Homework 3 – SF2568 Parallel Computations for Large-Scale Problems

Cuong Duc Dao, Donggyun Park
`cuongdd@kth.se, donggyun@kth.se`

March 19, 2021

Question 1:

(a) Propose an algorithm based on Rank Sort with a running time of $O(\log N)$!

(b) How many processors are needed? What is the processor efficiency?

(c) Would you use this algorithm in practice? Why or why not?

Solution:

(a)
We have a list A with N numbers and a list B for the rank. Start with N distributed numbers in N of processors, so one processor stores one number. And each processor is supposed to make (N-1) times of comparison but we will use N(N-1) processors so that each processor will have made only one comparison at the same time in parallel.
Regarding comparison for each element with A[i] ($0 \leq$ i $\leq$ N-1), all processors have 0 or 1 after one comparison of A[i] with the number stored in the beginning and N reductions need to be done on (N-1) processors in parallel. These reductions take $\log N$ time complexity. So now we have counted all numbers less than A[i]. Finally the element with count k is placed in the rank list B[k].
In conclusion, there are no communication delays as the shared memory is used thus the time complexity is $\log N$ from the reductions (computing the rank count k)

(b)
N(N-1) processors are needed.
The processor efficiency E can be expressed as below;

$$E = \frac{\text{speedup}}{\text{the number of processors}} = \frac{N^2}{P \log(N)} \tag{1}$$

and in this problem we have $P \approx N^2$, i.e., $N(N-1)$ more precisely, so E becomes $\frac{1}{\log(N)}$, which is not a significant improvement.

(c)
No, it may be useful only in terms of the time complexity but it would require N(N-1) processors which could be ridiculous in large-scale problems in reality.

Question 2: Determine what is wrong with the Odd-Even Transposition program and correct it.

Solution:
The problem with the given code is deadlock. Both of the cases in 'if' conditions send the data first without receiving any. This will result in deadlock as soon as the program is implemented. We propose to explicitly divide the problem into 4 cases with if statements as in the lecture slides. Them we can properly implement the Odd-Even Transposition Sort without mistakes.(assume data is already distributed to each processor.)

(pseudo codes, processor index starts from 0)

```
for step from 0 to P − 1
    if (even processor & even phase)
        receive(x, p + 1)
        send(a, p + 1)
        if x < a then a = x
    else if (even processor & odd phase)
        if (p ≥ 1)
            receive(x, p − 1)
            send(a, p − 1)
            if x > a then a = x
    else if (odd processor & even phase)
        send(x, p − 1)
        receive(a, p − 1)
            if a < x then a = x
    else if (odd processor & odd phase)
        if (p ≤ N − 2)
            send(x, p + 1)
            receive(a, p + 1)
            if a > x then a = x
```

Question 3: Write an MPI program which implements the Odd-Even Transposition Sort

(a) Implement the program and show its correctness for a small N and a small number of processors.

(b) Measure the efficiency of your program using a large N (say $N = 10^7$) on $P = 1, 2, 3, 4, 8$ processors. Provide a plot of the speedup. Draw conclusions!

Solution:

We want to sort an array of N elements in parallel using P processors. Assuming that the array is load-balanced linearly distributed across the processors After sorting, each processor p will hold I_p elements such that

- All elements in processor 0 are sorted, and are smaller than the smallest element in processor 1
- All elements in processor 1 are sorted, and are smaller than the smallest element in processor 2
- so on and so forth...

In this problem's setting, each processor generates a random array of I_p elements and sort them locally. We use the built-in quick-sort **`qsort`** in C for this task. Subsequently, the processors exchange sorted data through odd-even transposition algorithm to make sure that the data is globally sorted and follow the description above.

(a) We implement the odd-even transposition shown in the lecture. The code of our implementation is shown in Listing 1 below, as well as in the attached file to the submission. We handle the case when P doesn't divide N as following:

- Each processor will have $I_p = N/P + 1$ elements
- For all processors p whose rank is $r_p \geq N\%P$, we change the last element in p to -1.0.
- Perform local sort and global odd-even transposition
- When accumulating the final result, we discard those elements whose values are -1.0.

This works for this problem because the array elements are randomly generated between 0 and 1, thus, -1.0 could be thought of an placeholder.
The result for $N = 10$ and $P = 3$ is shown in Figure 1. As can be seen from the figure, our program can sort the array properly even when P does not divide N. We also write a small Python script (Listing 2) to verify that the array is properly sorted.

