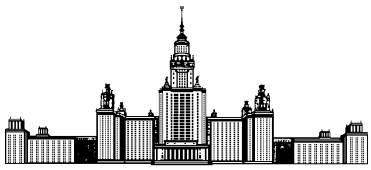
Московский государственный университет имени М. В. Ломоносова



Факультет Вычислительной Математики и Кибернетики
Кафедра Системного программирования

Отчет по заданию курса "Суперкомпьютеры и Параллельная Обработка Данных"

"Разработка параллельной версии программы для сортировки данных"

Выполнил:
студент 3 курса 327 группы
Майоров Егор Андреевич

Москва, 2022

Содержание

1 OpenMP

1.1 Алгоритм

Рассмотрим алгоритм сортировки пузырьком. Он достаточно сложен для распараллеливания в прямом виде, так как сравнение пар значений должно происходить последовательно.

```
void BubbleSort(double a[], int n) {
    for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i, j++)
            compare_exchange(a[j], a[j+1]);
}
```

В связи с этим для распараллеливания была использована более простая модификация этого алгоритма - алгоритм чет-нечетной сортировки. Суть алгоритма состоит в том, что в него вводятся два правила выполнения итераций метода - в зависимости от четности или нечетности номера итерации для обработки выбираются элементы с четными или нечетными индексами соответственно. Сравнение выделяемых значений осуществляется всегда с их правыми соседними элементами.

```
void OddEvenSort (double a[], int i) {
    for (int i = 0; i < n; i++)
        if (i % 2) {    // нечетная итерация
            for (int j = 0; j < n/2-2; j++)
                compare_exchange(a[2*j+1], a[2*j+2]);
            if (n % 2)  // сравнение последней пары при нечетном n
                compare_exchange(a[n-2], a[n-1]);
            } else      // четная итерация
                for (int j = 1; j < n/2-1; j++)
                    compare_exchange(a[2*j], a[2*j+1]);
}
```

Получение параллельного варианта для метода чет-нечетной перестановки оказывается довольно простым. Несмотря на то, что четные и нечетные итерации должны выполняться последовательно, сравнения пар значений на итерациях сортировки являются независимыми и могут быть выполненными параллельно.

Поскольку все вычислительные элементы имеют доступ к каждому значению в сортируемом массиве, сравнение значений a_i и a_j может быть выполнено любым вычислительным элементом. При наличии нескольких вычислительных элементов появляется возможность одновременно выполнять операции сравнения и перестановки над несколькими парами значений.

1.2 Результаты

В этой программе использовалось следующее средство OpenMP:

```
# pragma omp parallel for num_threads(num_threads)
```

Директива $parallel for$ определяет параллельный регион, который является кодом, который будет выполняться несколькими нитями параллельно, вызывает разделение работы в цикле внутри параллельного региона.

Клауза $num\_threads$ позволяет указать количество нитей, которое будет использовано для работы.

В таблице 1 представлено время работы программы в зависимости от количества нитей и размера массива, который был отсортирован.

Каждый тест был запущен несколько раз и время работы было выбрано усредненным.

Данные \ Нити	1	2	4	8	16	32	64
1024	0.0406	0.0271	0.0375	–	–	–	–
4096	0.0302	0.0285	0.0471	–	–	–	–
16384	0.2868	0.1505	0.1286	0.1563	–	–	–
65536	6.5958	1.9219	1.3159	0.9672	–	–	–
262144	94.1839	80.7403	94.8337	97.6151	74.2627	–	–
524288	322.5820	121.3423	233.6435	108.5008	297.0080	121.6732	–
1048576	1686.8181	1576.5273	317.7820	178.0112	1023.1223	424.3811	243.7150
2097152	–	–	1476.2896	>1800.0	>1800.0	>1800.0	906.6285

Таблица 1: Время работы программы в зависимости от количества данных и количества нитей.

