Assignment No. B7

Title:
Implement concurrent ODD-Even Merge sort algorithm.

Aim:
Perform concurrent ODD-Even Merge sort using HPC infrastructure (preferably BBB) using Python/ Scala/ Java/ C++.

Objectives:

- To understand working of Cluster of BBB (BeagleBone Black)
- Implement concurrent ODD-Even Merge sort algorithm.

Prerequisites:
- 64-bit Ubuntu or equivalent OS with 64-bit Intel-i5/i7
- Cluster of BBB, that is two or more BBB

Theory:

Building a Compute Cluster with the BeagleBone Black

Configuring the BeagleBones

Once the hardware is set up and a machine is connected to the network, Putty or any other SSH client can be used to connect to the machines. The default hostname to connect to using the above image is ubuntu-armhf. My first task was to change the hostname. I chose to name mine beaglebone1, beaglebone2 and beaglebone3. First I used the hostname command:

sudo hostname beaglebone1

Next I edited /etc/hostname and placed the new hostname in the file. The next step was to hard code the IP address for so I could probably map it in the hosts file. I did this by editing /etc/network/interfaces to tell it to use static IPs. In my case I have a local network with a router at 192.168.1.1. I decided to start the IP addresses at 192.168.1.51 so the file on the first node looked like this:

```
iface eth0 inet static
address 192.168.1.51
netmask 255.255.255.0
network 192.168.1.0
broadcast 192.168.1.255
gateway 192.168.1.1
```

It is usually a good idea to pick something outside the range of IPs that your router might assign if you are going to have a lot of devices. Usually you can configure this range on your router. With this done, the final step to perform was to edit /etc/hosts and list the name and IP address of each node that would be in the cluster. My file ended up looking like this on each of them:

127.0.0.1 localhost
192.168.1.51 beaglebone1
192.168.1.52 beaglebone2
192.168.1.53 beaglebone3

Creating a Compute Cluster With MPI
After setting up all 3 BeagleBones, I was ready to tackle my first compute project. I figured a good starting point for this was to set up MPI. MPI is a standardized system for passing messages between machines on a network. It is powerful in that it distributes programs across nodes so each instance has access to the local memory of its machine and is supported by several languages such as C, Python and Java. There are many versions of MPI available so I chose MPICH which I was already familiar with. Installation was simple, consisting of the following three steps:

sudo apt-get update
sudo apt-get install gcc
sudo apt-get install libcr-dev mpich2 mpich2-doc

MPI works by using SSH to communicate between nodes and using a shared folder to share data. The first step to allowing this was to install NFS. I picked beaglebone1 to act as the master node in the MPI cluster and installed NFS server on it:

sudo apt-get install nfs-client

With this done, I installed the client version on the other two nodes:

sudo apt-get install nfs-server

Next I created a user and folder on each node that would be used by MPI. I decided to call mine hpcuser and started with its folder:

sudomkdir /hpcuser

Once it was created on all the nodes, I synced up the folders by issuing this on the master node:

echo "/hpcuser *(rw,sync)" | sudo tee -a /etc/exports

Then I mounted the master's node on each slave so they can see any files that are added to the master node:

sudo mount beaglebone1:/hpcuser /hpcuser

To make sure this is mounted on reboots I edited /etc/fstab and added the following:

beaglebone1:/hpcuser /hpcusernfs

Finally I created the hpcuser and assigned it the shared folder:

sudouseradd -d /hpcuserhpcuser

With network sharing set up across the machines, I installed SSH on all of them so that MPI could communicate with each:

sudo apt-get install openssh-server

The next step was to generate a key to use for the SSH communication. First I switched to the hpcuser and then used ssh-keygen to create the key.

su - hpcuser
sshkeygen-t rsa

When performing this step, for simplicity you can keep the passphrase blank. When asked for a location, you can keep the default. If you want to use a passphrase, you will need to take extra steps to prevent SSH from prompting you to enter the phrase. You can use ssh-agent to store the key and prevent this. Once the key is generated, you simply store it in our authorized keys collection:

cd .ssh
cat id_rsa.pub >>authorized_keys

I then verified that the connections worked using ssh:

ssh hpcuser@beaglebone2

Testing MPI
Once the machines were able to successfully connect to each other, I wrote a simple program on the master node to try out. While logged in as hpcuser, I created a simple program in its root directory /hpcuser called mpi1.c. MPI needs the program to exist in the shared folder so it can run on each machine. The program below simply displays the index number of the current process, the total number of processes running and the name of the host of the current process. Finally, the main node receives a sum of all the process indexes from the other nodes and displays it:

```
#include <mpi.h>
#include <stdio.h>
int main(intargc, char* argv[])
{
int rank, size, total;
char hostname[1024];
gethostname(hostname, 1023);
MPI_Init(&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Reduce(&rank, &total, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
printf("Testing MPI index %d of %d on hostname %s\n", rank, size,
hostname);
if (rank==0)
{
printf("Process sum is %d\n", total);
}
MPI_Finalize();
return 0;
}
```

