Assignment 1 – MPI Programming

Weiyu Liu, Hang Shao

1. Distributed Sorting:

1.1 Pseudo-code of algorithm:

The algorithm is used in each process.
For input part, firstly we get array's size from main's argument and generate random numbers with this size. The pseudo-code is as follows.

```
MPI_Init();
Get input;
Set input as array_size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &npes);

If(rank==0){
    Set sub_array_size to 0;
}else if(rank != npes-1){
    Set sub_array_size to array_size/npes;
}else{
    Set sub_array_size to array_size - (array_size)/npes*(npes-1);
}

For i = 0: sub_array_size
    Generate a random number random;
    Set subarray[i] to random;
```

In the next step, we added MPI_IO to get data from specified data file. The related pseudo-code is as below.

```
MPI_Init();
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &npes);
Open file in read only mode with mpi_io;
MPI_File_get_size(file, & filesize);
If(rank == 0){
    Set sub_array_size to 0;
}else if(rank ==  npes -1){
    Set sub_array_size to filesize-(npes-2)*filesize/(npes-1);
}

Set size_array to filesize/(npes-1);
MPI_File_seek(fh, (rank-1)*size_array*sizeof(int), MPI_SEEK_SET);
```

```
MPI_File_read(fh, subarray, size_array, MPI_INT, &status);
MPI_File_close(&fh);
```

The sorting part is the same in these two situation.

```
If process is not root {
        Qsort (subarray);// local sort subarray
        Samples = PickSplitters (subarray);// pick samples from subarray in subprocesses
}

MPI_Gather (sample, sampleNum, MPI_INT, rbuf, sampleNum, MPI_INT, ROOT,
MPI_COMM_WORLD) // get all samples from sub-process into root

If process is root{
        receive samples;
        qsort(samples);//local sort all samples
        splitters = PickSplitters(allSamples);//pick splitters from sorted samples
        // from root broadcast  all splitters into each sub-array
        MPI_Bcast(splitters, sampleNum, MPI_INT, ROOT, MPI_COMM_WORLD);
}

If process is not root{
        // subprocesses receive splitters from root
        MPI_Bcast(splitters, sampleNum, MPI_INT, ROOT, MPI_COMM_WORLD);
        //compute send counts for all to all exchange
        for i = 0 : size_array,j = 0 : sample number {
                if subarray[i] <= splitters[j]
                {
                        i++;
                        sendcounts[j+1]++;
                }
                else j++;
        }
        sendcounts[last] = size_array - i;
}

// send send counts to corresponding process's receive counts
MPI_Alltoall(send counts, recieve counts);
based on sendcounts compute sdispels //send displacements array
based on receivecounts compute rdispels //receive displacements array
MPI_Alltoallv(subarray, scount, sdispls, MPI_INT, subarray_sorted, rcount, rdispls,
MPI_INT, MPI_COMM_WORLD); // all to all exchange bins into corresponding subprocess

Qsort(subarray_sorted);
```

For usage of MPI_IO, the algorithm also contains the following part to write result into a file.

```
Set output_file to result file name;
```

```
Set length to the size of subarray_sorted;
Set rbuf to NULL;
MPI_Allgather(&length, 1, MPI_INT, rbuf, 1, MPI_INT, MPI_COMM_WORLD);
MPI_File_open(MPI_COMM_WORLD, output_file,
MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
For i = 0: rank-1
        Set offset to offset + rbuf[i];
Set offset to offset*sizeof(int);
MPI_File_write_at(fh, offset, subarray_sorted, length, MPI_INT, &status);
MPI_File_close(&fh);

MPI_Finalize();

End;
```

1.2 Performance:

Our posted task in Kingspeak cluster is in idle status and thus we tested our code locally with partial data from $SCRATCH/data/sort_data_debug.dat and the result showed its correctness.

2. Graph Coloring:

2.1 We chose hash_map to represent the graph with each node as its keys and associated neighbors as values. If notating the number of nodes as n and edges as e, the memory used to store the graph will be (n + 2*e)*sizeof(int). If a matrix used for the same graph instead, its required memory size will be n*n*sizeof(int), which is larger than the one with hash_map for all listed graphs.

2.2 In the algorithm, it needs to check the neighbors of nodes from other processes. As a result, we use fstream from c++ library to read the whole graph into each process.

2.3 We are not able to get the optimal coloring for a representative set of graphs.

2.4 Performance:

Our posted tasks have not gotten their slots totally. As a result, we tested our code with different data set locally. The results are as follows:

Data Set	Thread Num / Core Num	Time Consumed Per Thread (s)
le450_5a.col	4/2	2.4
le450_15b.col	4/2	3.6
le450_25a.col	4/2	3.5