По таблице видно, что в какой-то момент при увеличении количества нитей программа теряла эффективность, а затем увеличение количества нитей снова давало прирост в производительности, но не такой, как можно было наблюдать при наилучшем количестве нитей.

На графике 1 более наглядно представлена зависимость времени работы программы от количества нитей и размера массива.

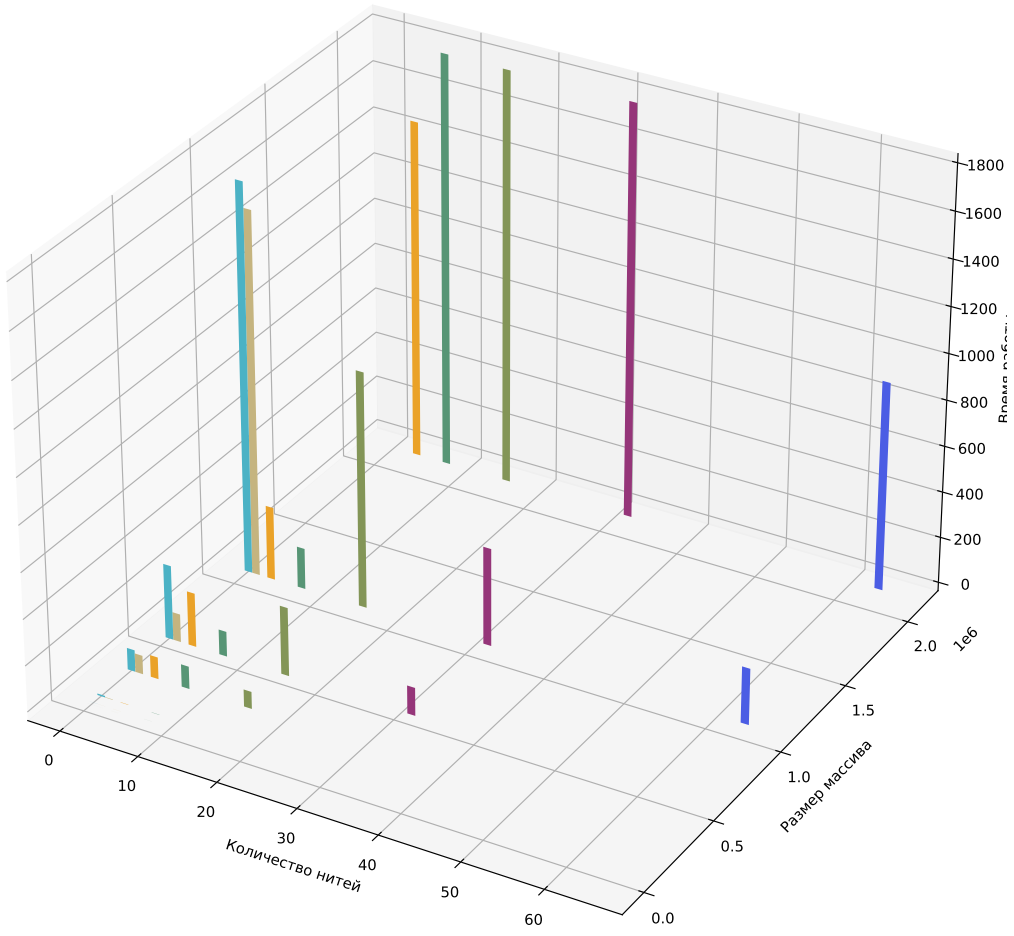


Рис. 1: Зависимость времени работы программы от количества нитей и размера массива

Компиляция программы:

```
xlc_r -qsmp=omp openmp.c -o openmp
```

Запуск программы:

```
mpisubmit.pl -w 00:30 openmp -- <кол-во_нитей> <размер_массива>
```

2 MPI

2.1 Алгоритм

Для этой программы также был выбран алгоритм сортировки пузырьком. Реализована схема, которая напоминает чет-нечетную сортировку. Массив, который необходимо отсортировать, разбивается на части равного размера. Считается, что количество процессов делит размер массива нацело. Каждая часть посылается отдельному процессу для сортировки с помощью функции $MPI\_Scatter()$.

