МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Федеральное государственное образовательное учреждение
высшего образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт математики, механики и компьютерных наук им. И. И. Воровича

Отчёт

«Распараллеливание сортировки»

Выполнил: студент III курса, Радько В. А.
Проверил: доцент каф. ВМиМФ, Говорухин В. Н.

Ростов-на-Дону, 2018

Содержание

1 Распараллеливание сортировки с помощью OpenMP

В качестве базового способа сортировки была выбрана простейшая сортировка «пузырьком». Метод распараллеливания заключается в том, чтобы разбить массив на приблизительно равные участки, отсортировать каждый в отдельности, а затем слить эти участки в один отсортированный массив. Массив разбивается на k участков $[a_i, a_{i+1}]$, где $a_i = a_0 + i \cdot \frac{n}{k}$, а длина такого участка будет равна $(i+1) \cdot \frac{n}{k} - i \cdot \frac{n}{k}$.

Все следующие тесты проводились 10 раз, и бралось среднее значение, чтобы приблизить результаты эксперимента к математическому ожиданию.

1.1 Тест $p = 1$, k – переменная

Тест при фиксированном одном потоке и переменном количестве разбиений показывает, что алгоритм сортировки «пузырьком» ускоряется даже на одном потоке. Результаты такого теста на рис. 1

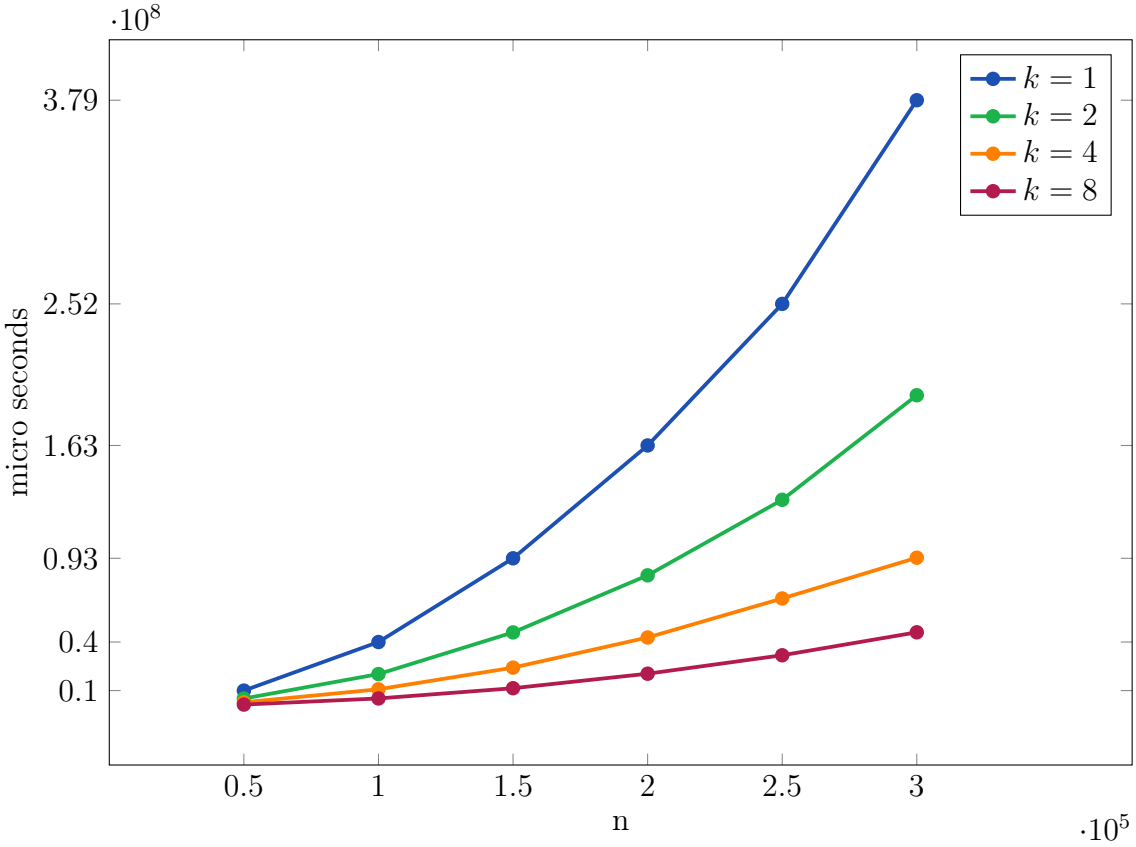

Рис. 1: Ускорение при увеличении параметра k, один поток

1.2 Тест при увеличении обоих параметров

Результаты на рисунках 2–5 показывают, что при $k < p$ результатов от параллельности нет, поскольку алгоритм параллелится по участкам, которых k штук.