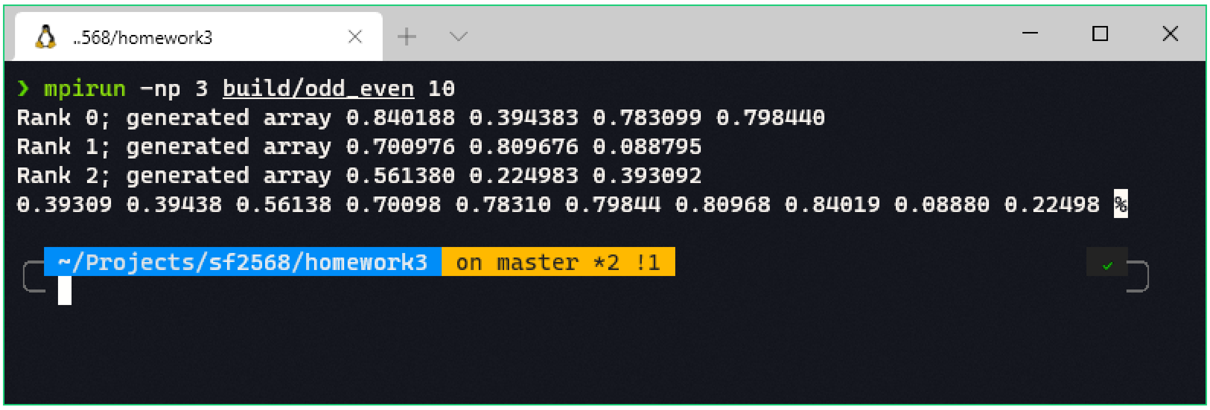

Figure 1: Result of our implementation with $N = 10$ and $P = 3$

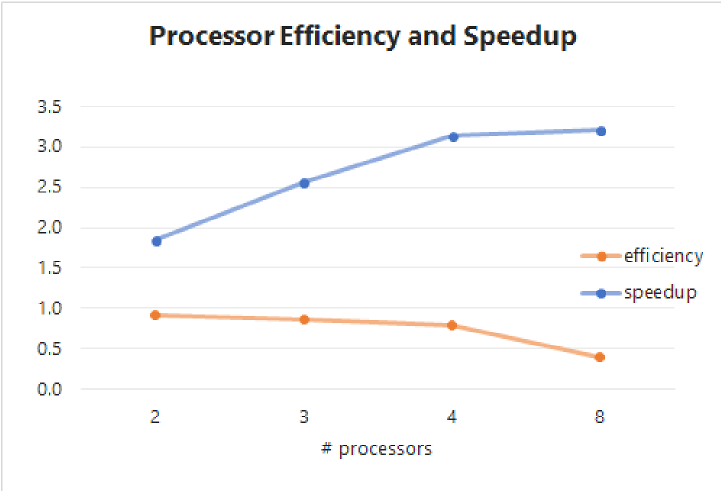

Figure 2: Parallel efficiency and speedup w.r.t number of processors P

(b) We plot the efficiency and speedup w.r.t to number of processors of our implementation in Figure 2.
Conclusion: the time spent on the communications between the processors are relatively longer than the time spent on the arithmetic operations, so sometimes total parallel computations may not be as fast as we expect. Therefore, carefully consider the number of processors(to be economical) and the number of communications among the processors!

Listing 1: Implementation of Odd-Even Sort

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

#define EMPTY_PLACEHOLDER -1.0

void print_array(double *arr, const int len);
int compute_neighbor(int phase, int rank, int size);
int compare(const void *, const void *);
void merge_arrays(double *, int, double *, int, double *, unsigned int);

int main(int argc, char *argv[]) {
    int size, rank;
    double time_spent = 0.0;
    double time_spent2 = 0.0;

    if (argc < 2) {
        printf("N needs to be given\n");
        exit(1);
    }

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    MPI_Status status;

    // Local variables
    int N = atoi(argv[1]);
    // Data is load-balanced linearly distributed into processes
    // int I = (N + size - rank - 1) / size;
    int I = N / size;
    if (N % size != 0)
        I += 1;

    // Data generation
    srandom(rank + 1);
    double *x = malloc(sizeof(double) * I);
    double *a = malloc(sizeof(double) * I);
    double *temp = malloc(sizeof(double) * I *2);

    for (int i = 0; i < I; i++) {
        x[i] = ((double) random()) / RAND_MAX;
    }

    if ((N % size != 0) && (rank >= N % size)) {
        x[I-1] = EMPTY_PLACEHOLDER;
    }
    clock_t begin = clock();
    // Local sort
    qsort(x, I, sizeof(double), compare);
    clock_t end=clock();
    // Odd-even transposition
    clock_t begin2 = clock();
    for (int phase = 0; phase < size; phase++) {
        MPI_Barrier(MPI_COMM_WORLD);
        int neighbor = compute_neighbor(phase, rank, size);

        if (neighbor >= 0 && neighbor < size) {
            MPI_Sendrecv(x, I, MPI_DOUBLE, neighbor, phase,
                         a, I, MPI_DOUBLE, neighbor, phase,
                         MPI_COMM_WORLD, &status);

            if (rank < neighbor) {
                merge_arrays(x, I, a, I, temp, 1);
            } else {
                merge_arrays(x, I, a, I, temp, 0);
            }
        }
```

