CS309 ASSIGNMENT2

Krishanu Saini - (190001029)

September 17, 2021

1 Problem Statement

In this assignment, you will develop MPI program for the following tasks.

1. Parallel merge sort starts with n/comm_size keys assigned to each process. It ends with all the keys stored on process 0 in sorted order.
   To achieve this, it uses the same tree-structured communication that we used to implement a global sum.

   However, when a process receives another process' keys, it merges the new keys into its already sorted list of keys.

   Notes

   (a) Write a program that implements parallel mergesort. Process 0 should read in n and broadcast it to the other processes.

   (b) Each process should use a random number generator to create a local list of n/comm_size ints.

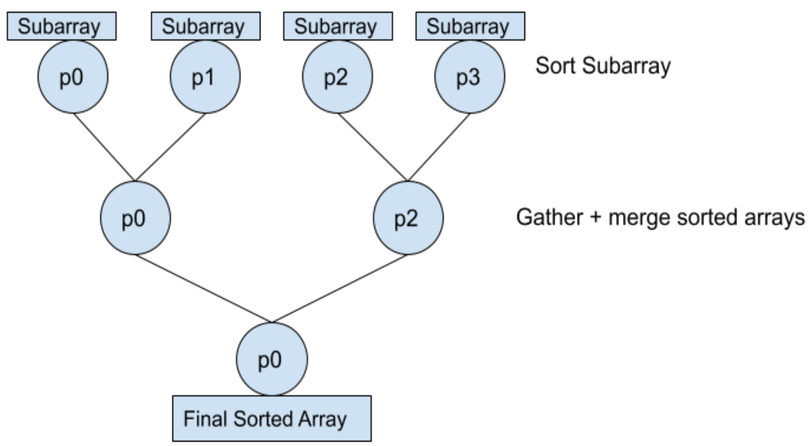
   (c) Each process should then sort its local list, and process 0 should gather and print the local lists.

   (d) Then the processes should use tree-structured communication to merge the global list onto process 0, which prints the result. (MPI, 20 marks)

2 Solution

Algorithm

1. Input size of array - n. Broadcast it
2. Generate random array of $n/world\_size$ integers
3. Sort subarray in each processor.
4. Merging to be done like balanced binary tree
5. Print result in master processor (0).



3 Analysis

3.1 Time Complexity

Let n be number of elements, p be the number of processors.

1. Input of array $O(n/p)$ per processor
2. Sort subarray $O(n/p \cdot log(n/p))$
3. Merging sorted arrays $O(n)$
   There are $log_2(p)$ levels. Each level requires $O(2*size(prev\_subarray))$.
   ie T = $2n/p \cdot (1 + 2 + ... + 2^{log(p)-1}) = 2n/p \cdot 2^{log(p)} - 1 = 2n$
4. Communication time = $log(p)$

Net Time Complexity = $O(n/p + log(p) + n/p \cdot log(n))$

3.2 Space Complexity

1. Input array total O(n)
2. Subarray buffer O(n/world_size)
3. Temporary arrays max $n/p \cdot (p + p/2 + p/4 + ...$+log(p) times) =$O(n)$

Net Space Complexity = $O(n)$

4 Code

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

int randInt()
{
  return rand() % 1000;
}

  /********** Merge Function **********/
void merge(int *a, int *b, int l, int m, int r)
{

  // printf("][][] array - ");
  // for(int i=l;i<=r;i++){
  //   printf("%d ", a[i]);
  // }
  // printf("\n");
  int h, i, j, k;
  h = l;
  i = l;
  j = m + 1;

  while ((h <= m) && (j <= r))
  {

    if (a[h] <= a[j])
    {

      b[i] = a[h];
      h++;
    }

    else
    {
```

```
      b[i] = a[j];
      j++;
    }

    i++;
  }

  if (m < h)
  {

    for (k = j; k <= r; k++)
    {

      b[i] = a[k];
      i++;
    }
  }

  else
  {

    for (k = h; k <= m; k++)
    {

      b[i] = a[k];
      i++;
    }
  }

  for (k = l; k <= r; k++)
  {

    a[k] = b[k];
  }
}

/********** Recursive Merge Function **********/
void mergeSort(int *a, int *b, int l, int r)
{
```

```
  int m;

  if (l < r)
  {

    m = (l + r) / 2;

    mergeSort(a, b, l, m);
    mergeSort(a, b, (m + 1), r);
    merge(a, b, l, m, r);
  }
}


int main(int argc, char *argv[])
{
  int world_rank, world_size;
  double time1, time2,duration;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  srand(world_rank+1);
  /*---------- Take Input ----------*/
  int n;
  if (world_rank == 0)
  {
    printf("Input array size(n): ");
    scanf("%d", &n);
  }


  time1 = MPI_Wtime();
  /*---------- Broadcast ----------*/
  MPI_Bcast(&n, sizeof(int), MPI_INT, 0, MPI_COMM_WORLD);

  /*---------- Scatter ----------*/
  if (n % world_size != 0)
  {
    printf("Use correct number of processors\n");
    exit(0);
```