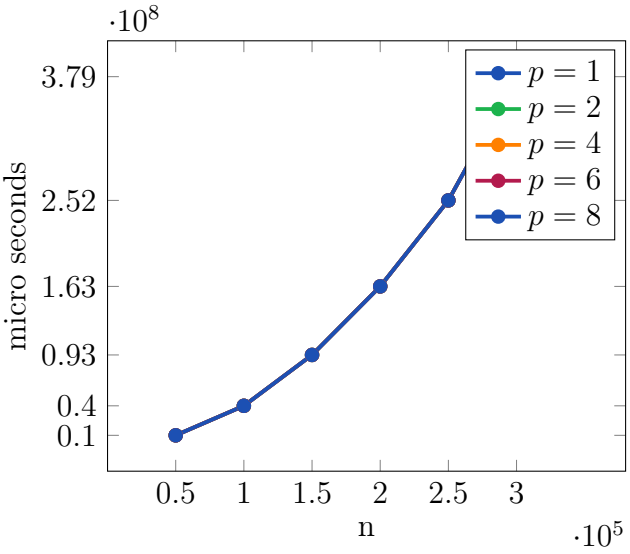

Рис. 2: Ускорение при фиксации параметра $k = 1$

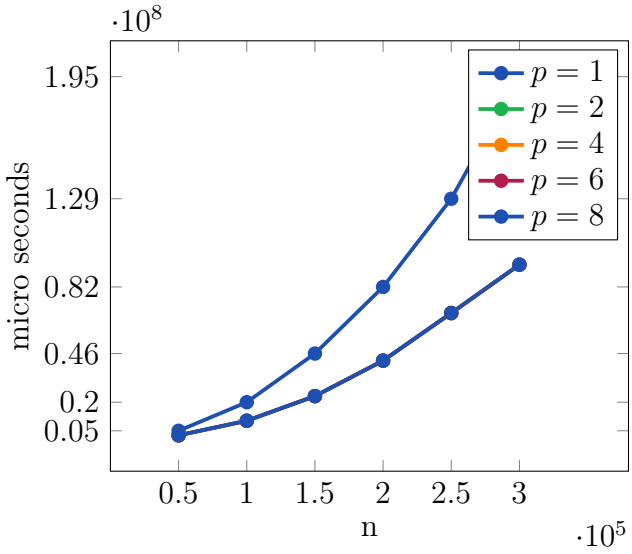

Рис. 3: Ускорение при фиксации параметра $k = 2$

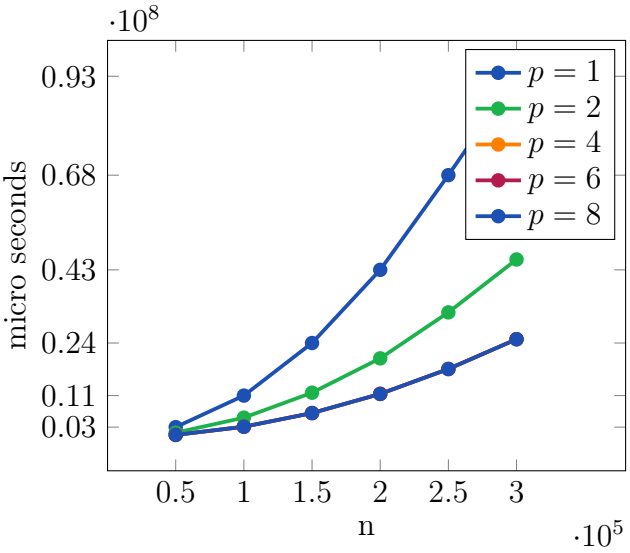

Рис. 4: Ускорение при фиксации параметра $k = 6$

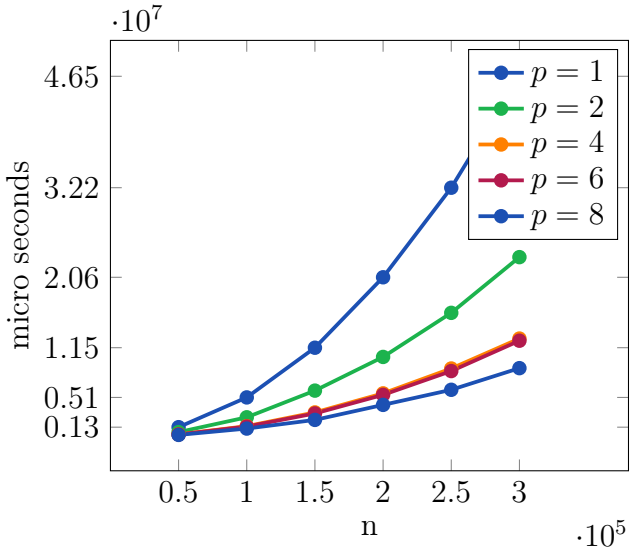

Рис. 5: Ускорение при фиксации параметра $k = 8$

1.3 Результаты эксперимента при переменном количестве потоков

Зафиксируем количество разбиений $k = 100$. Результаты эксперимента на рис. 6 показывают несколько вещей:

- при увеличении потоков в m раз, ускорение всегда происходит в $t < m$ раз;
