МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В. Ломоносова

Факультет Вычислительной Математики и Кибернетики

Практикум по учебному курсу

"Суперкомпьютеры и Параллельная Обработка Данных"

Задание №1

Разработка параллельной версии программы для задачи Sor2D

Отчет

студентки 320 группы

факультета ВМК МГУ

Кирсановой Софьи Игоревны

2021 год

Постановка задачи

Разработать параллельную версию программы для задачи Sor2D с использованием технологии OpenMP, исследовать масштабируемость полученной программы в зависимости от числа нитей и размерности задачи, построить графики зависимости времени её выполнения от числа используемых ядер.

Необходимо оценить время выполнения программы в зависимости от количества нитей и размерности задачи.

Метод релаксации (Successive over-relaxation, SOR) — итерационный метод решения систем линейных алгебраических уравнений. Под размерностью задачи понимается количество линейных уравнений в системе.

Код параллельной программы

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

#define  Max(a,b) ((a)>(b)?(a):(b))
#define  Min(a,b) ((a)<(b)?(a):(b))

#define  N   (2048 + 2)
double   maxeps = 0.1e-7;
int itmax = 100;
int i, j, k;
double eps;
int nThreads = 4;

double A [N][N];

void relax();
void init();
void verify();

int main(int an, char **as)
{
    int it;
    if (omp_get_num_threads() > 4) { }
    double time = omp_get_wtime();
    omp_set_num_threads(1);
```

```
    init();
    for (it = 1; it <= itmax; it++)
    {
        eps = 0.;
        relax();
        printf("it =%4i eps = %f\n", it, eps);
        if (eps < maxeps) break;
    }
    verify();
    printf("time = %f\n", omp_get_wtime() - time);
    return 0;
}


void init()
{
    for (i = 0; i <= N - 1; i++)
    #pragma omp parallel for private(j),  shared(A, i)
    for (j = 0; j <= N - 1; j++)
    {
        if (i == 0 || i == N - 1 || j == 0 || j == N - 1) A[i][j] = 0.;
        else A[i][j] = (1. + i + j) ;
    }
}


void relax()
{
    int iam, limit;
    int numt = omp_get_num_threads();
    int* isync = (int*) malloc(numt * sizeof(int));
    double maxii = 0;
    double maxii_shared = 0;

#pragma omp parallel private(iam, limit, i), shared(A,numt, maxii_shared) firstprivate(maxii)
{
    iam = omp_get_thread_num();
    limit = Min(numt-1, N-2);
    isync[iam] = 0;

    omp_lock_t lock;
    omp_init_lock(&lock);

    #pragma omp barrier
    for (i = 1; i <= N - 2; i++)
    {
        if ((iam > 0) && (iam <= limit))
        {
```

```
            for ( ; isync[iam - 1] == 0; )
            {
                #pragma omp flush(isync)
            }
            isync[iam - 1] = 0;
            #pragma omp flush(isync)
        }
        #pragma omp parallel for private(j), shared(A, i), reduction(max:maxii)
        for (j = 1; j <= N - 2; j++)
        {
            double e;
            e = A[i][j];
            A[i][j] = (A[i - 1][j] + A[i + 1][j] + A[i][j - 1] + A[i][j + 1]) / 4.;
            maxii = Max(maxii, fabs(e -A[i][j]));
        }

        if (iam < limit)
        {
            for ( ; isync[iam] == 1; )
            {
                #pragma omp flush(isync)
            }
            isync[iam] = 1;
            #pragma omp flush(isync)
        }
    }
#pragma omp critical
    if(maxii > eps) maxii_shared = maxii;
    omp_destroy_lock(&lock);
    }
    eps = maxii_shared;
}


