Indian Institute of Technology Indore

Report for Lab Assignment 3

Parallel Computing Lab (CS 359)

Author
Aniket Sangwan (cse180001005@iiti.ac.in)

April 13, 2021

Contents

1 Report **1**
1.1 Problem Statement 1
1.2 Approach 1
1.3 Input 2
1.4 Output 2
1.5 Complexity 3
1.6 Runtime Comparsion 4
1.7 Conclusion 4
1.8 Implementation 4

Chapter 1

Report

1.1 Problem Statement

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. Write a program that implements parallel mergesort. Process 0 should read in n and broadcast it to the other processes. Each process should use a random number generator to create a local list of n/comm_size ints. Each process should then sort its local list, and process 0 should gather and print the local lists. Then the processes should use tree-structured communication to merge the global list onto process 0, which prints the result.

1.2 Approach

Merge Sort is a divide and conquer algorithm which divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The tree structured communication that we used to implement global sum is used in the parallel merge sort. Whenever a process receives a sorted array from some process, it merges its array with the received array using Merge() function in such a way that the resulting array is sorted as well.

In this algorithm, the master process receives the size of array, n, as input from the user. Once this value is received by the other processes, they randomly generate an array of size n/p where p is the number of processes. This array is locally sorted using sort() function. Once the array is generated, each process calls the below function for tree based communication.

```
void Merge_sort(int A[], int local_n, int my_rank, int p, MPI_Comm comm){

  int partner,done=0,size=local_n;
  unsigned bitmask=1;
  int *B,*C;
  MPI_Status status;

```

```
    B=new int[p*local_n];
    C=new int[p*local_n];

    while(!done && bitmask<p){

        //Selecting slave process using bitmasking
        partner=my_rank^bitmask;
        if(my_rank>partner){

            // sorted array is sent to the master process
            MPI_Send(A, size, MPI_INT, partner, 0, comm);
            done=1;

        }else{

            //Master process receives the sorted array.
            MPI_Recv(B, size, MPI_INT, partner, 0, comm, &status);

            Merge(A, B, C, size);
            size=2*size;
            bitmask<<=1;

        }
    }

    free(B);
    free(C);
}
```

The arguments which are provided to the function are: The locally sorted array, size of this local sorted array, rank/id of the process, total number of processes and MPI Communication world object. bitmask helps in selecting the process that would communicate with the current process. For example, initially bitmask pairs processes with id 0 and 1 together. But in the next step, bitmask becomes 2. Thus, It pairs the process 0 with process $(0|2) = 2$ because, in the previous step, array from the process 1 was merged with the array from process 0. This method is followed until we get the final sorted array in the master process.

1.3 Input

- The first line of input contains n, the number of elements in the array.

  For example: 100

1.4 Output

- The first line outputs the time taken by the parallel merge sort program.
- To print the initial and final array, uncomment lines 95 and 166 in the program.

The image below shows sample input and output for a small test case.

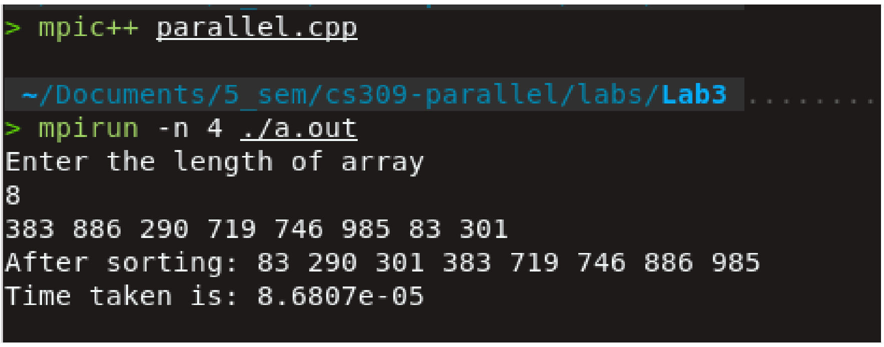


Figure 1.1: Serial Program

1.5 Complexity

- The time complexity of the serial program is $\mathbf{O(n \cdot log(n))}$, where n is the number of elements in the array.
- For parallel merge sort, assume that the number of processes are p. Then the time taken by each processor at Step 1 will be $(n/p) \cdot \log(n/p)$, because each process will have n/p elements to sort.

