



ASSIGNMENT 2

Group #3



Contents

Solution	1
Source Code.....	4
Results	12
Compilation and running.....	13

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} .

```
0 0 1 2 3 5 7 7 8 8 8 10 11 14 15 15 16 16 16 17 19 19 19 22 24 27 27 28 30 32 33 33 34 34
36 36 38 41 42 44 45 48 48 49 50 51 52 52 53 53 59 59 60 61 61 62 62 62 63 64 65 65 66 69
69 71 73 75 76 76 77 77 78 79 79 80 83 83 84 86 88 90 91 92 93 93 95 98 98 99 102 103 103
103 104 106 107 107 108 109 110 112 113 114 115 116 116 117 117 120 120 122 123 127 128
132 134 135 136 136 139 140 140 140 141 142 143 144 145 145 146 147 148 148 150 152 152
153 155 155 157 157 159 160 161 161 162 164 164 165 165 166 167 168 168 169 171 172 173
175 175 176 177 177 178 180 181 184 187 188 188 189 195 196 196 198 199 199 199 200 202
204 205 205 207 208 209 214 214 214 214 217 218 221 222 222 222 223 224 225 227 229 229
230 231 231 231 233 233 234 235 236 238 238 242 243 244 246 248 249 250 250 256 256 257
258 258 260 260 261 262 263 264 264 266 267 267 268 268 269 270 271 272 272 272 273 273
275 275 278 279 281 281 282 282 286 287 287 290 293 294 296 297 297 298 298 299 299 299
299 300 300 302 304 305 305 306 310 310 311 313 313 314 315 315 320 320 322 324 325 327
328 332 333 333 334 335 337 338 340 341 341 341 342 344 345 345 345 346 348 350 350 351
354 355 356 356 358 359 361 362 365 366 366 368 368 369 370 371 372 372 374 374 376 376
378 379 379 379 380 380 385 385 386 386 388 388 391 392 392 393 393 395 395 396 401 403
405 406 407 408 409 409 410 411 413 413 415 416 416 417 417 417 418 418 418 420 421 421
423 423 423 425 425 425 426 426 427 428 428 429 429 430 430 430 431 431 431 432 433 433
434 434 437 437 442 442 442 443 444 445 446 446 447 448 449 450 450 451 451 452 453 455
456 456 456 456 459 462 462 463 463 464 464 464 466 468 474 474 474 476 477 477 478
479 479 479 480 482 485 487 488 488 489 489 489 491 491 491 493 494 495 497 498 499 499
499 502 504 504 504 504 509 510 512 514 516 517 519 520 520 525 526 527 527 527 528 529
529 530 533 533 534 534 534 535 535 535 536 538 540 544 547 547 548 549 549 549 550 553
554 554 555 556 557 557 558 559 559 561 561 562 562 565 566 566 566 567 568 568 569 569
570 572 572 574 574 575 575 577 578 578 579 580 580 581 582 582 582 583 583 584 584 586
587 590 591 592 595 595 595 596 597 598 598 600 602 603 604 605 606 610 611 611 612 613
613 614 615 617 617 618 621 621 623 624 624 625 626 626 627 629 629 631 631 633 636 637
638 640 640 641 642 645 647 648 648 650 650 652 652 653 653 654 654 654 655 657 657 658
658 658 661 667 668 669 670 671 673 673 673 674 674 675 676 677 677 678 678 679 680 681
681 681 682 682 684 684 684 687 687 688 688 689 691 691 693 693 694 695 696 696 697 698
699 699 701 704 705 705 705 706 707 707 709 711 711 711 713 713 715 715 715 716 717 719
723 726 726 729 730 731 732 734 734 735 736 736 737 738 738 739 740 742 743 744 744 747
748 749 751 752 752 753 754 755 755 755 756 759 759 760 760 764 765 766 766 769 769 773
775 781 781 786 786 786 789 790 791 791 791 793 794 794 796 796 797 798 798 800 801 801
802 802 802 806 809 809 810 811 811 812 813 813 813 813 815 816 816 819 820 820 822
823 824 825 826 826 827 828 829 829 830 831 833 833 834 834 835 836 836 838 839 840
841 841 842 842 843 843 846 847 848 849 849 851 852 853 854 857 857 860 860 862 863 864
864 864 867 868 868 871 873 873 875 875 876 878 881 883 883 884 886 887 887 887 888 888
888 891 891 893 895 896 898 899 901 904 905 907 909 909 910 911 914 915 917 918 918 918
918 921 921 923 924 925 928 931 931 931 933 933 934 934 934 935 935 936 936 937 937
937 939 939 940 942 944 945 947 947 947 949 949 950 955 956 957 957 959 960 961 962 962
962 963 965 966 966 966 967 967 968 969 972 972 972 973 973 974 974 974 977 979 979
979 981 982 983 986 988 989 989 990 991 992 993 994 994 994 995 996 999 999 999 999
1000 1000 1003 1009 1010 1012 1012 1013 1015 1015 1015 1016 1016 1016 1018 1022 1023 1023
```