```
}

int r = n / world_size;
int *subarray = (int *)calloc(r, sizeof(int));
int *sorted = (int *)calloc(r, sizeof(int));

int sizeSorted = r;

// MPI_Scatter(arr, r, MPI_INT, subarray, r, MPI_INT, 0,
   MPI_COMM_WORLD);

printf("Processor %d: subarray: ", world_rank);
for (int i = 0; i < r; i++)
{
  subarray[i] = randInt();
  printf("%d ", subarray[i]);
  sorted[i] = subarray[i];
}
printf("\n");

mergeSort(subarray, sorted, 0, r-1);

/*---------- tree based merging -----------*/
/*
   n n n n n n n n
   |/  |/  |/  |/
   n   n   n   n
   |  /    |  /
   n       n
   | _____/
   n
*/

int cur_world_size = world_size;
int np = 2;
while(cur_world_size > 1){
  if(world_rank % (np/2) != 0) {
    break;
  }
  int r;
```

```
  int *other_sorted = (int* )malloc(sizeSorted*sizeof(int));
  if(world_rank % np == (np/2)){
    r = sizeSorted;
    for(int i=0;i<r;i++){
      other_sorted[i] = sorted[i];
    }
    MPI_Send(&r, 1, MPI_INT, world_rank-np/2, 5, MPI_COMM_WORLD);
    MPI_Send(other_sorted, r, MPI_INT, world_rank-np/2, 11,
        MPI_COMM_WORLD);
  } else if(world_rank/(np/2) != cur_world_size-1){ // last one
      excluded if cur_world_size odd
    MPI_Recv(&r, 1, MPI_INT, world_rank+np/2, 5, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
    MPI_Recv(other_sorted, r, MPI_INT, world_rank+np/2, 11,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    /*---------- merging 2 sorted arrays -----------*/
    int r1 = sizeSorted;
    sorted = (int* )realloc(sorted, (r+r1)*sizeof(int));
    sizeSorted = r+r1;
    for(int i=0;i<r;i++){
      sorted[r1+i] = other_sorted[i];
    }
    int* newSorted = (int*)malloc(sizeSorted*sizeof(int));
    merge(sorted, newSorted, 0, r1-1, r+r1-1);
    sorted = newSorted;

  }
  // MPI_Barrier(MPI_COMM_WORLD);
  np *= 2;
  cur_world_size = (cur_world_size+1) / 2;
}

if(world_rank == 0){
  time2 = MPI_Wtime();

  duration = time2 - time1;

  printf("Sorted array - ");
  int r = sizeSorted;
```

```
    for(int i=0;i<r;i++){
      printf("%d ", sorted[i]);
    }
    printf("\n");
    printf("Time taken by program = %0.9f\n", duration*1e6);

  }

  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();
  return 0;
}
```

5 OUTPUT

```
krishanu2001@LAPTOP-V4CKFTKN:/mnt/c/Users/krishanu/Desktop/sem5/PARALLEL/mpi/lab3$ mpicc merge_sort_mpi.c -o merge_sort_mpi
krishanu2001@LAPTOP-V4CKFTKN:/mnt/c/Users/krishanu/Desktop/sem5/PARALLEL/mpi/lab3$ mpirun -n 2 ./merge_sort_mpi
Input array size(n): 4
Array: 560 568 921 968
Processor 0: subarray: 560 568
Processor 1: subarray: 921 968
Sorted array - 560 568 921 968
```

```
krishanu2001@LAPTOP-V4CKFTKN:/mnt/c/Users/krishanu/Desktop/sem5/PARALLEL/mpi/lab3$ mpirun -n 4 ./merge_sort_mpi
Input array size(n): 8
Array: 322 275 193 599 79 336 97 43
Processor 0: subarray: 322 275
Processor 1: subarray: 193 599
Processor 2: subarray: 79 336
Processor 3: subarray: 97 43
Sorted array - 43 79 97 193 275 322 336 599
```

```
krishanu2001@LAPTOP-V4CKFTKN:/mnt/c/Users/krishanu/Desktop/sem5/PARALLEL/mpi/lab3$ mpirun -n 4 ./merge_sort_mpi
Input array size(n): 16
Array: 239 963 707 228 52 217 514 875 820 177 491 210 375 761 678 51
Processor 0: subarray: 239 963 707 228
Processor 1: subarray: 52 217 514 875
Processor 2: subarray: 820 177 491 210
Processor 3: subarray: 375 761 678 51
Sorted array - 51 52 177 210 217 228 239 375 491 514 678 707 761 820 875 963
```

6 Explanation

Comparing parallel algorithm runtime vs serial.

n	p	Parallel(μs)	Serial(μs)
10	2	32	45
100	4	40	100
1k	4	87	144
10k	4	1029	3250
100k	4	4646	11544

Note: We observe speedup is not exactly p times. There is some over head due to communication and memory delays. For larger problem size - speedup is better.

Comparing Speedup vs processors

n	100	1k	10k	100k
p=1	14(μs)	98	1133	13025
p=2	35	93	1030	6945
p=4	65	83	472	4716
p=10	95	105	415	4344