  At next step, the number of processors will be reduced to half. Thus the time taken will be $2 \cdot (n/p)$ (We need to merge two arrays of size n/p). The total length of tree will be $log_2 p - 1$. Hence, time taken will be

$$T = \frac{n}{p} \cdot log\frac{n}{p} + 2 * n/p + 4 * n/p + \ldots (log_2 p - 1) times \tag{1.1}$$

$$T \equiv \frac{n}{p} \cdot log\frac{n}{p} + O(n) \tag{1.2}$$

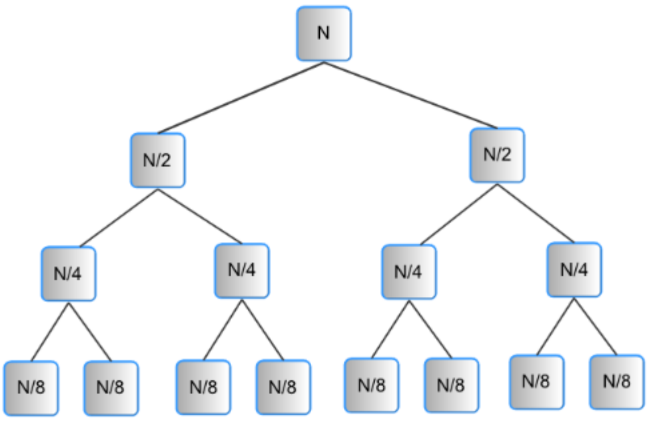


Figure 1.2: Serial Program

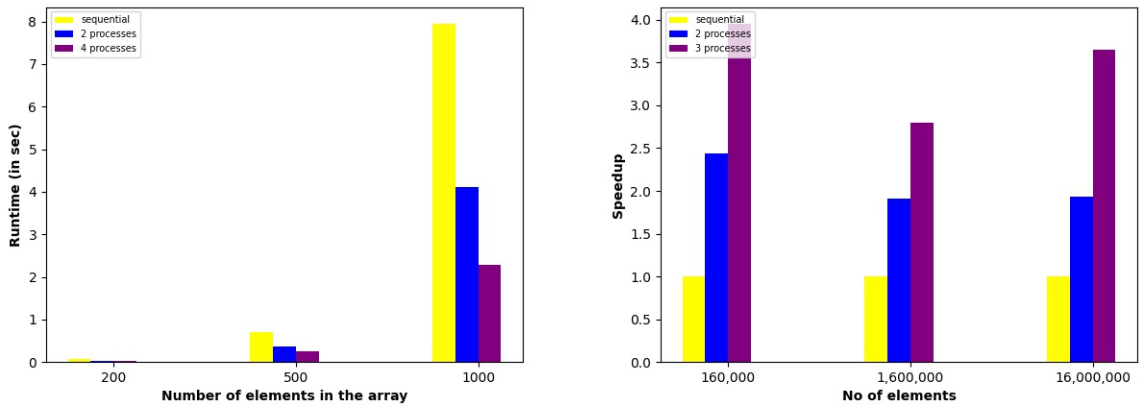
- Therefore, the time complexity is $O(\frac{n}{p} \cdot log\frac{n}{p} + n)$, where p is the number of processors available.

1.6 Runtime Comparsion

To evaluate the performance for different cases, I used **three different values for number of elements in the array** (160000, 1600000 and 16000000). I compared the runtime of each test case for serial and parallel programs and generated graphs to visualise the performance.

	Runtime (in sec)		
Array Length	1 Process	2 Processes	4 Processes
160,000	0.0810207	0.0332742	0.0205079
1,600,000	0.712934	0.374044	0.254543
16,000,000	7.94048	4.10953	2.27585

Table 1.1: Run-Times



(a) Runtime vs No of elements

(b) Speedup vs No of elements

Figure 1.3

The above figures represent the changes in runtime and speedup with the change in number of elements of the array and change in number of processes.

1.7 Conclusion

We conclude that runtime increases with increase in the size of the array. Also, runtime can be significantly reduced by increasing the number of processes. There is a slightly less change in speedup when the number of processes are taken as 4 due to hardware limitations.