Процессы сортируют свою часть с помощью обычной сортировки пузырьком, а затем происходит обмен данными между процессами-соседями.

Каждый из процессов выполняет *tasks* итераций отбмена с соседями, где *tasks* - число MPI процессов.

Обмен данными, то есть слияние двух уже отсортированных массивов в один происходит с помощью блокирующих операций $MPI\_Send()$ и $MPI\_Recv()$.

На четной итерации процессы с четными номерами обмениваются данными с соседями справа, а процессы с нечетными номерами обмениваются с соседями слева. Два массива объединяются в один, который сортируется и снова распределяется между двумя процессами.

На нечетной итерации происходят аналогичные действия, только процессы с четными и нечетными номерами меняются ролями.

В итоге после *tasks* итераций получается отсортированный массив, который собирается в главном процессе с помощью функции $MPI\_Gather()$.

2.2 Результаты

Функция $MPI\_Init()$ должна быть вызвана первой среди всех функций MPI. Она определяет среду выполнения.

Функция $MPI\_Comm\_size()$ определяет число процессов в коммуникаторе.

Функция $MPI\_Comm\_rank()$ определяет номер процесса в коммуникаторе. Нумерация начинается с 0.

Функция $MPI\_Wtime()$ возвращает для каждого вызвавшего ее процесса время в секундах (астрономическое), прошедшее с некоторого момента в прошлом. Момент времени, используемый в качестве точки отсчета, не будет изменен за время существования процесса.

$MPI\_Send()$ - стандартная блокирующая передача. Аргументы функции - адрес первого элемента в буфере передачи; количество элементов в буфере передачи; тип данных MPI (используется MPI_INT) пересылаемого элемента; ранг процесса получателя; тег сообщения; коммуникатор.

$MPI\_Recv()$ - стандартный блокирующий прием. Аргументы функции - такие же, как и в $MPI\_Send()$, но также тут используется код завершения MPI_STATUS_IGNORE.

Функция $MPI\_Scatter()$ разбивает сообщение из буфера посылки главного процесса на равные части и посылает i-тую часть в буфер приема процесса с номером i (в том числе и самому себе). Аргументы функции - адрес начала размещения блоков распределяемых данных (используется только в главном процессе); число элементов, посылаемых каждому процессу; тип посылаемых элементов; адрес начала буфера приема; число получаемых элементов; тип получаемых элементов; номер процесса-отправителя; коммуникатор. Главный процесс использует оба буфера (посылки и приема), поэтому в вызываемой им подпрограмме все параметры являются существенными. Остальные процессы группы являются только получателями, поэтому для них параметры, специфицирующие буфер посылки, не существенны. Операция Scatter является обратной по отношению к Gather.

Функция $MPI\_Gather()$ собирает блоки данных, посылаемых всеми процессами группы в один массив главного процесса. Аргументы функции - адрес начала размещения данных; число посылаемых элементов; тип посылаемых элементов; адрес начала буфера примера (используется только главным процессом); число элементов, получаемых от каждого процесса (также используется только главным процессом); тип получаемых элементов; номер процесса получателя; коммуникатор. Длина блоков предполагается одинаковой. Объединение происходит в порядке увеличения номеров процессов-отправителей. Длина массива, в который будут размещены данные, должна быть достаточной для их размещения. Операция Gather является обратной по отношению к Scatter.

Функция $MPI\_Finalize()$ завершает работу программы.

В таблице 2 представлено время работы программы в зависимости от количества процессов и размера массива, который был отсортирован.

Каждый тест был запущен несколько раз и время работы было выбрано усредненным.