- несмотря на то, что количество потоков может быть больше, чем количество физических ядер, реального ускорения программы при количестве потоков p большем, чем количество физических ядер – не происходит.

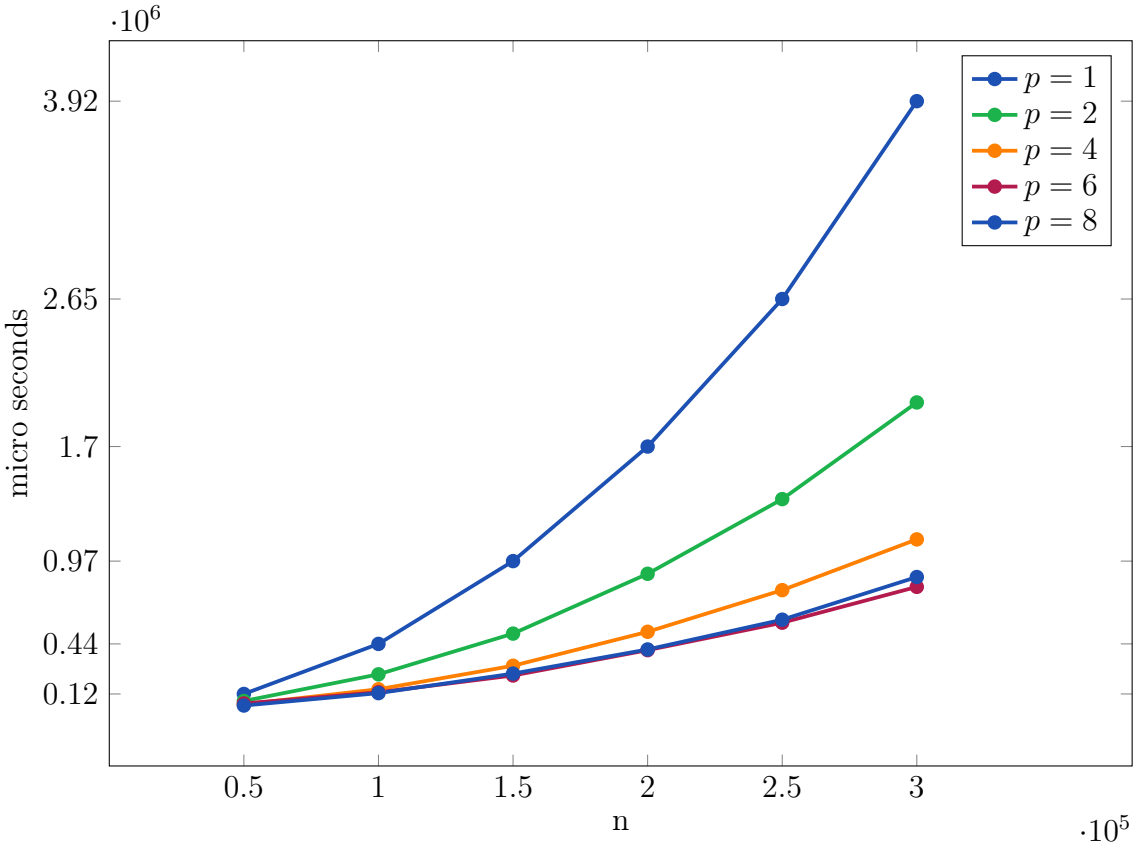

Рис. 6: Ускорение при значении параметра $k = 100$ и разном количестве потоков

2 Распараллеливание сортировки с помощью MPI

Алгоритм распараллеливания с помощью MPI отличается от того же алгоритма на OpenMP. В алгоритме с MPI я фиксирую количество разбиений массива количеством используемых узлов. Каждый узел получает свою часть массива, которую сортирует и тогда нулевой узел собирает массив обратно и собирает отсортированный массив из полученных кусков. Представить такой алгоритм можно таким образом:

