Group 4
Brian Tiner
Adam Gumieniak
Matthew Buchanan
Duc Trung Nguyen

CP431 A2 Parallel Merge of Sorted Arrays

Our algorithm is based on course notes. To demonstrate the correctness of our data partitioning and binary search we populated the arrays with random numbers to show that our algorithm accepts robust array inputs. We then created a work division function to partition the task between processors. It divides array A equally based on the number of processes with the standard work division algorithm. It then takes array B and divides it based on the upper value in each section of array A. This is done using a binary search. Output array C is partitioned based on the combined sizes of the sections of arrays A and B on each processor and the partitions of C by each process with a rank less than itself. Each processor sends its higher C index to the next processor so it can set its lower bound accordingly. This round of communication is done sequentially.

We then use a while loop to run through these two arrays and merge values into the result array correctly. It runs until all of the values from one array has been put into the result array. It then runs through the array that still has values in it and puts them all into the result array.

It then uses the slave master approach. The 0 process computes its own section of the array and then receives data from all the other processes. Each other process computes its part of the array then sends it to the 0 process. All the data gets combined in the 0 process for the final merged array.

Results

Number of Processors	**Input Size**	**CPU Wall Time(Seconds)**
4	1000000	9
4	10000000	8
4	100000000	6
8	1000000	9
8	10000000	6
8	100000000	6
16	1000000	9
16	10000000	8
16	100000000	8

Observations

According to our results, we correctly merge the arrays but the speedup of this algorithm does not scale as you increase the number of processors. It also gets faster as you increase the size of the problem. This could be because we generate random numbers for the input every time the program is run. In some edge cases the generation might create bad partitioning when we conduct the work division which makes the program run slower. If the inputs were hardcoded and partitioned in a more efficient manner then the speedup may have been represented better.

Output

Problem Size	**Index 0**	**Index 1**	**Index 2**	**Index n - 2**	**Index n - 1**	**Index n**
1000000	8	13	14	3003098	3003103	3003108
10000000	14	16	18	30004292	30004293	30004295
100000000	16	17	18	300010564	300010567	300010571

All outputs here are tested on 8 cores. To show our algorithm is correct we printed the first and last three indexes of the merged array. Since the indexes are printed in order, we assume the entire array is also sorted.

Source Code

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

//we write into out first and last
void work_division(int n, int p, int rank, int *out_first, int *out_last){
	int delta = n/p;
	int range_start = delta * rank;
```

```
        int range_end = delta * (rank + 1) - 1;
        if (rank < n%p){
                range_start += rank;
                range_end += rank + 1;
        }
        else{
                range_start += n%p;
                range_end += n%p;
        }
        *out_first = range_start;
        *out_last = range_end;
}

int binary_search(int *a, int n, int v){
   int orig_first = 0;
   int first = 0;
   int last = n - 1;
   int middle = (first + last)/2;
   while (first < last) {
      if (a[middle] <= v){
         first = middle + 1;
           }
      else{
         last = middle - 1;
           }
      middle = (first + last)/2;
   }

        if (a[middle] <= v){
                return first;
        }
   else if(first - 1 < orig_first){
                return -1;
        }
        else{
                return first - 1;
        }
}
```

```
int main(int argc, char **argv){
	int n = 1000000; //problem size
	Int *temp, *a, *b, *merged_arr; //arrays
	int p; //number of processors
	int delta; //size between ranges
	int rank; //process rank
	Int index_1 = 0, index_2 = 0, index_merged = 0; //array index variables
	int a_first, a_last, b_first, b_last, c_first, c_last, c_delta; //array range variables
	MPI_Status status; //MPI status
	clock_t start, end;
	double cpu_time_used;
	//allocating arrays
	a = malloc(sizeof(int) * n);
	b = malloc(sizeof(int) * n);
	temp = malloc(sizeof(int) * n * 2);
	merged_arr = malloc(sizeof(int) * n * 2);
	//populating arrays
	srand(time(0));
	int a_val = (rand() % 50) + 1, b_val = (rand() % 50) + 1;
	for (int i = 0; i < n; i++) {
		a[i] = a_val + (rand() % 5) + 1;
		a_val = a[i];
		b[i] = b_val + (rand() % 5) + 1;
		b_val = b[i];
	}
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &p);
	//starting clock
	if (rank == 0){
		start = clock();
	}

	//work division
	work_division(n, p, rank, &a_first, &a_last);
	//setting second array ranges based on first array
	if (rank < p - 1){
		b_last = binary_search(b, n, a[a_last]);
	}
```

```
	else{
		b_last = n - 1;
	}
	if (rank == 0){
		b_first = 0;
	}
	else{
		b_first = binary_search(b, n, a[a_first - 1]) + 1;
	}
	//setting result array range
	c_delta = (a_last - a_first) + (b_last - b_first) + 2;

	if (rank == 0){
		c_first = 0;
	}
	//recieveing low range
	else{
		MPI_Recv(&c_first, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD,
&status);
		c_first += 1;
	}

	c_last = c_first + c_delta - 1;
	//sending high range
	if (rank < p - 1){
		MPI_Send(&c_last, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
	}
	//initialing indexes
	index_1 = a_first;
	index_2 = b_first;
	index_merged = c_first;

	//looping for both arrays
	while(index_1 <= a_last && index_2 <= b_last){
		if (a[index_1] < b[index_2]){
			merged_arr[index_merged] = a[index_1];
			index_1++;
		}
		else{
```

```
                merged_arr[index_merged] = b[index_2];
                index_2++;
            }
            index_merged++;

        }
        //if array 1 is longer
        while (index_1 <= a_last){
            merged_arr[index_merged] = a[index_1];
            index_1++;
            index_merged++;

        }
        //if array 2 is longer
        while (index_2 <= b_last){
            merged_arr[index_merged] = b[index_2];
            index_2++;
            index_merged++;
        }
        //receiving data from all processes
        if (rank == 0){
            for (int i = 1; i < p; i++){
                int temp_c_delta, temp_c_first;
                MPI_Recv(&temp_c_first, 1, MPI_INT, i, 2, MPI_COMM_WORLD,
&status);
                MPI_Recv(&temp_c_delta, 1, MPI_INT, i, 3, MPI_COMM_WORLD,
&status);
                MPI_Recv(merged_arr + temp_c_first, temp_c_delta, MPI_INT, i, 1,
MPI_COMM_WORLD, &status);
            }
        }
        //sending data to process 0
        else{
            MPI_Send(&c_first, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
            MPI_Send(&c_delta, 1, MPI_INT, 0, 3, MPI_COMM_WORLD);
            MPI_Send(merged_arr + c_first, c_delta, MPI_INT, 0, 1,
MPI_COMM_WORLD);
        }
```

```
    //printing final results
    if (rank == 0){
        for(int i = 0; i < 2*n; i++){
            if (i < 3 && i >= 0 || i < n * 2 && i >= n * 2 - 3){
                printf("%d: %d\n", i, merged_arr[i]);
            }
        }
        //ending clock
        end = clock();
        cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
        printf("%f\n", cpu_time_used);
    }
    MPI_Finalize();
    return 0;
}
```