Процессы / **Данные**	**1**	**2**	**4**	**8**	**16**	**32**	**64**
1024	0.0074	0.0019	0.0008	–	–	–	–
4096	0.1182	0.0301	0.0078	–	–	–	–
16384	1.8792	0.4719	0.1187	0.0321	–	–	–
65536	30.5530	7.6472	0.9259	0.5065	–	–	–
262144	438.0281	122.3935	31.3006	8.0863	2.0795	–	–
524288	>1800.0	486.2384	123.3313	30.5605	8.2154	2.2240	–
1048576	–	>1800.0	484.9612	129.9041	32.6914	8.1624	2c
2097152	–	–	>1800.0	517.6383	154.6050	127.1711	8c

Таблица 2: Время работы программы в зависимости от количества данных и количества процессов.

В таблице 2 в некоторых местах указано время работы >1800.0. Запустить более чем на 30 минут задачи не удалось, но предположительно во всех трех местах, где указано это число, программа должна работать около 33 минут. Также на 64 процессах указано не точное время, а примерное, полученное из анализа таблицы. Запустить программу на 64 процессах не удалось, так как она вставала в бесконечно долгую очередь.

Также при размере массива 2097152 иногда время работы оказывалось больше, чем предполагалось изначально. Например, при 16 процессах иногда время работы составляло около 500 секунд, что в 4 раза больше ожидаемого результата. И при 32 процессах несколько запусков программы показали среднее время 127 секунд, когда ожидалось около 30-32 секунд.

Можно заметить, что при увеличении числа процессов и размера массива в одно и то же количество раз (например в 2 или в 4 раза), время работы программы практически не изменяется. Это значит, что при увеличении размеров массива, который нужно отсортировать, мы можем получить очень большой выигрыш в производительности, если использовать большое количество процессов.

На графике 2 более наглядно представлена зависимость времени работы программы от количества процессов и размера массива.

