МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М. В. Ломоносова

Факультет вычислительной математики и
кибернетики

«Суперкомпьютеры и параллельная обработка данных»

Отчет
о выполненном задании
студента 321 группы факультета ВМК МГУ

Волчанинов Алексей Павлович

Москва
2022

Содержание

Постановка задачи и реализованный алгоритм

Задачей было реализовать метод прямоугольников. Для реализация была выбрана следующая спецификация: реализуем метод средних прямоугольников, то есть расчитываем значения целевой функции в серединах отрезков разбиения. Пользователь при запуске задает количество отрезков разбиения. Также пользователь задает концы отрезка, на котором должен быть вычислен интеграл и иные параметры, в зависимости от версии программы которую он запускает (см.далее)

Описание реализованных программ

Метод Прямоугольников - это метод приближенного вычисления определенного интеграла.

Пусть у нас задана функция $y = f(x)$ непрерывная на интервале [a;b] и требуется вычислить $\int_a^b f(x)dx$.

Разбиваем отрезок [a;b] на n отрезков вида $[x_{i-1}; x_i]$, i = 1,2,...,n с длиной $h = \frac{b-a}{n}$. Тогда точки x_i будут серединами отрезков $[x_{i-1}; x_{i+1}]$, i=1,2,...,n. Тогда видно, что $x_i = x_0 + h/2 + ih$, i=0,1,...,2n.

Далее считаем сумму значений функции в серединах отрезков, умноженную на длину отрезков разбиения.

Реализация последовательной программы на языке C:

```
#include "my_func.h"
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/time.h>

//function that change the function for
// which the integral is calculated
void
modify_file(char *new_str)
{
    int ds = open("my_func.c", O_CREAT|O_TRUNC|O_WRONLY);
    write(ds, "#include \"my_func.h\"\n\ndouble\n
____my_func(double x)\n{\n  return ", 60);
    write(ds, new_str, strlen(new_str));
    write(ds, ";\n}\n", 4);
    close(ds);
    return;
}

int
main(int argc, char **argv)
{
    //default number of iterations
    long int n = 1000000;
    //default segment ends
    double x0 = -10;
    double x1 = 10;

    //parsing command line: first is number of iterations
    //second and third are segment end and fourth is
    //function, if the user want to change it
    if (argc >= 2)
    {
```

```
        n = atoi(argv[1]);
    }
    if (argc >= 3)
    {
        x0 = strtod(argv[2], NULL);
    }
    if (argc >= 4)
    {
        x1 = strtod(argv[3], NULL);
    }
    if (argc >= 5)
    {
        modify_file(argv[4]);
        printf("You have set a new function ");
        printf("Please call make again and than run your programm\n");
        return 1;
    }

    //sequntial algorithm with loging the results into base file
    double res = 0;
    double h = (x1-x0)/n;
    double cur_x = x0 + h/2;
    struct timeval time_start, time_finish;
    gettimeofday(&time_start, NULL);
    for (int i = 0; i < n-1; i++)
    {
        res += my_func(cur_x);
        cur_x += h;
    }
    gettimeofday(&time_finish, NULL);
    float res_sec = (time_finish.tv_sec-time_start.tv_sec);
    float res_usec = (float)(time_finish.tv_usec-time_start.tv_usec);
    float res_time = res_sec + res_usec / 1000000;
    printf("result: %e has been achieved in %f s\n", res*h, res_time);
    FILE *f;
    f = fopen("base", "a");
    fprintf(f, "%d %f\n", n, res_time);
    fclose(f);
    return 0;
}
```

