top row is the number of cores and the far left column is the number of data points. The data in the middle is the time spent executing the task.

When the data was gathered the following surface was produced. Below in Figure 3 is a diagram of the surface made using Table 2 above.

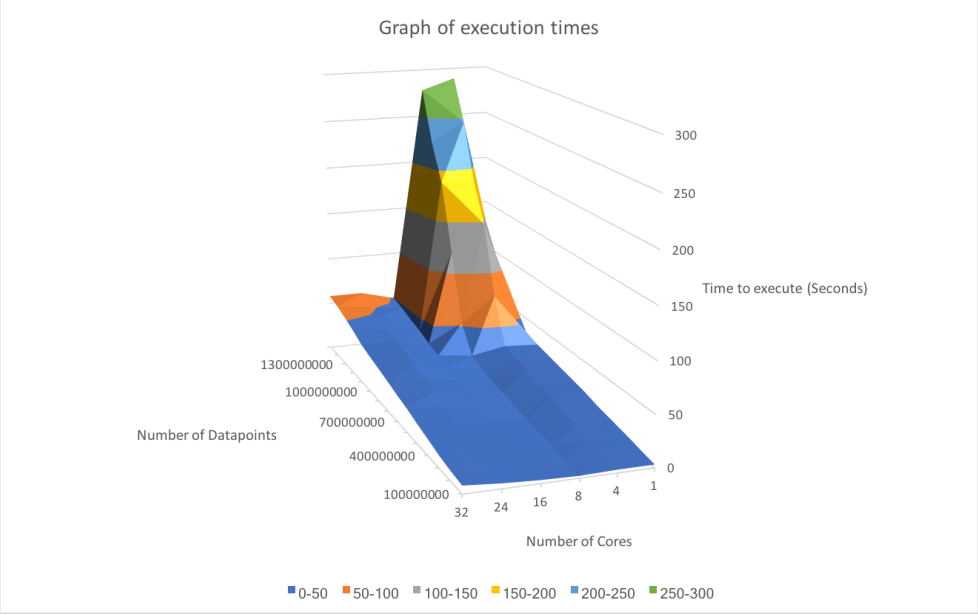


Figure 3: Diagram of the execution times using the processor configurations:
1 Core, 4 Cores, 8 Cores, 16 Cores, 24 Cores, and 32 Cores.

Speed up

After the parallel portion of the program was complete the next piece of information to look at is speed up. Speed up is given as Ts/Tp. Below is a table (Table 3) of the speed up for each process.

	1	4	8	16	24	32
100000000	0	1.00921136	1.45131728	0.92259344	0.62427074	0.39256806
200000000	0	1.01139141	1.41802821	0.97089449	0.81371304	0.64502523
300000000	0	1.05347637	1.5043777	0.97901061	0.86400067	0.79004338
400000000	0	0.99477288	1.42377547	0.97588278	0.85419592	0.77548396
500000000	0	0.99572285	1.40177095	0.98217914	0.83063399	0.78175032
600000000	0	1.0351193	1.48772074	0.9491316	0.87845993	0.82017187
700000000	0	1.04241401	1.38569457	0.98698613	0.79939202	0.82719079
800000000	0	0.97366954	1.3908814	0.99012868	0.82013539	0.82274947
900000000	0	0.97151526	1.31246368	0.98969571	0.77899188	0.82549026
1000000000	0	0.93361659	1.16111869	0.98184989	0.80317384	0.83712606
1100000000	0	0.43434882	0.6424757	0.98667018	0.84201163	0.84278193
1200000000	0	0.25936663	0.31836533	0.95838125	0.87953958	0.83864695
1300000000	0	0.22285403	0.22095705	0.99027323	0.85796575	0.82047159
1400000000	0	0.18780788	0.20451072	1.01336115	0.90789669	0.83901692
1500000000	0	0.17004844	0.17687358	1.00279993	0.84397264	0.84433567

Table 2: In the table above, the top row is the number of cores and the far left column is the number of data points. The data in the middle is speed up for each task

When the data was analyzed and placed into a surface a figure produced which is below looks a little odd.