1.8 Implementation

```
/*
 Author: 180001005 (Aniket Sangwan)
 */
```

```

#include<bits/stdc++.h>
#include<mpi.h>
using namespace std;

/* MAX-1 is the largest possible value of any element in the array*/
const int MAX = 1000;

// Merges two equal sized arrays A and B in the list A. C acts as a
    temporary array.
void Merge(int A[], int B[], int C[], int size){

    int a1=0,b1=0,c1=0;

    while(a1<size && b1<size){

       if(A[a1]<=B[b1]){

           C[c1]=A[a1];
           c1++;a1++;

       }else{

           C[c1]=B[b1];
           c1++;b1++;
       }
    }

    //Remaining elements are inserted
    if(a1>=size){
       for(;c1<2*size;c1++,b1++)  C[c1]=B[b1];
    }else{
       for(;c1<2*size;c1++,a1++)  C[c1]=A[a1];
    }

    memcpy(A, C, 2*size*sizeof(int));

}

// Parallel Merge Sort using tree-structured communication
void Merge_sort(int A[], int local_n, int my_rank, int p, MPI_Comm comm){

    int partner,done=0,size=local_n;
    unsigned bitmask=1;
    int *B,*C;
    MPI_Status status;

    B=new int[p*local_n];
    C=new int[p*local_n];

    while(!done && bitmask<p){

       //Selecting slave process using bitmasking
       partner=my_rank^bitmask;
       if(my_rank>partner){

           // sorted array is sent to the master process
           MPI_Send(A, size, MPI_INT, partner, 0, comm);
```

```
            done=1;

        }else{

            //Master process receives the sorted array.
            MPI_Recv(B, size, MPI_INT, partner, 0, comm, &status);

            Merge(A, B, C, size);
            size=2*size;
            bitmask<<=1;

        }
    }

    free(B);
    free(C);
}

// Outputs the array
void Print_list(int A[], int n){
    int i;

    for(i=0;i<n;i++) printf("%d ",A[i]);
    printf("\n");

}

// Prints the contents of the global array
void Print_global_list(int A[], int local_n, int my_rank, int p, MPI_Comm
    comm){

    int* global_A=NULL;

    if(my_rank==0){

        //Parent process gathers array from every child
        global_A=new int[p*local_n];
        MPI_Gather(A, local_n, MPI_INT, global_A, local_n, MPI_INT, 0, comm);

        // cout<<"Initial global array: ";
        // Print_list(global_A, p*local_n);
        free(global_A);

    }else{

        // Locally sorted array is sent to the master process by every child
    process
        MPI_Gather(A, local_n, MPI_INT, global_A, local_n, MPI_INT, 0, comm);

    }

}



// Random array generator for child processes
void Generate_list(int A[], int local_n, int my_rank){

```

```
    int i;

     srandom(my_rank+1);
     for(i=0;i<local_n;i++) A[i]=random() % MAX;
     sort(A, A+local_n);

}


/*---------------------------------------------------------------------------*/
int main(int argc, char* argv[]) {
    int my_rank, p; //Current process rank and no of processes
    int* A; // Stores the array
    int n,local_n;
    MPI_Comm comm;

    MPI_Init(&argc, &argv);
    comm=MPI_COMM_WORLD;
    MPI_Comm_size(comm, &p);
    MPI_Comm_rank(comm, &my_rank);

    if(ceil(log2(p))!=floor(log2(p))){ cout<<"Number of processes should be a power of 2. Rerun the program\n";exit(1);}

    if(my_rank==0){

       cout<<"Enter the length of array\n";
       cin>>n;
       if(n%p!=0){ cout<<"Length of array should be divisible by the number of processes p\n";exit(1);}

    }

    MPI_Barrier(comm);
    double start=MPI_Wtime(); //Starts the timer

    MPI_Bcast(&n, 1, MPI_INT, 0, comm); // Broadcasts the no of elements in the array

    local_n=n/p; // No of elements for each process

    A=new int[p*local_n];

    Generate_list(A, local_n, my_rank);

    Print_global_list(A, local_n, my_rank, p, comm);

    Merge_sort(A, local_n, my_rank, p, comm);

    double end=MPI_Wtime()-start; // Total runtime

    double max_time;

    MPI_Reduce(&end, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD); // Gets the maximum of time taken by every process


    if (my_rank==0){
```

```
        // cout<<"After sorting: ";
        // Print_list(A, p*local_n);
        cout<<"Time taken is: "<<max_time<<endl;

    }

    free(A);

    MPI_Finalize();

    return 0;

}
```