Результаты распараллеливания представлены на рисунке ниже:

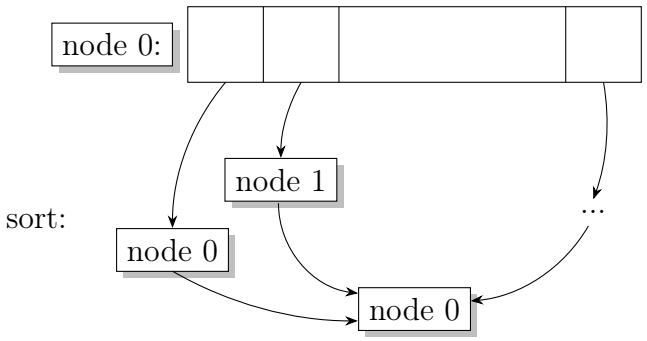

Рис. 7: Алгоритм сортировки

Таблица 1: Таблица средних ускорений

Количество узлов	Среднее ускорение
1	1
2	4
4	$11 - 16$
6	$38 - 40$
8	$66 - 69$
12	$122 - 134$

Листинг 1: Исходный код программы для OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <string.h>
#include <omp.h>

int nthrds = 0;

void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int* sort(int* a, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = i+1; j < n; ++j) {
```

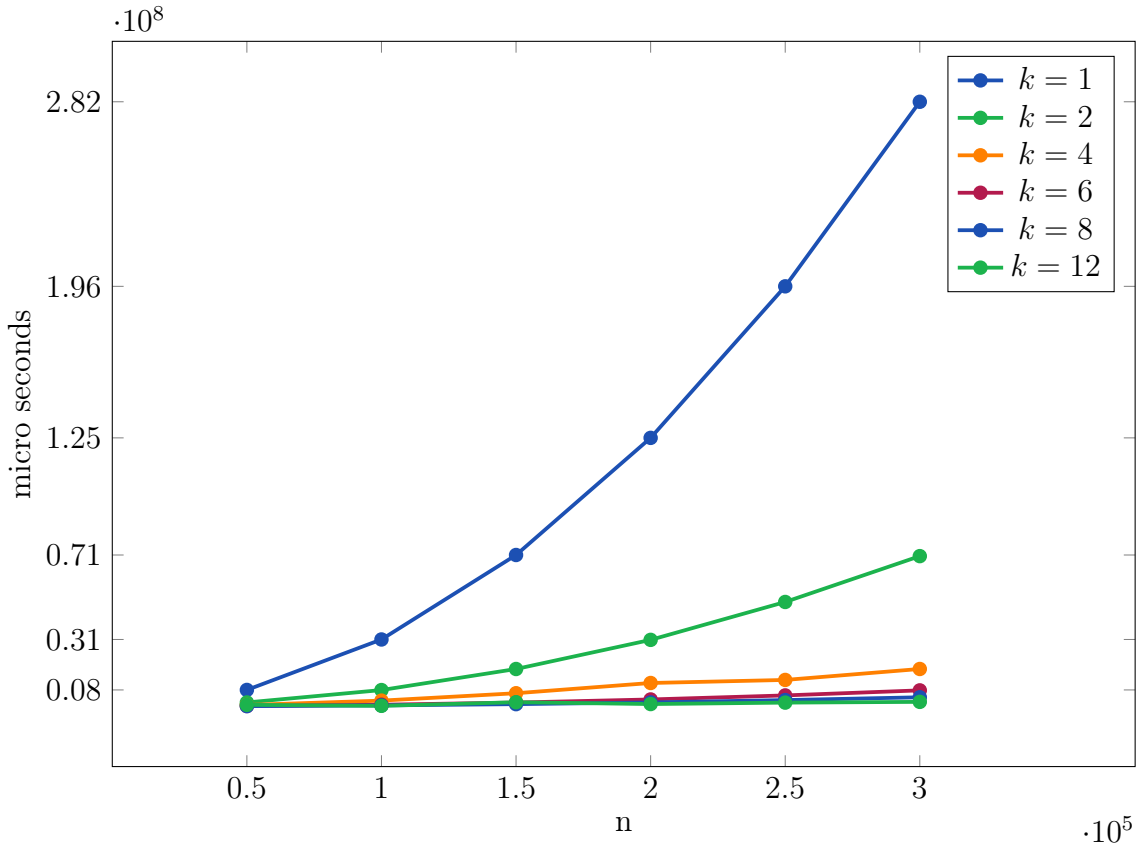

Рис. 8: Время при распараллеливании MPI