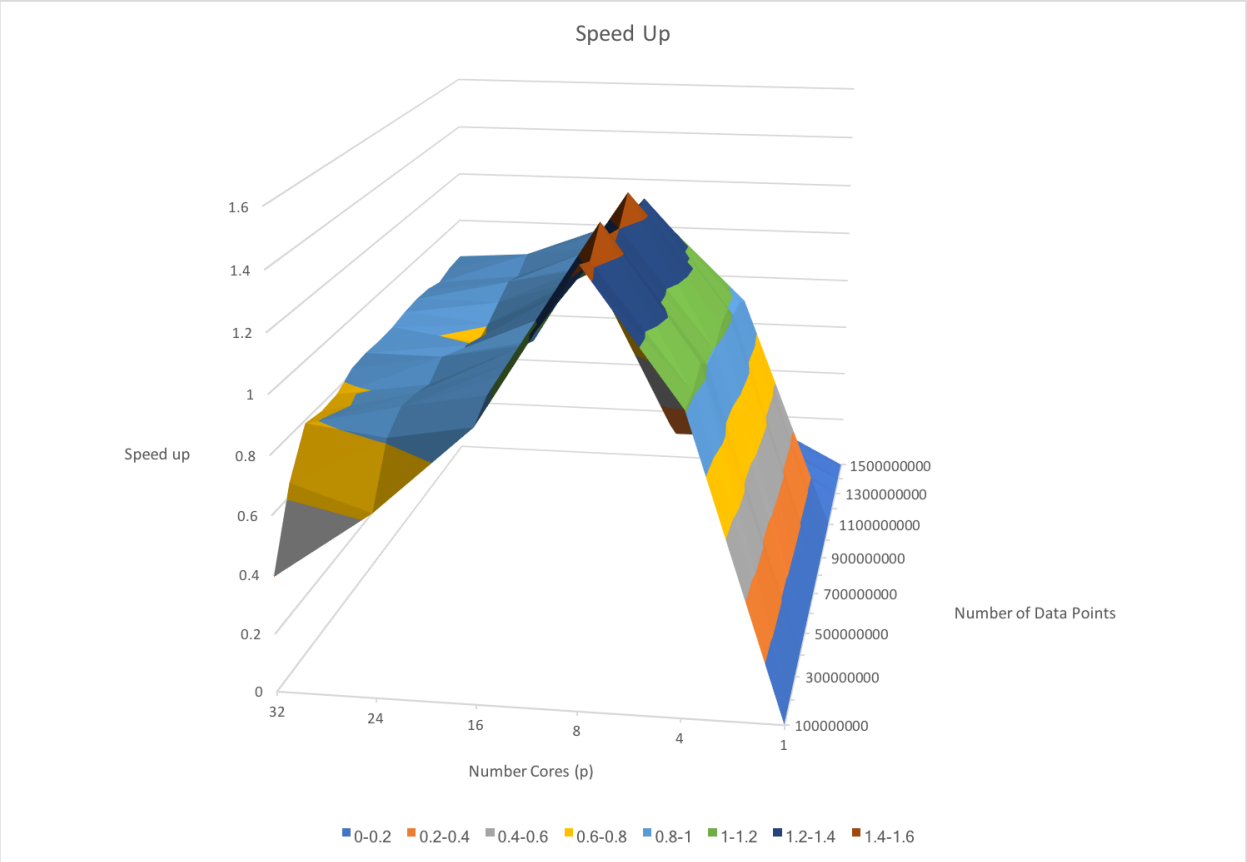


Figure 4: Diagram showing the speed up using each processor configuration (p)

When looking at the number of cores which is presented as ‘p’ in the diagram above (Figure 4) and then taking the speed up; the speed up calculated is not very high. This could be due to the sorting algorithm used for the bucket sort which is standard sort which is part of c++ library <algorithm>.

Conclusion

When looking at the sequential vs parallel portions of the project the fact that very slight speed up happened was a little odd but considering the fact that there is more communication in the parallel version the slight speed up isn’t too out of the ordinary. Below is the source code for the parallel portion for reader convenience.