Next I created a file called machines.txt in the same directory and placed the names of the nodes in the cluster inside, one per line. This file tells MPI where it should run:

```
beaglebone1
beaglebone2
beaglebone3
```

With both files created, I finally compiled the program using mpicc and ran the test:

mpicc mpi1.c -o mpiprogram
mpiexec -n 8 -f machines.txt. /mpiprogram

This resulted in the following output demonstrating it ran on all 3 nodes:

```
Testing MPI index 4 of 8 on hostname beaglebone2
Testing MPI index 7 of 8 on hostname beaglebone2
Testing MPI index 5 of 8 on hostname beaglebone3
Testing MPI index 6 of 8 on hostname beaglebone1
Testing MPI index 1 of 8 on hostname beaglebone2
Testing MPI index 3 of 8 on hostname beaglebone1
Testing MPI index 2 of 8 on hostname beaglebone3
Testing MPI index 0 of 8 on hostname beaglebone1
Process sum is 28
```

ODD-Even Merge sort algorithm:

In computing, an odd–even sort or odd–even transposition sort is a relatively simple sorting algorithm, developed originally for use on parallel processors with local interconnections. It is a comparison sort related to bubble sort, with which it shares many characteristics. It functions by comparing all odd/even indexed pairs of adjacent elements in the list and, if a pair is in the wrong order (the first is larger than the second) the elements are switched. The next step repeats this for even/odd indexed pairs (of adjacent elements). Then it alternates between odd/even and even/odd steps until the list is sorted.

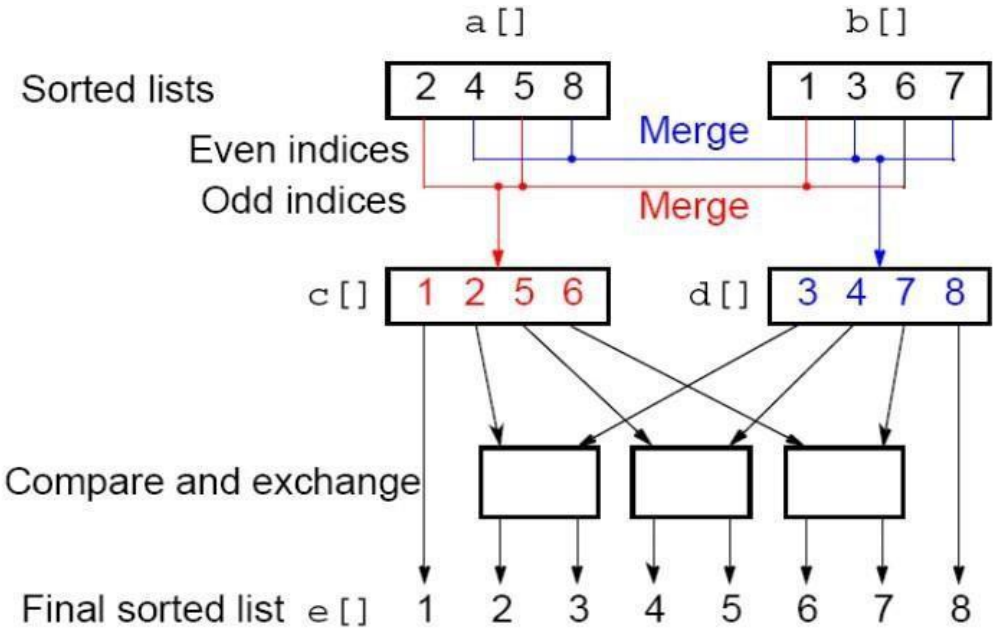
Sorting on processor arrays

On parallel processors, with one value per processor and only local left–right neighbor connections, the processors all concurrently do a compare–exchange operation with their neighbors, alternating between odd–even and even–odd pairings. This algorithm was originally presented, and shown to be efficient on such processors, by Habermann in 1972.