```
            if (a[i] > a[j]) swap(a+i, a+j);
        }
    }

    return a;
}

int* fill(int* a, int n) {
    for (int i = 0; i < n; ++i) {
        a[i] = rand();
    }

    return a;
}

int* print(int* a, int n) {
    printf("[%d", a[0]);
    for (int i = 1; i < n; ++i) {
```

```
        printf(", %d", a[i]);
    }
    printf("]");

    return a;
}

long long get_time() {
    struct timeval tv;
    gettimeofday(&tv, NULL);

    return tv.tv_usec + tv.tv_sec*1000000LL;
 }

long seconds(long long t) {
    return (t / 1000000);
}

long useconds(long long t) {
    return (t % 1000000);
}

int* partial_print(int* a, int n, int k) {
    printf("[");
    print(a, n/k);
    for (int i = 1; i < k; ++i) {
        printf(", ");
        print(a + i*n/k, (i+1)*n/k - i*n/k);
        // the length of subarray is ((a + (i+1)*n/k) - (a + i*n/k))
    }
    printf("]");

    return a;
}

int* merge_parts(int* a, int n, int* b, int k, int* c) {
    int* a_end = a + n;
    int* b_end = b + k;
```

```
    while ((a < a_end) && (b < b_end)) {
        if (*a < *b) {
            *c++ = *a++;
        } else {
            *c++ = *b++;
        }
    }

    if (!(a < a_end) && b < b_end) {
        while (b < b_end) {
            *c++ = *b++;
        }
    }
    if (!(b < b_end) && a < a_end) {
        while (a < a_end) {
            *c++ = *a++;
        }
    }

    return c;
}

int* merge(int* a, int n, int k) {
    int* b = calloc(n, sizeof(int));
    for (int i = 0; i < k; ++i) {
        merge_parts(a, i*n/k, a + i*n/k, (i+1)*n/k - i*n/k, b);
        memcpy(a, b, ((i+1)*n/k)*sizeof(int));
        memset(b, 0, n*sizeof(int));
    }
    free(b);

    return a;
}

int* partial_sort(int* a, int n, int k) {
#pragma omp parallel
    {
```

```
        nthrds = omp_get_num_threads();
#pragma omp for
        for (int i = 0; i < k; ++i) {
            sort(a + i*n/k, (i+1)*n/k - i*n/k);
            // the length of subarray is ((a + (i+1)*n/k) - (a + i*n/k))
        }
    }
    printf("n threads = %d\n", nthrds);
    nthrds = 0;

    return a;
}

long long test_sort(int n, int turns) {
    long long time = 0;
    for (int i = 0; i < turns; ++i) {
        int* a = calloc(n, sizeof(int));
        fill(a, n);

        long long t1 = get_time();
        sort(a, n);
        long long t2 = get_time();

        time += t2 - t1;
        free(a);
    }

    return time/turns;
}

long long test_psort(int n, int k, int turns) {
    long long time = 0;
    for (int i = 0; i < turns; ++i) {
        int* a = calloc(n, sizeof(int));
        fill(a, n);

        long long t1 = get_time();
        partial_sort(a, n, k);
```

```
        merge(a, n, k);
        long long t2 = get_time();

        // partial_print(a, n, k); printf("\n\n");

        time += t2 - t1;
        free(a);
    }

    return time/turns;
}

int main(int argc, char** argv) {
    if (argc < 4) {
        printf("usage: ./m n k turns\n");
        return 1;
    }

    srand(time(0));

    int n = atoi(argv[1]);
    int k = atoi(argv[2]);
    int turns = atoi(argv[3]);

    if (k > n) {
        k = n;
        printf("k > n => assumed k = n\n");
    }

    long long tpsort = test_psort(n, k, turns);

    printf("time of part: %lldus\n", tpsort);