Листинг 1: последовательная реализация алгоритма

Функция и запуск на суперкомпьютере

Для иллюстрации работы алгоритма была выбрана следующая функция:

$$y = e^{(sin(e^{(-x)}) + cos(e^{(x)}))}$$

Реализация параллельной программы на OpenMP

Для распараллеливания были задействованы 2 "прагмы"OpenMP. Первая создает параллельную область с редукцией для получения макс. времени выполнения нити. Вторая - распределяет главный цикл между нитями с редукцией на суммирование результата. Подробнее - см. комментарии.

```
#include <omp.h>
#include <stdio.h>
#include "my_func.h"


int
main(int argc, char **argv)
{
    //default number of iterations
    long int n = 1000000;
    //default values of segment ends
    double x0 = -10;
    double x1 = 10;
    //default thread number
    int k = 100;

    //parsing the command line:
    //first is number of iterations second and
    //third are segment ends and fourth is amount of threads
    if (argc >= 2)
    {
        n = atoi(argv[1]);
    }
    if (argc >= 3)
    {
        x0 = strtod(argv[2], NULL);
    }
    if (argc >= 4)
    {
        x1 = strtod(argv[3], NULL);
    }
    if (argc >= 5)
    {
```

```
        k = atoi(argv[4]);
    }

    //making all variables we need
    //calculation result
    double res = 0;
    //the size of one segment of the partition
    double h = (x1-x0)/n;
    //starting point for calculating the function:
    // implement the method of middle rectangles
    double cur_x = x0 + h/2;
    //variables under time: start, end and difference.
    double time_start, time_finish, res_time;
    //launching parallel area with the reduction operation
    //to calculate max time of thread being in main loop
    //variables time_start and time_finish are private,
    //all the other - shared
    #pragma omp parallel default(none) private(time_start, time_finish)
     shared(x1, h, cur_x, n, res) reduction(max: res_time) num_threads(k)
    {
        time_start = omp_get_wtime();
        //launching main loop distribution between threads
        //with reduction operation to summarize the results
        //writing nowait, to measure maximum time of working
        #pragma omp for reduction(+ : res) nowait
        for (int i = 0; i < n; i += 1)
        {
            res += my_func(cur_x + i*h);
        }
        time_finish = omp_get_wtime();
        res_time = time_finish-time_start;
    }
    //parallel area has been done
    //showing result, and writing it down to omp_stats file
    printf("result: %e\n had been achieved in %e sec", res*h, res_time);
    FILE *f;
    f = fopen("omp_stats", "a");
    fprintf(f,"%d %d %f %f\n", k, n, (float)res*h, (float)res_time);
    fclose(f);
    return 0;
}
```

Листинг 2: OpenMP

Реализация параллельной программы с MPI

Для распараллеливания была выбрана следующая схема: процесс с рангом 0 является корневым и не принимает участия в расчете, он ждет результаотов от MPI_REDUCE. Остальные процессы считают свои результаты и время выполнения и передают их с помощью MPI_REDUCE (с спецификаторами MPI_SUM и MPI_MAX) корневому процессу. Подробнее - см. комментарии.

```
#include <mpi.h>
#include <stdio.h>
#include "my_func.h"

int
main(int argc, char **argv)
{
    //default iterations value
    long int n = 1000000;
    //default values for ends of segment
    double x0 = -10;
    double x1 = 10;

    //parsing launch parameters
    //first is number of iterations
    //second and third are ends of segment
    if (argc >= 2)
    {
        n = atoi(argv[1]);
    }
    if (argc >= 3)
    {
        x0 = strtod(argv[2], NULL);
    }
    if (argc >= 4)
    {
        x1 = strtod(argv[3], NULL);
    }

    //making all variables we need
    //calculation result
    double res = 0;
    //size of one segment of the partition
    double h = (x1-x0)/n;
    //initiating point to count function at:
    double cur_x = x0 + h/2;
    //measuring time: start, finish and difference
    double time_start, time_finish, res_time;
    //root-process variables for getting results from MPI_REDUCE
```

```
    double my_res, my_time;
    //initializing MPI workspace
    int rank, commSize;
    MPI_Init(&argc, &argv);
    time_start = MPI_Wtime();
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &commSize);
    //scenario of the main "working" processes
    if (rank != 0){
        //every working process making steps of cicle with numbers n,
        //so that: n%(commSize-1) == rank-1
        for (int i = rank-1; i < n; i += commSize-1)
        {
            res += my_func(cur_x + i*h);
        }
        //after making main loop, sending results with MPI_Reduce to
        //the root process, which will count sum of these results
        MPI_Reduce(&res ,&my_res, 1, MPI_DOUBLE,
         MPI_SUM, 0, MPI_COMM_WORLD);
        //measuring execution time and sending it to root-process
        //wih MPI_Reduce function with max specification
        time_finish = MPI_Wtime();
        res_time = time_finish-time_start;
        MPI_Reduce(&res_time, &my_time, 1, MPI_DOUBLE,
         MPI_MAX, 0, MPI_COMM_WORLD);
    }
    //root process scenario. getting values with
    //MPI_Reduce and writing them into "mpi_stats" file
    else
    {
        MPI_Reduce(&res ,&my_res, 1, MPI_DOUBLE,
         MPI_SUM, 0, MPI_COMM_WORLD);
        MPI_Reduce(&res_time, &my_time, 1, MPI_DOUBLE,
         MPI_MAX, 0, MPI_COMM_WORLD);
        printf("result: %e had been achieved in %e sec\n",
         my_res*h, my_time);
        FILE *f;
        f = fopen("mpi_stats", "a");
        fprintf(f,"%d %ld %f %f\n", commSize, n,
         (float)my_res*h, (float)my_time);
        fclose(f);
    }
    MPI_Finalize();
    return 0;
}
```