```c
    }
    clock_t end2=clock();
    time_spent += (double)(end-begin)/CLOCKS_PER_SEC;
    time_spent2 += (double)(end2-begin2)/CLOCKS_PER_SEC;
    printf("time spent for qsort is %f seconds\ntime spent for merging is %f seconds\n", time_spent,time_spent

    // Sequentially write the sorted array
    int signal = 0;
    FILE *f;
    char fname[50];
    sprintf(fname, "sorted_array_s-%d_N-%s.txt", size, argv[1]);

    if (rank == 0) { // master process
            f = fopen(fname, "w");
        for (int i = 0; i < I; i++) {
            if (x[i] != EMPTY_PLACEHOLDER) {
                    fprintf(f, "%1.10f\n", x[i]);
            }
        }
        fclose(f);
        signal = 1;
        MPI_Send(&signal, 1, MPI_INT, rank+1, 100, MPI_COMM_WORLD);
    } else {
        MPI_Recv(&signal, 1, MPI_INT, rank-1, 100, MPI_COMM_WORLD, &status);
        if (signal == 1) {
            f = fopen(fname, "a");
            for (int i = 0; i < I; i++) {
                if (x[i] != EMPTY_PLACEHOLDER) {
                           fprintf(f, "%1.10f\n", x[i]);
                }
            }
            fclose(f);
            if (rank != size - 1) {
                MPI_Send(&signal, 1, MPI_INT, rank+1, 100, MPI_COMM_WORLD);
            }
        }
    }

    free(x);

    MPI_Finalize();
    return 0;
}

void print_array(double *arr, const int len) {
    for (int i = 0; i < len; i++) {
        printf("%f ", arr[i]);
    }
    printf("\n");
}

int compute_neighbor(int phase, int rank, int size) {
    int neighbor;

    if (phase % 2 != 0) {  // Odd phase
        if (rank % 2 != 0) { // Odd processor
            neighbor = rank + 1;
        } else { // Even processor
            neighbor = rank - 1;
        }
    } else { // Even phase
        if (rank % 2 != 0) { // Odd processor
            neighbor = rank - 1;
        } else { // Even processor
            neighbor = rank + 1;
        }
    }

    if (neighbor < 0 || neighbor >= size) {
        neighbor = -1;
    }
    return neighbor;
}
```

```c

int compare(const void *x, const void *y) {
    double xx = *(double*)x, yy = *(double*)y;
    if (xx < yy) return -1;
    if (xx > yy) return 1;
    return 0;
}

/**
 * Merge two **sorted** array `src` and `rec` into the `temp` array
 * `keep_low` decides if we want to keep the lower part in the `src`
 * or the upper part in the `src` array after this merge operation
 */
void merge_arrays(
    double *src, int len_src, double *recv, int len_recv,
    double *temp, unsigned int keep_low
) {
    int i = 0, j = 0, k = 0;

    while (i < len_src && j < len_recv) {
        if (src[i] < recv[j])
            temp[k++] = src[i++];
        else
            temp[k++] = recv[j++];
    }

    // Store remaining elements of first array
    while (i < len_src)
        temp[k++] = src[i++];

    // Store remaining elements of second array
    while (j < len_recv)
        temp[k++] = recv[j++];

    if (keep_low == 1) {
        // Keep the lower part of the sorted `temp` array in
        // `src` for this process
        for (int i = 0; i < len_src; i++) {
            src[i] = temp[i];
        }
    } else if (keep_low == 0) {
        // Keep the upper part of the sorted `temp` array in
        // `src` for this process
            for (int i = len_src, j = 0; j < len_recv; i++, j++) {
            src[j] = temp[i];
        }
    }
}
```

Listing 2: Python program to validate that the resulted array is properly sorted

```python
"""
Validate if an array is sorted.
The array is read from a text file
"""
import sys
import numpy as np


def main():
    arr = []

    with open(sys.argv[1], "r") as f:
        for line in f:
            arr.append(float(line.strip()))

    print("Array read! %d elements" % len(arr))

    ori_arr = np.array(arr)
    sorted_arr = np.sort(ori_arr)

    if (ori_arr == sorted_arr).all():
```

```python
        print("Array is properly sorted!")
    else:
        print("Array might have not been sorted properly!")


if __name__ == '__main__':
    main()
```