Source Code:

```
#include <iostream>
#include <mpi.h>
#include <fstream>
#include <vector>
#include <sys/time.h>
#include <stdlib.h>
#include <time.h>
#include <vector>
#include <algorithm>

using namespace std;


// tags and const that are important to the project
#define TERMINATE_TAG 0
#define DATA_TAG 1
#define RESULT_TAG 2
#define MY_MPI_DATA_TAG 3
#define MASTER 0
#define MAX_NUM 100000

// complex struct

// function prototype for calculation
void master(char **argv);
void slave( int taskId );
void genNumbers( int *genArray, int size );

// main function
int main( int argc, char **argv ) {

   int rank;
   int numProcessors;
   MPI_Init( &argc, &argv );
          MPI_Comm_rank( MPI_COMM_WORLD, &rank );

   //int numProcessors;
   int *arr;
   int i, j;
   ////////////////////////////////////////////////////////////////////////////////////////////////
   string fileName = "data.txt";
   int index = 0;
   int size, num;
   double start, finished, dt;
   ////////////////////////////////////////////////////////////////////////////////////////////////
          if( rank == 0 ) // Master
   {
      master(argv);
          }

   ////////////////////////////////////////////////////////////////////////////////////////////////
          else { // Slave
      slave(rank);
   }

   ////////////////////////////////////////////////////////////////////////////////////////////////
          MPI_Finalize();
```

```
            return 0;
}

/////////////// THIS IS FOR THE INCLUDED FUNCTION ////////////

void master(char **argv )
   {
   // Init Variables
   FILE *fpMaster;
   fpMaster = fopen( argv[2], "a+" );

   double t0, deltaTime, end;

   int i;
   int numProcessors;
   int index = 0;
   int index2, index3;
   MPI_Comm_size( MPI_COMM_WORLD, &numProcessors );
   MPI_Status status;
   int capacity = atoi(argv[1]);
   int counter = 0;
   int split = capacity/numProcessors;
   int partition = MAX_NUM / numProcessors;
   int *masterArray = new int[split];
   int delta;
   int bucketPlacement;
   vector <int> myInts[numProcessors];
   vector <int> myBigBucket;
   vector <int> myRecievedBucket;
   int *arr = new int [capacity];
   genNumbers( arr, capacity );


   for( i = 1; i < numProcessors; i++) // 'i' is processor
      {
      MPI_Send(&arr[counter], split, MPI_INT, i, MY_MPI_DATA_TAG, MPI_COMM_WORLD); // Make size/numProcessors a
better variable
      counter += split;
      }

      MPI_Barrier(MPI_COMM_WORLD ); // STOPPED AT MPI_Barrier

     t0 = MPI_Wtime();

  // cout << "Masters numbers is: " << endl;
   delta = capacity - counter;
   while( counter < capacity )
      {
      masterArray[index] = arr[counter];
      counter++;
      index++;
      }


   for( index = 0; index < delta; index++ )
      {
      bucketPlacement = masterArray[index]/partition;
```

```
    if( bucketPlacement >= numProcessors )
    bucketPlacement--;

    myInts[bucketPlacement].push_back(masterArray[index]);
    }

    /* PLACE MY NUMBERS INSIDE BIG BUCKET */


 ////////////////////////////////////////////////////////////////////////////

            /* SENDING AND RECIEVING SMALL BUCKETS */

 ////////////////////////////////////////////////////////////////////////////


 // cout << "Masters small buckets: " << endl;

 for( index = 0; index < numProcessors; index++ )
    {
       if( index == MASTER )
          {
            for( index2 = 0; index2 < numProcessors; index2++ )
               {
                  if( index2 != MASTER )
                     {
                        myRecievedBucket.clear();
                        //myRecievedBucket.resize(0);
                     //   fprintf( fpMaster,  "Probe \n" );
                        MPI_Probe(index2, MY_MPI_DATA_TAG, MPI_COMM_WORLD, &status );

                        MPI_Get_count( &status, MPI_INT, &capacity );

                        myRecievedBucket.resize(capacity);
                     //   fprintf( fpMaster,  "MASTER RECV\n" );
                      //  cout << "Master Recv" << endl;
                        MPI_Recv( &myRecievedBucket[0], capacity, MPI_INT, index2, MY_MPI_DATA_TAG,
MPI_COMM_WORLD, MPI_STATUS_IGNORE ); // '0' needs to be master variable

                     //  fprintf( fpMaster,  "MASTER RECVIED\n" );
                           for( index3 = 0; index3 < capacity; index3++ )
                              {
                                 if( !myRecievedBucket.empty())