The algorithm extends efficiently to the case of multiple items per processor. In the Baudet–Stevenson odd–even merge-splitting algorithm, each processor sorts its own sublist at each step, using any efficient sort algorithm, and then performs a merge splitting, or transposition–merge, operation with its neighbor, with neighbor pairing alternating between odd–even and even–odd on each step.

1. It starts by distributing n/p sub-lists (p is the number of processors and n the size of the array to sort) to all the processors.
2. Each processor then sequentially sorts its sub-list.
3. The algorithm then operates by alternating between an odd and an even phase :
    1. In the even phase, even numbered processors (processor i) communicate with the next odd numbered processors (processor i+1). In this communication process, the two sub-lists for each 2 communicating processes are merged together. The upper half of the list is then kept in the higher number processor and the lower half is put in the lower number processor.

2. In the odd phase, odd number processors (processor i) communicate with the previous even number processors (i-1) in exactly the same way as in the even phase.



Implementation:
The first step is to distribute a sub list to each process, the master process send to all process a sub-array, to do that we use the MPI_Scatter() function :

```
int *subArray = malloc(N/hostCount * sizeof(int));
if(rank == 0) { /* The master send data to everyone */
MPI_Scatter(arrayToSort,N/hostCount,MPI_INT,subArray,N/hostCount,MPI_INT,0
,
      MPI_COMM_WORLD);
}
```

After that each process has to receive data, we use MPI_Scatterv(), we need 2 particular array : displs that specifies the displacement of sub-array relative to arrayToSort and sendcnts that specifies the number of elements to send to each host.

```
int *displs = malloc(hostCount * sizeof(int));
int i;
for (i=0;i<hostCount;i++) {
      displs[i] = i*(N/hostCount);
}
int *sendcnts = malloc(hostCount * sizeof(int));
for (i=0;i<hostCount;i++) {
      sendcnts[i]=N/hostCount;
}
/* reieve data */
MPI_Scatterv(arrayToSort,sendcnts,displs,MPI_INT,subArray,N/hostCount,MPI_
INT,0,MPI_COMM_WORLD);
free(displs);
free(sendcnts);
```

The next step is to write a sequential sort algorithm. I choose the odd-even sort algorithm. Here is my C function:

```
void sequentialSort(int *arrayToSort, int size) {
int sorted = 0;
while( sorted == 0) {
      sorted= 1;
      int i;
      for(i=1;i<size-1; i += 2) {
            if(arrayToSort[i] >arrayToSort[i+1])
            {
                  int temp = arrayToSort[i+1];
                  arrayToSort[i+1] = arrayToSort[i];
                  arrayToSort[i] = temp;
                  sorted = 0;
            }
      }
      for(i=0;i<size-1;i+=2) {
            if(arrayToSort[i] >arrayToSort[i+1])
            {
                  int temp = arrayToSort[i+1];
                  arrayToSort[i+1] = arrayToSort[i];
                  arrayToSort[i] = temp;
                  sorted = 0;
            }
      }
      }
}
```

The third step is the odd-even phases. For this step we need 2 functions. One function to send data to the next host and keep the lower part of the to arrays. And another one to send data to the previous host and keep the higher part.

```
/* Parameter :
* subArray : an integer array
* size : the size of the integer
* rank : the rank of the host
* Send to the next host an array, recieve array from the next host
* keep the lower part of the 2 array
*/
void exchangeWithNext(int *subArray, int size, int rank)
{
      MPI_Send(subArray,size,MPI_INT,rank+1,0,MPI_COMM_WORLD);
      /* recieve data from the next odd numbered host */
      int *nextArray = malloc(size*sizeof(int));
      MPI_Status stat;
      MPI_Recv(nextArray,size,MPI_INT,rank+1,0,MPI_COMM_WORLD,&stat);
      /* Keep the lower half of subArray and nextArray */
      lower(subArray,nextArray,size);
      free(nextArray);
}
/* Parameter :
* subArray : an integer array
* size : the size of the integer
```

```
* rank : the rank of the host
* Send to the previous host an array, recieve array from the previous host
* keep the higher part of the 2 array
*/
void exchangeWithPrevious(int *subArray, int size, int rank)
{
	/* send our sub-array to the previous host*/
	MPI_Send(subArray,size,MPI_INT,rank-1,0,MPI_COMM_WORLD);
	/* recieve data from the previous host */
	int *previousArray = malloc(size*sizeof(int));
	MPI_Status stat;
	MPI_Recv(previousArray,size,MPI_INT,rank-1,0,MPI_COMM_WORLD,&stat);
	/* Keep the higher half of subArray and previousArray */
	higher(subArray,previousArray,size);
	free(previousArray);
}
```