Листинг 3: MPI

Тестирование и запуск

Реализации параллельной программы OMP и MPI были многократно (по 5 раз на каждм наборе параметров)запущены на машине POLUS с разным количеством отрезков разбиения и задействованных ресурсов.

количество отрезков принимало следующие значения:

{10000000, 25000000, 35000000, 50000000}

число процессов для MPI программы принимало значения:

{2, 4, 8, 16, 32}

число нитей для OpenMP программы принимало значения:

{2, 4, 8, 16, 32, 64, 128, 160}

Для запуска использовались следующие скрипты(m-число итераций, n-число процессов/нитей)

OpenMP:

```
 source /polusfs/setenv/setup.SMPI
#BSUB -W 00:15
#BSUB -o omp.out
#BSUB -e omp.err
OMP_NUM_THREADS=n mpiexec ./omp_solution m 1 10 n
```

MPI:

```
 mpisubmit.pl -p n mpi_solution m 1 10
```

Сборка программмы

Осуществлялась с помощью makefile:

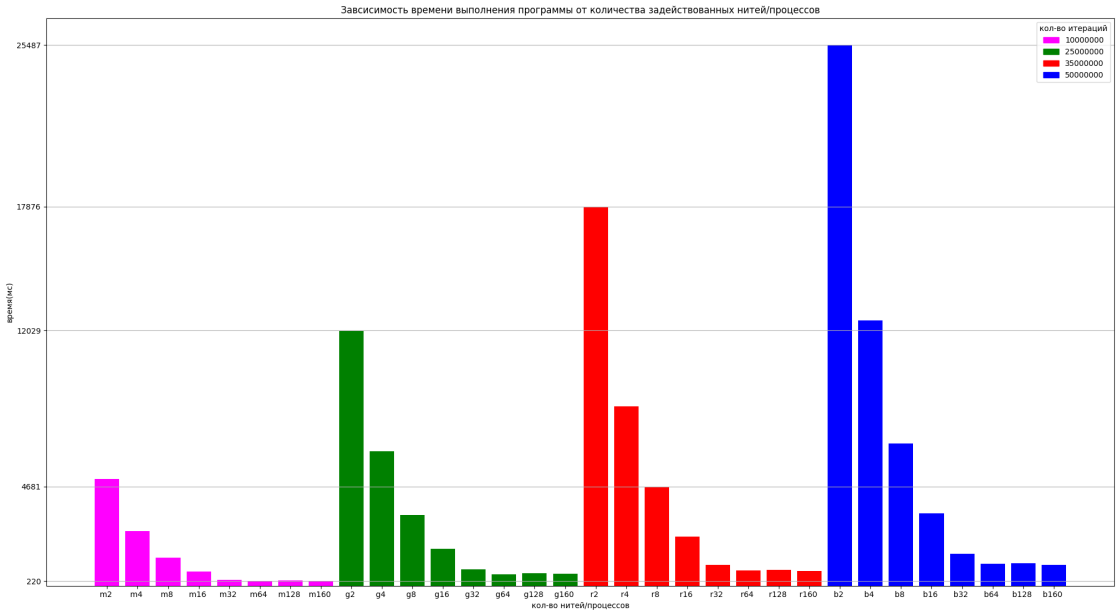
```
first: mpi_vers omp_vers simple_vers

simple_vers: my_func.o simple_vers.o
	gcc simple_vers.o my_func.o -lm -o simple_solution

mpi_vers: mpi_vers.o my_func.o
	mpicc mpi_vers.o my_func.o -lm -o mpi_solution

omp_vers: omp_vers.o my_func.o
	gcc -fopenmp omp_vers.o my_func.o -lm -o omp_solution

simple_vers.o :simple_vers.c my_func.h
	gcc simple_vers.c -c -lm

mpi_vers.o: my_func.h mpi_vers.c
	mpicc mpi_vers.c -c -lm

omp_vers.o: my_func.h omp_vers.c
	gcc -fopenmp omp_vers.c -c -lm

my_func.o: my_func.c my_func.h
	gcc my_func.c -c -lm

clean:
	rm -f simple_solution omp_solution mpi_solution
```