void verify()
{
    double s;
    s = 0.;
    for(i = 0; i <= N - 1; i++)
    #pragma omp parallel for private(j), shared(A, i), reduction(+:s)
    for(j = 0; j <= N - 1; j++)
        s = s + A[i][j] * (i + 1) * (j + 1)/(N * N);
    printf("    S = %f\n", s);
}
```

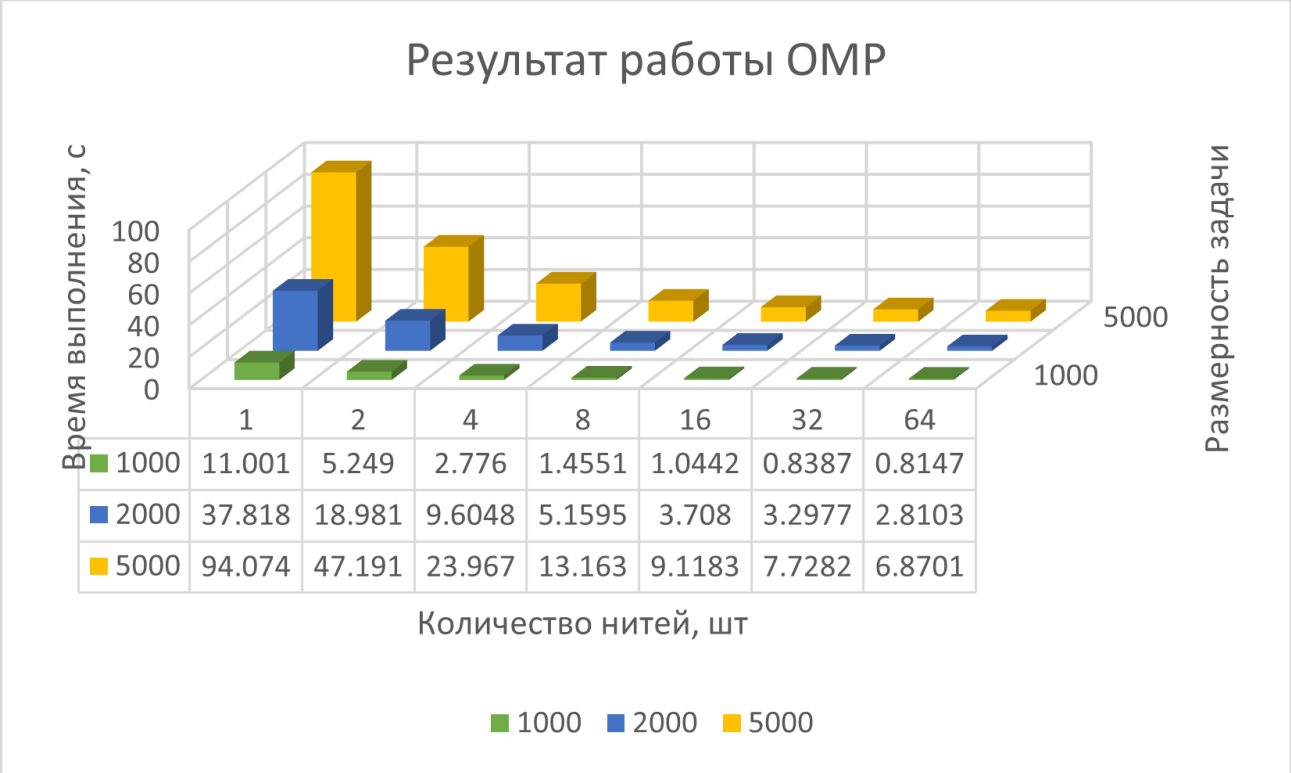
Результаты замеров времени выполнения

Работа программы была рассмотрена на суперкомпьютере Polus с различным числом нитей (1, 2, 4, 8, 16, 32 и 64) и с различной размерностью задачи (1000, 2000 и 5000).

Таблица с результатами OMP

Нити / время выполнения	*1000*	*2000*	*5000*
1	11.000788	37.817510	94.073729
2	5.248969	18.980589	47.191494
4	2.775974	9.604778	23.966895
8	1.455101	5.159510	13.163472
16	1.044231	3.708013	9.118330
32	0.838682	3.297730	7.728176
64	0.814691	2.810341	6.870052

График зависимости времени выполнения программы от количества потоков



	1	2	4	8	16	32	64
1000	11.001	5.249	2.776	1.4551	1.0442	0.8387	0.8147
2000	37.818	18.981	9.6048	5.1595	3.708	3.2977	2.8103
5000	94.074	47.191	23.967	13.163	9.1183	7.7282	6.8701

Вывод

Согласно результатам и диаграмме графика MPI, можно наблюдать, что время выполнения прямо пропорционально размерности задачи и обратно пропорционально количеству нитей. Таким образом, с увеличением количества нитей время выполнения снижается при любой рассмотренной размерности.