Table 2.1 Results of Graph Coloring Test

3. Maximum Sub-array Sum:

3.1 The pseudo-code of a divide & conquer algorithm is as follows:

```
Max_crossing_subarray(a,1,n/2,n){
        Set left_sum to -inf;
        Set sum to 0;
        for i = n/2 : -1: 0{
```

```
                Set sum to sum+A[i];
                if sum > left_sum
                        Set left_sum_sum to sum;
        }
        Set right_sum to -inf;
        Set sum to 0;
        for i = n/2+1 : 1: n{
                Set sum to sum+A[i];
                if sum > right_sum
                        set right_sum_sum to sum;
        }
        return left_sum + right_sum;
}

Max_subarray(a,low,high){
        if low == high
                return a[low];
        else{
                Set mid to (low + high) / 2;
                Set left_sum to Max_subarray (a,low,mid);
                Set right_sum to Max_subarray(a,mid+1,high);
                Set cross_sum to Max_crossing_subarray(a,low,mid,high);
                return max{left_sum,right_sum,cross_sum};
        }
}
```

The pseudo-code of linear algorithm is as below:

```
Max_subarray_linear(a,n){
        Set sum, current_max to 0;
        for k = 1:n
                if sum + a[k] > 0
                        Set sum to sum +a[k];
                        if sum > current_max
                                Set current_max to sum;
                else
                        Set sum to 0;
        return  current_max;
}
```

3.2 The pseudo-code for parallel divide & conquer algorithm is:

```
prefix_sum(array,length,npes,subarray,MIP_COMM_WORLD){
        prefixSum;
        if length == 1
                Set prefixSum to array;
        total_prefix;
```

```
        MPI_SCATTER (array,length / npes,MPI_INT,subarray,length /
                neps,MPI_INT,MIP_COMM_WORLD);
        for i = 0 : (length / noes-1){
                Set prefix [i] to prefix sum of subarray;
        }
        MPI_SCAN(prefix[length/npes-1],total_prefix,1,datatype,MPI_SUM,comm );
        for m = 0 : (length / noes-1){
                Set prefix[m] to total_prefix - prefix[length / noes-1];
        }
        return prefix;
}

suffix_sum(array,length,npes,subarray,MIP_COMM_WORLD){
        suffixSum;
        if length == 1
                Set suffixSum to array;
        total_suffix;
        for i = (npes-1) : 0{
                Set sendcounts[i] to length / npes ;
                Set displs[i] to ( length / npes - 1);
        }

        MPI_SCATTERV (array,sendcounts,displs,MPI_INT,subarray,length /
                npes,MPI_INT,length / neps,MPI_INT,root,MIP_COMM_WORLD);
        for i = (length / noes-1) : 0{
                Set suffix [i] to suffix sum of subarray;
        }
        MPI_SCAN(suffix[length/npes-1],total_suffix,1,datatype,MPI_SUM,comm );
        for m = 0 : (length / noes-1){
                Set suffix[m] to total_suffix - suffix[0];
        }
        return suffix;
}

Max_crossing_subarray(a,1,n/2,n){
        Set max_prefix_process to max(prefix_sum(a[n/2:n]));
        Set max_suffix_process to max(suffix_sum(a[1:n]));
        scan (max_prefix_process, max_prefix, 1, MPI_INT, MPI_MAX, root,comm);
        scan (max_suffix_process, max_suffix, 1, MPI_INT, MPI_MAX, root,comm);
        return max_suffix + max_prefix;
}

Max_subarray(a,low,high){
        if low == high
                return a[low];
        else{
```

```
                Set mid to (low + high) / 2;
                Set left_sum to Max_subarray (a,low,mid);
                Set right_sum to Max_subarray(a,mid+1,high);
                Set cross_sum to Max_crossing_subarray(a,low,mid,high);
                return max{left_sum,right_sum,cross_sum};
        }
}

prefix_sum(array,length,npes,subarray,MIP_COMM_WORLD){
        prefixSum;
        if length == 1
                set prefixSum to array;
        total_prefix;
        MPI_SCATTER (array,length / npes,MPI_INT,subarray,length /
                neps,MPI_INT,MIP_COMM_WORLD);
        for i = 0 : (length / noes-1){
                if prefix [i -1] + array[i]  > 0
                        set prefix [i] to prefix sum of subarray;
                else set prefix[i] to 0;
        }
        MPI_SCAN(prefix[length/npes-1],total_prefix,1,datatype,MPI_SUM,comm );
        for m = 0 : (length / noes-1){
                if prefix + total_prefix - prefix[length / noes-1] > 0
                        set prefix[m] to prefix[m]+ total_prefix - prefix[length / noes-1];
                else
                        set prefix[m] to 0;
        }
        return prefix;
}

Max_subarray_linear(a,n){
        Set prefix to prefix_sum(a);
        Set max_process to max(prefix);
        scan (max_process, max, 1, MPI_INT, MPI_MAX, root,comm);
        return  max;
}
```

3.3 The best algorithm should use n/log(n) processes, where n is the size of array, and size of each subarray in processes is log(n).

It is called EREW PRAM algorithm, which has the following steps:

A. for all Pi, where i <- 1 to n/log(n) in parallel , do compute prefix sum as sum[1…n],end;
B. for all Pi, where i <- 1 to n/log(n) in parallel , do compute min[1…n], where min[i] = MIN{sum[0],…,sum[i-1]},end;
C. for all Pi, where i <- 1 to n/log(n) in parallel , do compute cand[1…n], where cand[i] = sum[i] - min[i],end;

D. for all Pi, where i <- 1 to n/log(n) in parallel , do compute M[1…n], where M[i] = MAX{cand[1], … , cand[i]}, end;
E. output M[n];