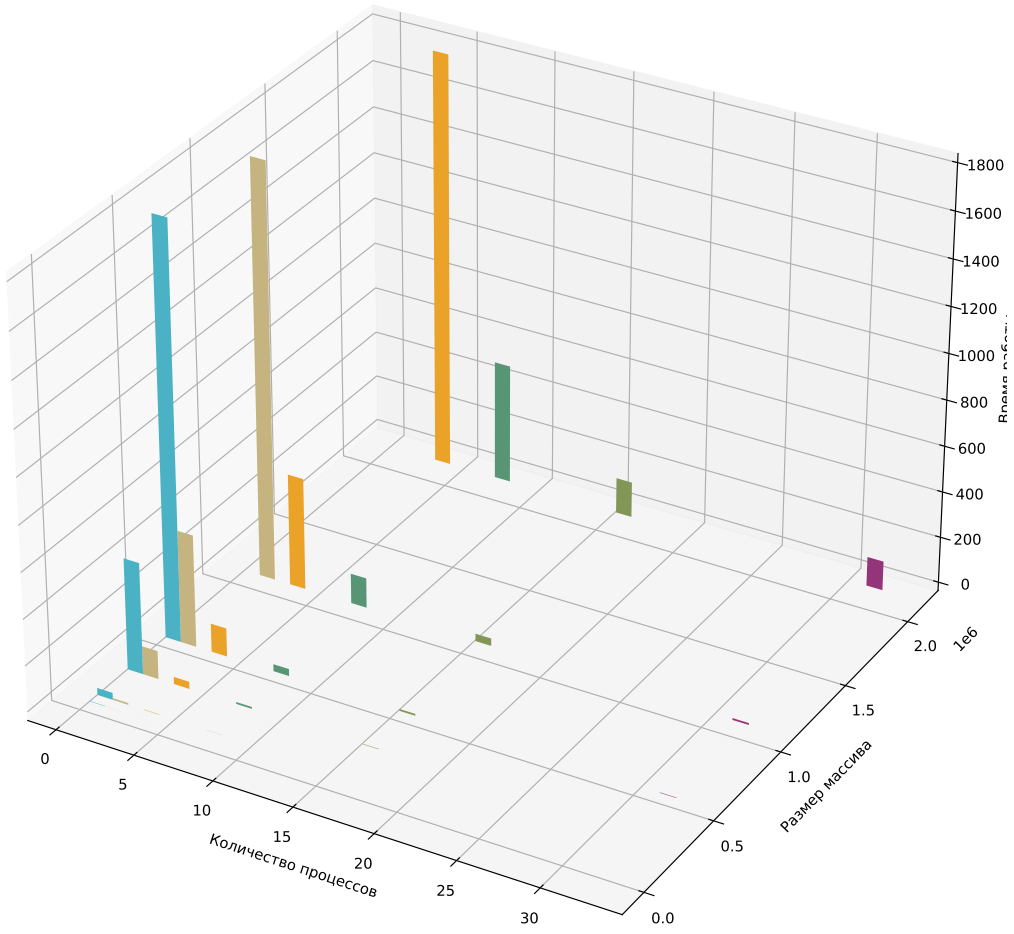


Рис. 2: Зависимость времени работы программы от количества процессов и размера массива

Компиляция программы:

```
mpicc mpi.c -o mpi
```

Запуск программы:

```
mpisubmit.pl -p <кол-во_процессов> -w 00:30 mpi -- <размер_массива>
```

3 Локальный компьютер

Также были проведены тесты на моем ноутбуке. Результаты запусков программ представлены ниже.

3.1 OpenMP

В таблице 3 представлены данные о времени работы программы openmp.c

Данные \ Нити	1	2	4
1024	0.0060	0.0061	0.0151
4096	0.1402	0.0923	0.1145
16384	2.0641	1.1622	1.3107
65536	8.1500	4.4803	6.2768
262144	143.01527	71.4109	69.3429
524288	601.7452	315.7415	275.1491

Таблица 3: Время работы программы в зависимости от количества данных и количества нитей.

3.2 MPI

В таблице 4 представлены данные о времени работы программы mpi.c

Данные \ Процессы	1	2	4
1024	0.0030	0.0007	0.0007
4096	0.1600	0.0300	0.0100
16384	2.6500	0.0632	0.1723
65536	10.4500	2.6412	0.7145
262144	169.0601	42.0032	12.1819
524288	678.4137	172.2198	44.0116

Таблица 4: Время работы программы в зависимости от количества данных и количества процессов.

4 Code

4.1 openmp.c

```c
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

void compare_exchange (int *i, int *j) {
    if (*i > *j) {
        int tmp = *i;
        *i = *j;
        *j = tmp;
    }
}

// Функция для алгоритма чет-нечетной сортировки
void parallel_odd_even_sort (int *arr, int n, int num_threads) {
    int upper_bound;
    if (n % 2 == 0)
        upper_bound = n / 2 - 1;
    else
        upper_bound = n / 2;
    for (int i = 0; i < n; i++) {
        if (i % 2 == 0) {
// четная итерация
#pragma omp parallel for num_threads(num_threads)
            for (int j = 0; j < n / 2; j++)
                compare_exchange(&arr[2 * j], &arr[2 * j + 1]);
        } else {
// нечетная итерация
#pragma omp parallel for num_threads(num_threads)
            for (int k = 0; k < upper_bound; k++)
                compare_exchange(&arr[2 * k + 1], &arr[2 * k + 2]);
        }
    }
}


int main (int argc, char **argv) {
    int num_threads = atoi(argv[1]);
    size_t size = atoi(argv[2]);
    int a[size];
    printf("Start generating array\n");
    // Заполнение массива случайными числами
    for (int i = 0; i < size; i++)
        a[i] = rand() % size;
    for (int i = 0; i < 15; i++)
        printf("%d\t", a[i]);
    printf("\nEnd generating array\n");
    double start = omp_get_wtime();
    parallel_odd_even_sort(a, size, num_threads); // вызов функции сортировки
    double end = omp_get_wtime();
    // вывод времени работы
    printf("Sort time: %f\n", end - start);
    return 0;
}
```