Then we can write easily the odd-even phase :

```
i =0;
for(i=0;i<hostCount;i++) {
	/* even phase */
	if (i%2==0) {
	/* even numbered host */
		if(rank%2==0) {
		/* even numbered host communicate with the next odd numbered host */
		/* make sure the next odd exits */
			if(rank<hostCount-1) {
			/* send our sub-array to the next odd numbered host
			* receive data from the next odd numbered host
			* Keep the lower half of our array and of the next host
array's
			*/
			exchangeWithNext(subArray,N/hostCount,rank);
			}
	} else {
		/* odd numbered host communicate with the previous even numbered
host */
		/* make sure the previous even exits */
			if (rank-1 >=0 ) {
				/* send our sub-array to the previous even numbered
host
				* receive data from the previous even numbered host
				* Keep the higher half of our array and of the previous
host array's
				*/
				exchangeWithPrevious(subArray,N/hostCount,rank);
			}
		}
	}
	/* odd phase */
	else {
		/* odd host */
		if(rank%2!=0) {
			/* In the odd phase odd numbered host communicate with the
next
```

```
                    * even numbered host make sure the next even exits */
                    if (rank<hostCount-1) {
                    /* send our sub-array to the next even numbered host
                    * receive data from the next even numbered host
                    * Keep the lower half of our array and the next host array's
                    */
                    exchangeWithNext(subArray,N/hostCount,rank);
                    }
             }
             /* even host */
             else {
                    /* In the odd phase even numbered host communicate with the
previous
                    * odd numbered host make sure the previous host exits */
                    if (rank-1 >=0 ) {
                    /* send our sub-array to the previous odd numbered host
                    * receive data from the previous odd numbered host
                    * Keep the higher half of our array and of the previous host
array's
                    */
                    exchangeWithPrevious(subArray,N/hostCount,rank);
                    }
             }
      }
}
```

Now our array is sorted, the lower element own to the host with the rank 0 and the higher to p-1. We just need to gather data, we use MPI_Gather()

```
MPI_Gather(subArray,
N/hostCount,MPI_INT,arrayToSort,N/hostCount,MPI_INT,0,MPI_COMM_WORLD);
```

Performances

The question is: what is the execution time of the parallel algorithm compare to the sequential one. Well it's pretty good, on average in a cluster of 8 machines and for an array of 128 000 elements the parallel algorithm run in 1.50 s, while the sequential one run in 81.71 s on a single machine. We are in a case of super-linearity because sort a sub-array 8 times smaller than the original array run 64 times faster.

Mathematical Model:

Odd-Even Merge sort is used for sorting of given data.

Following parameters are used for Odd-Even Merge sort:

M= {s, e, i, o, n, F, Success, Failure}

s = Start state = inserting numbers to array to be sorted.

e = End state = Sorted list displayed.

i is the set of input element.

is the set of required output.

n = Number of processors = {host names of processors in a file}

F is the set of functions required for Odd-Even Merge sort. = {f1, f2}

f1 = {Odd-Even Cycle}

f2 = {Even-Odd Cycle}

i= {a1, a2, a3,…….., an}. = Array of elements to be sorted.

o={Sorted list of elements by Odd-Even Merge sort}
Success – Sorted list of elements by Odd-Even Merge sort.
Failure-ϕ

Conclusion:

Thus we have implemented concurrent ODD-Even Merge sort algorithm.

Code:

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

int merge(double *ina, int lena, double *inb, int lenb, double *out) {
   int i,j;
   int outcount=0;

   for (i=0,j=0; i<lena; i++) {
       while ((inb[j] < ina[i]) && j < lenb) {
           out[outcount++] = inb[j++];
       }
       out[outcount++] = ina[i];
   }
   while (j<lenb)
       out[outcount++] = inb[j++];

   return 0;
}

int domerge_sort(double *a, int start, int end, double *b) {
   if ((end - start) <= 1) return 0;

   int mid = (end+start)/2;
   domerge_sort(a, start, mid, b);
   domerge_sort(a, mid,   end, b);
   merge(&(a[start]), mid-start, &(a[mid]), end-mid, &(b[start]));
int i;
   for (i=start; i<end; i++)
       a[i] = b[i];

   return 0;
}

int merge_sort(int n, double *a) {
   double b[n];
   domerge_sort(a, 0, n, b);
```