A screen shot of 'sa1.txt'

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

```
#define MAX 32768
```

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`.

```
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");
}
```

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.

```
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);
```

```
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);
```

Data distribution step

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

```
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++;
        }
    }
}
```

The merging step of each processor

Last, processor 0 gathers all merged array.

```
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);
```

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 found_index;
}

// 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

0	0	1	2	3	5	7	7	8	8	8	10	11	14	15
73	75	76	76	77	77	77	78	79	79	80	80	81	82	83
136	136	139	140	140	140	140	141	142	142	143	143	144	145	146
178	180	181	184	187	188	188	189	190	191	192	193	194	195	196
235	236	238	238	242	243	244	246	247	248	249	250	251	252	253
287	287	290	293	294	296	297	297	298	299	300	301	302	303	304
341	342	344	345	345	345	346	348	349	350	351	352	353	354	355
392	392	393	393	395	395	396	401	402	403	404	405	406	407	408
430	431	431	431	432	433	433	434	435	436	437	438	439	440	441
474	476	477	477	478	479	479	479	480	481	482	483	484	485	486

The second set

0	0	1	1	2	2	3	4	5	10	10	11	12	13	14
68	73	74	77	77	81	82	83	85	87	88	89	90	91	92
126	127	128	128	129	129	130	131	132	133	134	135	136	137	138
169	171	172	173	173	173	174	175	176	177	178	179	180	181	182
219	219	220	221	221	221	223	225	226	227	228	229	230	231	232
256	256	256	258	260	262	264	265	266	267	268	269	270	271	272
302	302	303	303	304	304	304	305	306	307	308	309	310	311	312
333	333	333	334	335	337	337	338	339	340	341	342	343	344	345
381	382	383	384	387	387	387	388	389	390	391	392	393	394	395

The result:

```
0 0 0 0 1 1 1 2 2 2 3 3 4 5 5 7 7 8 8 8 10 10 10 11 11 12
38 38 38 38 41 41 42 42 44 44 44 44 44 44 45 45 46 46 47
71 73 73 74 75 76 76 77 77 77 77 78 79 79 80 81 82 83 83
106 107 107 107 108 109 110 110 110 110 111 111 112 112
128 129 129 130 131 132 132 133 133 134 135 135 135 136
152 153 153 153 154 154 155 155 156 157 157 159 160 160
173 174 174 174 175 175 175 175 176 177 177 177 178 179
201 201 202 202 204 205 205 207 207 208 208 209 210 211
229 229 229 229 230 230 230 231 231 231 231 231 232 233
250 250 250 250 252 252 252 253 253 254 255 255 256 256
271 272 272 272 272 273 273 273 274 274 275 275 275 275
294 296 297 297 298 298 298 299 299 299 299 300 300 300
312 312 313 313 313 314 314 314 315 315 316 316 317 320
337 337 337 338 339 339 340 341 341 341 341 342 344 344
365 365 366 366 366 367 368 368 369 369 370 370 371 371
```

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 “sal.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.