

ASSIGNMENT 2

Group #3



Contents

[**Solution 1**](#_Toc24301843)

[**Source Code 4**](#_Toc24301844)

[**Results 12**](#_Toc24301845)

[**Compilation and running 13**](#_Toc24301846)

# Solution

Our solution is divided into two sections - generating the array files and merging the numbers.

We use ‘random.c’ to generate large arrays of numbers and store them into files - ’sa1.txt’ and ‘sa2.txt’. Both of the files contain a maximum of 32768 numbers which is 2 ^ 15.



A screen shot of ‘sa1.txt’

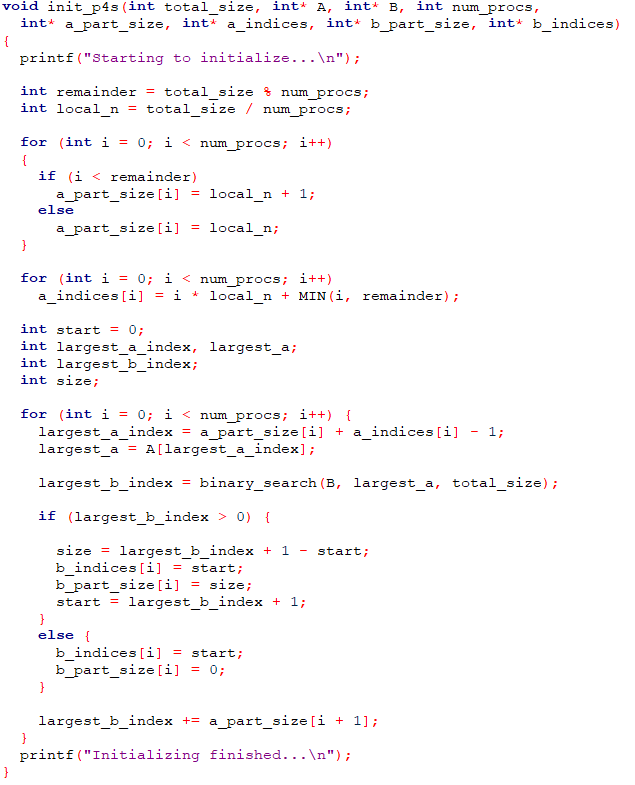
If you want to change the number, please change the following area in both two .c files:



**Our merging algorithm contains five steps.**

**First**, partition array A into n groups, and each will have a\_part\_size[i] elements starting at index a\_indices[i].

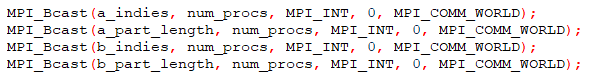
**Second**, find the b\_part\_size and b\_indices.



**The initializing step**

**The first two steps are very important since we are using MPI\_Scatterv() not MPI\_Scatter(). The former can scatter the arrays to different processor with ununiform size while the latter function can’t do this. But we need to calculate all the sizes and indices.**

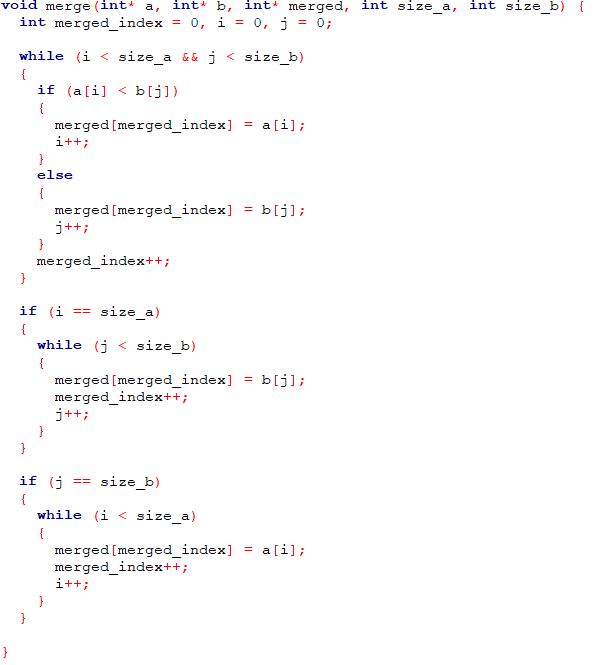
**Third**, processor 0 broadcasts the initialized array partition information to all processors and scatters each divided array to the corresponding processor.