Листинг 4: Makefile

Результаты запусков

При выполнении программы данные складываются в файлы omp_stats и mpi_stats (для OpenMP и MPI версий соответственно). Каждому запуску программы соответсвует одна запись в этих файлах. Запись имеет вид: <число нитей/процессов><число итераций><результат><время выполнения>. Также при запуске последовательной версии данные складываются в файл base в формате <число итераций><время выполнения>. Для визуализации результатов было написано 2 скрипта на языке python3. Первый из них отрисовывает таблицу с файлов omp_stats и mpi_stats либо с ускорением относительно последовательной программы, либо со временем выполнения(в с) и с заданным числом знаков после запятой. Второй отрисовывает диаграмму зависимости времени(в мс) выполнения от числа итераций и количества задействованных ресурсов.

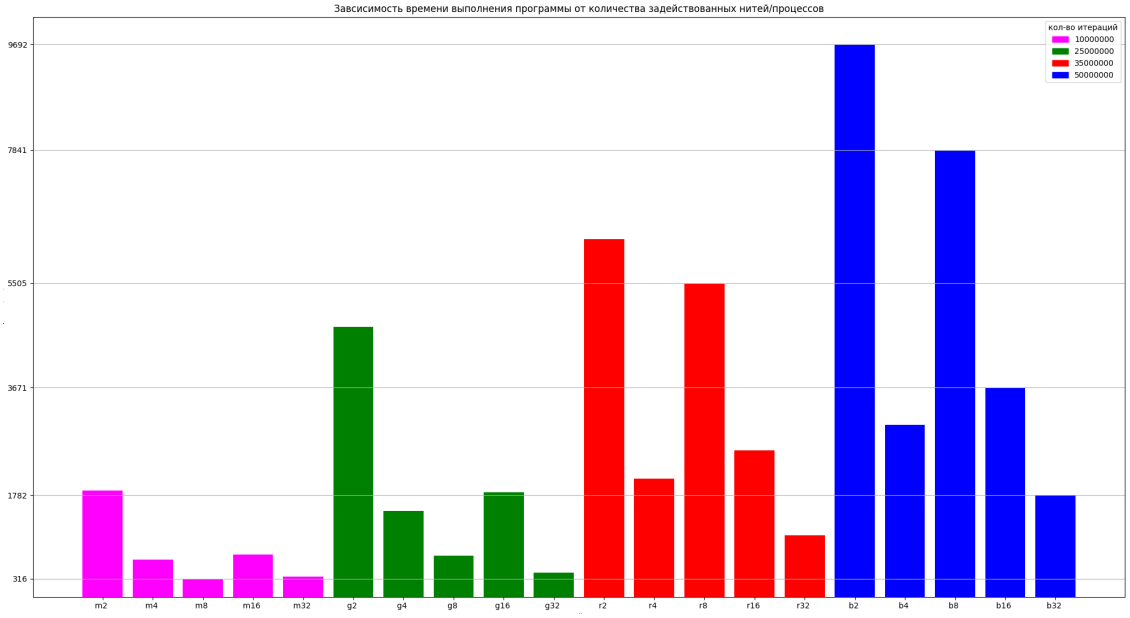
результаты для OpenMP




```
================================================ RESTART: /home/volch/3_kurs/SKiPOD/tabler.py ================================================
Введите файл, для которого хотиите просмотреть статистику: omp_stats_2
Если Вы хотите увидеть время выполнения программы, в зависимости от кол-ва используемых ресурсов, введите 1
Если Вы хотите увидеть ускорение выполнения программой по отношению к последовательной версии, введите 2
2
введите кол-во знаков после запятой в выводе: 5

+----------------------------------------+--------+--------+--------+--------+--------+--------+--------+
| Кол-во итераций \ Кол-во нитей/процессов |    2   |    4   |    8   |   16   |   32   |   64   |  128   |  160   |
+----------------------------------------+--------+--------+--------+--------+--------+--------+--------+
|                10000000                | 0.29738 | 0.57881 | 1.12253 | 2.20228 | 5.35474 | 6.81916 | 5.69224 | 6.50174 |
|                25000000                | 0.31175 | 0.59160 | 1.12241 | 2.16323 | 4.77760 | 7.10369 | 6.44692 | 6.69995 |