    return 0;
}
```

Листинг 2: Исходный код программы для MPI

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <string.h>
#include <mpi.h>
#include <unistd.h>

int nthrds = 0;

void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int* sort(int* a, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = i+1; j < n; ++j) {
            if (a[i] > a[j]) swap(a+i, a+j);
        }
    }

    return a;
}

int* fill(int* a, int n) {
    for (int i = 0; i < n; ++i) {
        a[i] = rand();
    }

    return a;
}

int* print(int* a, int n) {
    printf("[%d", a[0]);
    for (int i = 1; i < n; ++i) {
```

```
        printf(", %d", a[i]);
    }
    printf("]");

    return a;
}

long long get_time() {
    struct timeval tv;
    gettimeofday(&tv, NULL);

    return tv.tv_usec + tv.tv_sec*1000000LL;
 }

long seconds(long long t) {
    return (t / 1000000);
}

long useconds(long long t) {
    return (t % 1000000);
}

int* partial_print(int* a, int n, int k) {
    printf("[");
    print(a, n/k);
    for (int i = 1; i < k; ++i) {
        printf(", ");
        print(a + i*n/k, (i+1)*n/k - i*n/k);
        // the length of subarray is ((a + (i+1)*n/k) - (a + i*n/k))
    }
    printf("]");

    return a;
}

int* merge_parts(int* a, int n, int* b, int k, int* c) {
    int* a_end = a + n;
    int* b_end = b + k;
```

```
    while ((a < a_end) && (b < b_end)) {
        if (*a < *b) {
            *c++ = *a++;
        } else {
            *c++ = *b++;
        }
    }

    if (!(a < a_end) && b < b_end) {
        while (b < b_end) {
            *c++ = *b++;
        }
    }
    if (!(b < b_end) && a < a_end) {
        while (a < a_end) {
            *c++ = *a++;
        }
    }

    return c;
}

int* merge(int* a, int n, int k) {
    int* b = calloc(n, sizeof(int));
    for (int i = 0; i < k; ++i) {
        merge_parts(a, i*n/k, a + i*n/k, (i+1)*n/k - i*n/k, b);
        memcpy(a, b, ((i+1)*n/k)*sizeof(int));
        memset(b, 0, n*sizeof(int));
    }
    free(b);

    return a;
}

long long test_sort(int n) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
int k;
MPI_Comm_size(MPI_COMM_WORLD, &k);

long long t1 = get_time();

if (rank == 0) {
    int local_n = n/k;
    int* a = calloc(n, sizeof(int));
    fill(a, n);

    // send parts to nodes
    for (int i = 1; i < k; ++i) {
        MPI_Send(a + i*n/k,        // buff
                (i+1)*n/k - i*n/k, // count
                MPI_INT,           // type
                i,                 // dest
                i,                 // tag
                MPI_COMM_WORLD);   // comm
    }

    sort(a, local_n);

    // recv parts from nodes
    for (int i = 1; i < k; ++i) {
        // recv part
        MPI_Status stat;
        MPI_Recv(a + i*n/k,        // buff
                (i+1)*n/k - i*n/k, // count
                MPI_INT,           // type
                i,                 // source
                i,                 // tag
                MPI_COMM_WORLD,    // comm
                &stat);
    }

    merge(a, n, k);

    free(a);
```

```
    } else {
        int local_n = (rank+1)*n/k - rank*n/k;
        int* a = calloc(local_n, sizeof(int));

        // recv part
        MPI_Status stat;
        MPI_Recv(a,                     // buff
                local_n,                // count
                MPI_INT,                // type
                0,                      // source
                rank,                   // tag
                MPI_COMM_WORLD,         // comm
                &stat);                 // stat

        // usleep(rank*1000);

        sort(a, local_n);

        MPI_Send(a,                 // buff
                local_n,            // count
                MPI_INT,            // type
                0,                  // dest
                rank,               // tag
                MPI_COMM_WORLD);    // comm

        free(a);
    }

    long long t2 = get_time();

    return t2 - t1;
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank;
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    srand(time(0));

    FILE* f = fopen("data/mpi_sort k = 1.data", "w");
    if (!f) {
        printf("bad file\n");
        return 1;
    }

    for (int i = 0; i < 6; ++i) {
        int n = 50000*(i+1);
        long long t = test_sort(n);
        fprintf(f, "%d %lld\n", n, t);
        if (rank == 0) printf("%f%%\n", ((double)(i+1)/6)*100);
    }

    // if (rank == 0) printf("time on n = %d is %ld s. and %ld us.\n", n, seconds(t), useconds(t)

    fclose(f);

    MPI_Finalize();
    return 0;
}
```