C:\Users\ly\AppData\Roaming\Tencent\Users\823989065\QQ\WinTemp\RichOle\V0$PX`E7GQ~8P}HB][S9B08.png

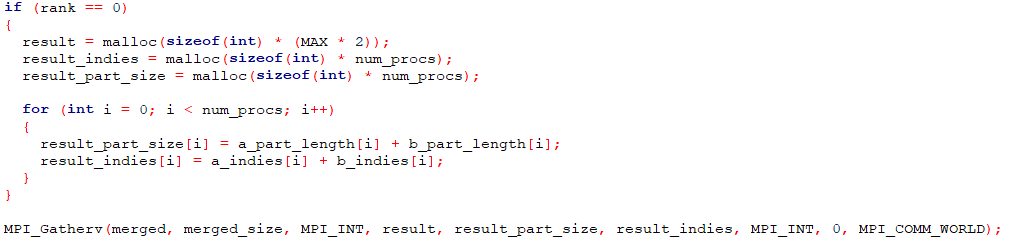
**Data distribution step**

**Forth**, each processor merges assigned array a and array b.



**The merging step of each processor**

**Last**, processor 0 gathers all merged array.



**Result collection step**

# Source Code

**Dear professor, please be careful when copying the code to run on your own computer. It may have some syntax errors because of the copying operation. But it will usually work I think.**

**random.c**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX 32768

void merge(int array[], int left, int m, int right) {

int aux[MAX] = { 0 };

int i;

int j;

int k;

for (i = left, j = m + 1, k = 0; k <= right - left; k++)

{

if (i == m + 1)

{

aux[k] = array[j++];

continue;

}

if (j == right + 1)

{

aux[k] = array[i++];

continue;

}

if (array[i] < array[j])

{

aux[k] = array[i++];

}

else

{

aux[k] = array[j++];

}

}

for (i = left, j = 0; i <= right; i++, j++)

{

array[i] = aux[j];

}

}

void mergesort(int a[], int low, int high) {

int mid;

if (low < high) {

mid = (high + low) / 2;

mergesort(a, low, mid);

mergesort(a, mid + 1, high);

merge(a, low, mid, high);

}

}

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

int n = MAX;

int original\_array1[MAX];

int original\_array2[MAX];

int c;

srand(time(NULL));

for (c = 0; c < n; c++) {

original\_array1[c] = rand() % n;

original\_array2[c] = rand() % n;

}

mergesort(original\_array1, 0, n - 1);

mergesort(original\_array2, 0, n - 1);

FILE\* out1 = fopen("sa1.txt", "w");

FILE\* out2 = fopen("sa2.txt", "w");

int i = 0;

for (i = 0; i < n; i++) {

fprintf(out1, "%d ", original\_array1[i]);

fprintf(out2, "%d ", original\_array2[i]);

}

fclose(out1);

fclose(out2);

printf("Generation Done!\n");

}

**merging\_sorted\_arrays.c**

#include "mpi.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include <math.h>

#define MAX 32768

#define MIN(a,b) (a < b ? a : b)

// function used to read arrays from files

void file\_to\_array(int\* init\_array, int file)

{

FILE\* input;

if (file == 0)

input = fopen("sa1.txt", "r");

else

input = fopen("sa2.txt", "r");

printf("Starting to read file...\n");

int num;

int count = 0;

while (fscanf(input, "%d", &num) != EOF)

{

init\_array[count] = num;

count++;