4.2 mpi.c

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

void swap (int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

void sort (int size, int* arr) {
    int i, j;
    for (i = size - 2; i >= 0; i--)
        for (j = 0; j <= i; j++)
            if (arr[j] > arr[j + 1])
                swap (&arr[j], &arr[j + 1]);
}

void merge (const int *in_a, int len_a, const int *in_b, int len_b, int *out) {
    int i, j;
    int out_count = 0;
    for (i = 0, j = 0; i < len_a; i++) {
        while ((in_b[j] < in_a[i]) && j < len_b) {
            out[out_count++] = in_b[j++];
        }
        out[out_count++] = in_a[i];
    }
    while (j < len_b) {
        out[out_count++] = in_b[j++];
    }
}

void pairwise_exchange (int local_n, int *local_a, int send_rank, int recv_rank) {
    // процесс-отправитель отправляет свой массив и ждет результата
    // процесс-получатель сортирует массивы и возвращает нужную половину
    int rank;
    int remote[local_n];
    int buf_all[2*local_n];
    const int merge_tag = 1;
    const int sorted_tag = 2;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == send_rank) {
        MPI_Send(local_a, local_n, MPI_INT, recv_rank, merge_tag, MPI_COMM_WORLD);
        MPI_Recv(local_a, local_n, MPI_INT, recv_rank, sorted_tag, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    } else {
        MPI_Recv(remote, local_n, MPI_INT, send_rank, merge_tag, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        merge(local_a, local_n, remote, local_n, buf_all);
        int their_start = 0, my_start = local_n;
        if (send_rank > rank) {
            their_start = local_n;
            my_start = 0;
        }
        MPI_Send(&(buf_all[their_start]), local_n, MPI_INT, send_rank, sorted_tag,
```

```
                    MPI_COMM_WORLD);
            for (int i = my_start; i < my_start + local_n; i++)
                local_a[i - my_start] = buf_all[i];
        }
    }

    void parallel_odd_even_sort (int n, int *a, int rank, int tasks) {
        // предполагается, что размер массива делится нацело на число процессов
        int i;
        int *local_a;
        // получаем номер процесса
        local_a = malloc(n / tasks * sizeof(int));
        // распределение массива по процессам
        MPI_Scatter(a, n / tasks, MPI_INT, local_a, n / tasks, MPI_INT, 0, MPI_COMM_WORLD);
        // сортируем часть массива, которая досталась процессу
        sort(n / tasks, local_a);
        // четно-нечетная сортировка
        for (i = 1; i <= tasks; i++) {
            if ((i + rank) % 2 == 0) {  // у номера процесса и номера итерации одинаковая четность
                if (rank < tasks - 1) {
                    pairwise_exchange(n / tasks, local_a, rank, rank + 1);
                }
            } else if (rank > 0) {
                pairwise_exchange(n / tasks, local_a, rank - 1, rank);
            }
        }
        // собираем части массива в один
        MPI_Gather(local_a, n / tasks, MPI_INT, a, n / tasks, MPI_INT, 0, MPI_COMM_WORLD);
    }

    int main (int argc, char **argv) {
        MPI_Init(&argc, &argv);
        int rank, tasks;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &tasks);
        int size = atoi(argv[1]);
        int *a = NULL;
        if (rank == 0) {
            a = malloc(size * sizeof(int));
        // Заполнение массива случайными числами
            for (int i = 0; i < size; i++)
                a[i] = rand() % size;
            double start_time = MPI_Wtime();
            parallel_odd_even_sort(size, a, rank, tasks);
            double end_time = MPI_Wtime();
            printf("Parallel time: %f\n", end_time - start_time);
            free(a);
        } else {
            parallel_odd_even_sort(size, a, rank, tasks);
        }
        MPI_Finalize();
        return 0;
    }
```