```
   return 0;
}

void printstat(int rank, int iter, char *txt, double *la, int n) {
   printf("[%d] %s iter %d: <", rank, txt, iter);
int i,j;
   for (j=0; j<n-1; j++)
       printf("%6.3lf,",la[j]);
   printf("%6.3lf>\n", la[n-1]);
}

void MPI_Pairwise_Exchange(int localn, double *locala, int sendrank, int recvrank,
                           MPI_Comm comm) {

   /*
    * the sending rank just sends the data and waits for the results;
    * the receiving rank receives it, sorts the combined data, and returns
    * the correct half of the data.
    */
   int rank;
   double remote[localn];
   double all[2*localn];
   const int mergetag = 1;
   const int sortedtag = 2;

   MPI_Comm_rank(comm, &rank);
   if (rank == sendrank) {
       MPI_Send(locala, localn, MPI_DOUBLE, recvrank, mergetag, MPI_COMM_WORLD);
       MPI_Recv(locala, localn, MPI_DOUBLE, recvrank, sortedtag, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
   } else {
       MPI_Recv(remote, localn, MPI_DOUBLE, sendrank, mergetag, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
       merge(locala, localn, remote, localn, all);

       int theirstart = 0, mystart = localn;
       if (sendrank > rank) {
           theirstart = localn;
           mystart = 0;
       }
       MPI_Send(&(all[theirstart]), localn, MPI_DOUBLE, sendrank, sortedtag,
MPI_COMM_WORLD);
int i;
       for (i=mystart; i<mystart+localn; i++)
           locala[i-mystart] = all[i];
   }
}

int MPI_OddEven_Sort(int n, double *a, int root, MPI_Comm comm)
{
   int rank, size, i;
   double *local_a;

// get rank and size of comm
   MPI_Comm_rank(comm, &rank); //&rank = address of rank
   MPI_Comm_size(comm, &size);

   local_a = (double *) calloc(n / size, sizeof(double));


// scatter the array a to local_a
   MPI_Scatter(a, n / size, MPI_DOUBLE, local_a, n / size, MPI_DOUBLE,
       root, comm);
```

```
// sort local_a
   merge_sort(n / size, local_a);

//odd-even part
   for (i = 1; i <= size; i++) {

       printstat(rank, i, "before", local_a, n/size);

       if ((i + rank) % 2 == 0) {  // means i and rank have same nature
           if (rank < size - 1) {
               MPI_Pairwise_Exchange(n / size, local_a, rank, rank + 1, comm);
           }
       } else if (rank > 0) {
           MPI_Pairwise_Exchange(n / size, local_a, rank - 1, rank, comm);
       }

   }

   printstat(rank, i-1, "after", local_a, n/size);

// gather local_a to a
   MPI_Gather(local_a, n / size, MPI_DOUBLE, a, n / size, MPI_DOUBLE,
          root, comm);

   if (rank == root)
       printstat(rank, i, " all done ", a, n);

   return MPI_SUCCESS;
}

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

   MPI_Init(&argc, &argv);

   int n = argc-1;
   double a[n];
int i;
   for (i=0; i<n; i++)
       a[i] = atof(argv[i+1]);

   MPI_OddEven_Sort(n, a, 0, MPI_COMM_WORLD);

MPI_Finalize();

   return 0;
}
```

```
mpiu@DB21:/mirror/mpiu/CL4 prog/B-7$ mpicc odd_even.c
mpiu@DB21:/mirror/mpiu/CL4 prog/B-7$ mpiexec -n 4 -f machinefile ./a.out
[0] before iter 1: < 0.000>
[0] before iter 2: < 0.000>
[0] before iter 3: < 0.000>
[0] before iter 4: < 0.000>
[0] after iter 4: < 0.000>
[1] before iter 1: < 0.000>
[1] before iter 2: < 0.000>
[1] before iter 3: < 0.000>
[1] before iter 4: < 0.000>
[1] after iter 4: < 0.000>
[2] before iter 1: < 0.000>
```

```
[2] before iter 2: < 0.000>
[2] before iter 3: < 0.000>
[2] before iter 4: < 0.000>
[2] after iter 4: < 0.000>
[3] before iter 1: < 0.000>
[3] before iter 2: < 0.000>
[3] before iter 3: < 0.000>
[3] before iter 4: < 0.000>
[3] after iter 4: < 0.000>
[0]  all done  iter 5: < 0.000>
mpiu@DB21:/mirror/mpiu/CL4 prog/B-7$
```