|                35000000                | 0.29313 | 0.61874 | 1.11955 | 2.24920 | 5.31213 | 7.11651 | 7.03086 | 7.58244 |
|                50000000                | 0.29416 | 0.59889 | 1.11884 | 2.19355 | 4.94441 | 7.15412 | 7.10364 | 7.60123 |
+----------------------------------------+--------+--------+--------+--------+--------+--------+--------+--------+

В этой таблице отражено ускорение выполнения программы по сравнению с последовательной программой в зависмости от используемого числа нитей/процессов
```

```
========== RESTART: /home/volch/3_kurs/SKiPOD/tabler.py ==========
Введите файл, для которого хотиите просмотреть статистику: omp_stats_2
Если Вы хотите увидеть время выполнения программы, в зависимости от кол-ва используемых ресурсов, введите 1
Если Вы хотите увидеть ускорение выполнения программой по отношению к последовательной версии, введите 2
1
введите кол-во знаков после запятой в выводе: 5

+----------------------------------------+---------+---------+---------+---------+---------+---------+---------+
| Кол-во итераций \ Кол-во нитей/процессов |    2    |    4    |    8    |   16    |   32    |   64    |   128   |   160   |
+----------------------------------------+---------+---------+---------+---------+---------+---------+---------+
|                10000000                | 5.03376 | 2.58623 | 1.33353 | 0.67972 | 0.27955 | 0.21952 | 0.26298 | 0.23024 |
|                25000000                | 12.02850 | 6.33853 | 3.34093 | 1.73348 | 0.78489 | 0.52788 | 0.58166 | 0.55969 |
|                35000000                | 17.87640 | 8.46907 | 4.68061 | 2.32979 | 0.98646 | 0.73634 | 0.74531 | 0.69109 |
|                50000000                | 25.48744 | 12.51864 | 6.70100 | 3.41790 | 1.51633 | 1.04798 | 1.05542 | 0.98633 |
+----------------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+

В этой таблице отражено время выполнения программы при разном числе итераций в зависимости от используемого числа нитей/процессов
```