}

fclose(input);

printf("File reading finished...\n");

}

// function used to write the arrays to files

void array\_to\_file(int\* array)

{

FILE\* out = fopen("result.txt", "w");

int i = 0;

for (i = 0; i < 2 \* MAX; i++) {

fprintf(out, "%d ", array[i]);

}

fclose(out);

}

// binary search method to find a target in one array

int binary\_search(int\* array, int target, int size)

{

int left = 0;

int right = size - 1;

int mid = floor((left + right) / 2);

int found\_index = -1;

while (left <= right)

{

if (array[mid] < target)

{

left = mid + 1;

}

else if (array[mid] == target)

{

left = mid + 1;

found\_index = mid;

}

else

{

right = mid - 1;

}

mid = floor((left + right) / 2);

}

return mid;

}

// function to init all the parameters for MPI\_scatterv()

void init\_p4s(int total\_size, int\* A, int\* B, int num\_procs,

int\* a\_part\_size, int\* a\_indices, int\* b\_part\_size, int\* b\_indices)

{

printf("Starting to initialize...\n");

int remainder = total\_size % num\_procs;

int local\_n = total\_size / num\_procs;

for (int i = 0; i < num\_procs; i++)

{

if (i < remainder)

a\_part\_size[i] = local\_n + 1;

else

a\_part\_size[i] = local\_n;

}

for (int i = 0; i < num\_procs; i++)

a\_indices[i] = i \* local\_n + MIN(i, remainder);

int start = 0;

int largest\_a\_index, largest\_a;

int largest\_b\_index;

int size;

for (int i = 0; i < num\_procs; i++) {

largest\_a\_index = a\_part\_size[i] + a\_indices[i] - 1;

largest\_a = A[largest\_a\_index];

largest\_b\_index = binary\_search(B, largest\_a, total\_size);

if (largest\_b\_index > 0) {

size = largest\_b\_index + 1 - start;

b\_indices[i] = start;

b\_part\_size[i] = size;

start = largest\_b\_index + 1;

}

else {

b\_indices[i] = start;

b\_part\_size[i] = 0;

}

largest\_b\_index += a\_part\_size[i + 1];

}

printf("Initializing finished...\n");

}

// function used to merge two sorted arrays

void merge(int\* a, int\* b, int\* merged, int size\_a, int size\_b) {

int merged\_index = 0, i = 0, j = 0;

while (i < size\_a && j < size\_b)

{

if (a[i] < b[j])

{

merged[merged\_index] = a[i];

i++;

}

else

{

merged[merged\_index] = b[j];

j++;

}

merged\_index++;

}

if (i == size\_a)

{

while (j < size\_b)

{

merged[merged\_index] = b[j];

merged\_index++;

j++;

}

}

if (j == size\_b)

{

while (i < size\_a)

{

merged[merged\_index] = a[i];

merged\_index++;

i++;

}

}

}

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

int num\_procs;

int rank;

// MPI init

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &num\_procs);

int\* A;

int\* B;

// Define the arrays for using MPI\_Scatterv().

// We are not using MPI\_Scatter(), so this may be more complex.

int\* a\_indies = malloc(sizeof(int) \* num\_procs);

int\* a\_part\_length = malloc(sizeof(int) \* num\_procs);

int\* b\_indies = malloc(sizeof(int) \* num\_procs);

int\* b\_part\_length = malloc(sizeof(int) \* num\_procs);

// processor 0: read the arrays and init the parameters

if (rank == 0)

{

A = malloc(sizeof(int) \* MAX);

B = malloc(sizeof(int) \* MAX);

file\_to\_array(A, 0);

file\_to\_array(B, 1);

init\_p4s(MAX, A, B, num\_procs, a\_part\_length, a\_indies, b\_part\_length, b\_indies);