                                 myBigBucket.push_back(myRecievedBucket[index3]);
                              }
                     //  fprintf( fpMaster,  "MASTER DONATED TO BUCKET\n" );

                       // MPI_Barrier(MPI_COMM_WORLD); // Stopped at MPI Barrier
                     }


               }


         }
```

```
        else{

            MPI_Send(&myInts[index][0], myInts[index].size(), MPI_INT, index, MY_MPI_DATA_TAG, MPI_COMM_WORLD); //
Make size/numProcessors a better variable

        }


      }


      sort(myBigBucket.begin(), myBigBucket.end());

      MPI_Barrier(MPI_COMM_WORLD); // Stopped at MPI Barrier

      end = MPI_Wtime();


      fprintf(fpMaster, "%d, %f\n", atoi(argv[1]), end - t0 );

   // Free Memory
   delete arr;
   arr = NULL;

   delete masterArray;
   masterArray = NULL;

      myBigBucket.clear();
      myRecievedBucket.clear();

      for( int kill = 0; kill < numProcessors; kill++ )
      myInts[kill].clear();

   }

void slave( int taskId )
   {

   // Init Variables
   int numProcessors;
   int index, index2, index3;
   MPI_Comm_size( MPI_COMM_WORLD, &numProcessors );

   int capacity;
   MPI_Request req;
   MPI_Status status;
   vector <int> myInts[numProcessors];
   int partition = MAX_NUM / numProcessors;
   int small_bucket_index;
   int bucketPlacement;
   vector <int> myBigBucket;
   vector <int> myRecievedBucket;


  //  cout << "Probing " << endl;
   MPI_Probe(MASTER, MY_MPI_DATA_TAG, MPI_COMM_WORLD, &status );
```

```
   MPI_Get_count( &status, MPI_INT, &capacity );

   int *arr = new int [capacity];

   //cout << capacity << endl;
   MPI_Recv( arr, capacity, MPI_INT, 0, MY_MPI_DATA_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE ); // '0' needs to
be master variable

   MPI_Barrier(MPI_COMM_WORLD); // Stopped at MPI Barrier

   for( index = 0; index < capacity; index++ )
      {
        //  fprintf( fpSlave,  "Placement is:  %d \n", arr[index]/partition );
        //  cout << "Placement is: " << arr[index]/partition << endl;
          bucketPlacement = arr[index]/partition;
      if( bucketPlacement >= numProcessors )
      bucketPlacement--;
          myInts[bucketPlacement].push_back(arr[index]);
      }



   ////////////////////////////////////////////////////////////////////////////

               /* SENDING AND RECIEVING SMALL BUCKETS */

   ////////////////////////////////////////////////////////////////////////////

   // Better variable names will suffice

  //  cout << "Slaved small buckets: " << endl;

   for( index = 0; index < numProcessors; index++ )
      {
         if( index == taskId )
            {
               for( index2 = 0; index2 < numProcessors; index2++ )
                  {
                     if( index2 != taskId )
                        {

                           MPI_Probe(index2, MY_MPI_DATA_TAG, MPI_COMM_WORLD, &status );

                           MPI_Get_count( &status, MPI_INT, &capacity );

                           myRecievedBucket.resize(capacity);
                         //   cout << "Slaved Recv" << endl;
                           MPI_Recv( &myRecievedBucket[0], capacity, MPI_INT, index2, MY_MPI_DATA_TAG,
MPI_COMM_WORLD, MPI_STATUS_IGNORE ); // '0' needs to be master variable

                              for( index3 = 0; index3 < capacity; index3++ )
                                 {
                                    if( !myRecievedBucket.empty())
                                    myBigBucket.push_back( myRecievedBucket[index3] );
                                 }
                          //  MPI_Barrier(MPI_COMM_WORLD); // Stopped at MPI Barrier
```

```
                    }


                }


            }

        else{

            MPI_Send(&myInts[index][0], myInts[index].size(), MPI_INT, index, MY_MPI_DATA_TAG, MPI_COMM_WORLD); //
Make size/numProcessors a better variable

        }



    }
    sort(myBigBucket.begin(), myBigBucket.end());
    MPI_Barrier(MPI_COMM_WORLD); // Stopped at MPI Barrier

    myBigBucket.clear();
    myRecievedBucket.clear();
    for( int kill = 0; kill < numProcessors; kill++ )
    myInts[kill].clear();
  //   fclose(fpSlave);

  }

void genNumbers( int *genArray, int size )
  {
  int generatedNum;

  for( int i = 0; i < size; i++ )
    {
      //srand(1000); // Use a seed value
      genArray[i] = rand()%100001;
    }

  }
```