результаты для MPI




```
========== RESTART: /home/volch/3_kurs/SKiPOD/tabler.py ==========
Введите файл, для которого хотиите просмотреть статистику: mpi_stats
Если Вы хотите увидеть время выполнения программы, в зависимости от кол-ва используемых ресурсов, введите 1
Если Вы хотите увидеть ускорение выполнения программой по отношению к последовательной версии, введите 2
2
введите кол-во знаков после запятой в выводе: 5

+----------------------------------------+---------+---------+---------+---------+---------+
| Кол-во итераций \ Кол-во нитей/процессов |    2    |    4    |    8    |   16    |   32    |
+----------------------------------------+---------+---------+---------+---------+---------+
|                10000000                | 0.80195 | 2.28912 | 4.74197 | 2.02588 | 4.17106 |
|                25000000                | 0.79103 | 2.49059 | 5.18391 | 2.03873 | 8.73725 |
|                35000000                | 0.83418 | 2.52372 | 0.95192 | 2.03989 | 4.86228 |
|                50000000                | 0.77359 | 2.48598 | 0.95612 | 2.04241 | 4.20841 |
+----------------------------------------+---------+---------+---------+---------+---------+

В этой таблице отражено ускорение выполнения программы по сравнению с последовательной программой в зависмости от используемого числа нитей/процессов
```

```
========== RESTART: /home/volch/3_kurs/SKiPOD/tabler.py ==========
Введите файл, для которого хотиите просмотреть статистику: mpi_stats
Если Вы хотите увидеть время выполнения программы, в зависимости от кол-ва используемых ресурсов, введите 1
Если Вы хотите увидеть ускорение выполнения программой по отношению к последовательной версии, введите 2
1
введите кол-во знаков после запятой в выводе: 5

+----------------------------------------+---------+---------+---------+---------+---------+
| Кол-во итераций \ Кол-во нитей/процессов |    2    |    4    |    8    |   16    |   32    |
+----------------------------------------+---------+---------+---------+---------+---------+
|                10000000                | 1.86661 | 0.65393 | 0.31568 | 0.73891 | 0.35889 |
|                25000000                | 4.74053 | 1.50563 | 0.72337 | 1.83934 | 0.42919 |
|                35000000                | 6.28184 | 2.07637 | 5.50487 | 2.56885 | 1.07772 |
|                50000000                | 9.69157 | 3.01586 | 7.84143 | 3.67084 | 1.78152 |
+----------------------------------------+---------+---------+---------+---------+---------+

В этой таблице отражено время выполнения программы при разном числе итераций в зависимости от используемого числа нитей/процессов
```

Анализ результатов и выводы

По приведенным результатам можно сделать вывод, что версия на OpenMP хорошо горизантально масштабируется при количестве нитей от 2 до 64, однако при дальнейшем увелиинии числа нитей, ускорение программы по сути сходит на нет. Это может быть связано с тем, что при таком количестве нитей накладные расходы системы поддержки на их запуск и взаимодействие становятся более ощутимыми по сравнению с приростом скорости от количества используемых нитей. Масштабируемость MPI реализации не столь очевидна, что наглядно показано на диаграмме. Однако общая тенденция к ускорению сохраняется. Данные, которые получилось собрать с машины POLUS не достаточно показательные, чтобы судить о причинах недостаточной масштабируемости для MPI программы.

По таблицам с ускорением и временем выполнения видно, что при малом количестве нитей и процессов, MPI реализация выигрывает в скорости у OpenMP реализации. Однако при приближении к максимальному доступному числу ресурсов (160 нитей и 32 процесса соответственно), OpenMP программа начинает обгонять MPI реализацию. Также видно, что время выполнения почти линейно зависит от количества итераций. По результатам тестирования, можно сказать, что в среднем OpenMP программа при максимальном количестве задействованных нитей (160) ускоряется в 7 раз. В MPI реализации при максимальном числе задействованных процессов(32) на всех входных данных, кроме 25000000 итераций наблюдается прирост скорости примерно в 4.5 раза. На 25000000 итераций программа отработала незакономерно быстрее и ускорилась в 8.7 раз. Фактически, ускорение по сравнению с последовательной программой начинается с восьми нитей для OpenMP реализации и с четырех процессов для MPI.