}

// broadcast all the parameters

MPI\_Bcast(a\_indies, num\_procs, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(a\_part\_length, num\_procs, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(b\_indies, num\_procs, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(b\_part\_length, num\_procs, MPI\_INT, 0, MPI\_COMM\_WORLD);

int a\_size = a\_part\_length[rank];

int b\_size = b\_part\_length[rank];

int merged\_size = a\_size + b\_size;

int\* a = malloc(sizeof(int) \* a\_size);

int\* b = malloc(sizeof(int) \* b\_size);

int\* merged = malloc(sizeof(int) \* merged\_size);

// scatter the divided sub-arrays to processors

MPI\_Scatterv(A, a\_part\_length, a\_indies, MPI\_INT, a, a\_size, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Scatterv(B, b\_part\_length, b\_indies, MPI\_INT, b, b\_size, MPI\_INT, 0, MPI\_COMM\_WORLD);

// merge the sub-arrays

merge(a, b, merged, a\_size, b\_size);

int\* result;

int\* result\_indies;

int\* result\_part\_size;

// processor 0 gather all the merged sub-arrays

if (rank == 0)

{

result = malloc(sizeof(int) \* (MAX \* 2));

result\_indies = malloc(sizeof(int) \* num\_procs);

result\_part\_size = malloc(sizeof(int) \* num\_procs);

for (int i = 0; i < num\_procs; i++)

{

result\_part\_size[i] = a\_part\_length[i] + b\_part\_length[i];

result\_indies[i] = a\_indies[i] + b\_indies[i];

}

}

MPI\_Gatherv(merged, merged\_size, MPI\_INT, result, result\_part\_size, result\_indies, MPI\_INT, 0, MPI\_COMM\_WORLD);

// processor 0: write the final array to file "result.txt"

if (rank == 0)

{

/\*

for (int i = 32760; i < 32768; ++i)

printf("%d, ", A[i]);

printf("\n");

for (int i = 32760; i < 32768; ++i)

printf("%d, ", B[i]);

printf("\n");

for (int i = 65522; i < 65536; ++i)

printf("%d, ", result[i]);

printf("\n");

\*/

array\_to\_file(result);

// free all the memories for processor 0

free(A); free(B); free(result);

free(result\_indies); free(result\_part\_size);

}

// free all the memories

free(a\_indies);

free(a\_part\_length);

free(b\_indies);

free(b\_part\_length);

free(a);

free(b);

free(merged);

printf("Processor %d done!\n", rank);

MPI\_Finalize();

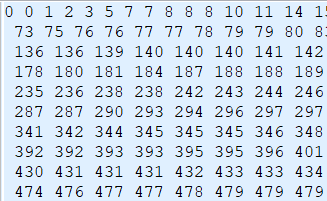
return 0;

}

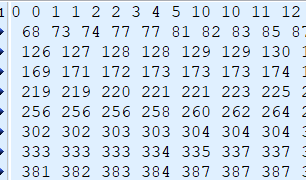
# Results

EX.

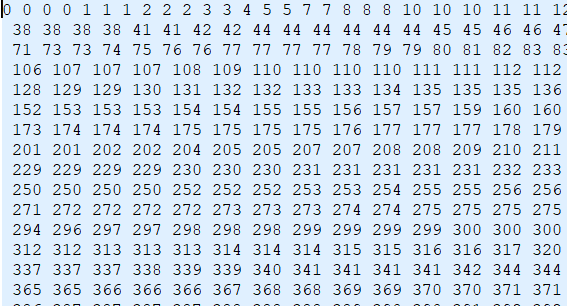
The first set



The second set



The result:



We can determine the correctness of the program based on the first (or last) few numbers or using a simple check function.

# Compilation and running

1. Compile and run the “random.c” (using gcc) and then we will get two array files named “sa1.txt” and “sat2.txt”.
2. Compile and run the “merging\_sorted\_arrays.c” (using mpicc and mpirun) and then we will get the file named “result.txt” which stores the merged array from the 2 sorted ones.