МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Практикум по курсу

"Суперкомпьютеры и параллельная обработка данных"

Разработка параллельной версии программы с использованием OpenMP

ОТЧЕТ

студента 321 учебной группы факультета ВМК МГУ

Сенюшкиной Алины Константиновны

Москва, 2023 г.

Оглавление

1 Цель работы

1. Разработка параллельной версии программы с использованием технологии OpenMP.
2. Исследовать масштабируемость полученной параллельной программы, построив графики зависимости времени выполнения от числа используемых ядер для различного объема входных данных.
3. Определить основные причины недостаточной масштабируемости программы при максимальном числе используемых ядер/процессоров.

2 Алгоритм

2.1 Базовый алгоритм

- На вход подается матрица размера N×N
- В функции init() происходит инициализация матрицы изначально заданным способом
- В функции relax() происходит основная часть работы алгоритма – релаксация исходной матрицы
- В функции verify() происходит финальное суммирование для вычисления ответа

2.2 Параллельная реализация с помощью директивы for

В функции relax() заданного алгоритма вычисление значения к матрице `A[i][j]`

- в первом вложенном цикле вычисляется по двум ближайшим соседям в столбце:

```
A[i][j] = (A[i - 1][j] + A[i + 1][j] + A[i - 2][j] + A[i + 2][j]) / 4.;
```

- во втором вложенном цикле вычисляется по двум ближайшим соседям в строке:

```
A[i][j] = (A[i][j - 1] + A[i][j + 1] + A[i][j - 2] + A[i][j + 2]) / 4.;
```

Таким образом, в первом вложенном цикле вычисления выполняются независимо в цикле по j, а во втором по i. Для вычисления максимума среди нитей используется секция `critical`, так как в ней может находиться одновременно не более одной нити.

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#define  Max(a,b) ((a)>(b)?(a):(b))

float maxeps = 0.1e-7;
int itmax = 100;
int i,j,k;

float eps;
float A [N][N];

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

int main(int argc, char **argv)
{
   omp_set_num_threads((int)strtol(argv[1], NULL, 10));
   double avg_time = 0.;

   int it;
   init();
   double start = omp_get_wtime();

   for (it = 1; it <= itmax; it++) {
       eps = 0;
       relax();
       if (!(it % 1000)) {
           printf("it=%4i   eps=%.6f\n", it, eps);
       }
       if (eps < maxeps) break;
   }
```

```
    double finish = omp_get_wtime();
    avg_time += finish - start;
    verify();
    printf("%lf\n", avg_time);
    return 0;
}

void init()
{
    for(i=0; i<=N-1; 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()
{
#pragma omp parallel shared(A, eps) private(i, j) default(none)
    {
        float e;

        for (i = 2; i <= N - 3; i++) {
#pragma omp for schedule(static)
            for (j = 1; j <= N - 2; j++) {
                A[i][j] = (A[i - 1][j] + A[i + 1][j] + A[i - 2][j] + A[i + 2][j]) / 4.;
            }
        }
#pragma omp barrier

#pragma omp for schedule(static)
        for (i = 1; i <= N - 2; i++) {
            for (j = 2; j <= N - 3; j++) {
                float tmp = A[i][j];
                A[i][j] = (A[i][j - 1] + A[i][j + 1] + A[i][j - 2] + A[i][j + 2]) / 4.;
                e = Max(e, fabs(tmp - A[i][j]));
            }
        }
#pragma omp critical
        eps = Max(eps, e);
    }
}

void verify()
{
    float s;
```

```
    s=0.;
    for (i=0; i<=N-1; i++) {
    for (j=0; j<=N-1; j++) {
        s = s + A[i][j] * (i + 1) * (j + 1) / (N * N);
    }
    }
    printf("\tS = %f\n", s);
}
```

2.3 Параллельная реализация с помощью директивы task

Нить попавшая в блок `single` создает пул задач, из которого остальные нити берут задачи на выполнение. Время выполнения существенно увеличивается с ростом количества задач, поэтому количество задач было уменьшено с помощью переменной `per_task`, которая равна количеству итераций в одном taske.

Далее приведен код только функции relax(), так как остальной код идентичен предыдущему.

```
void relax() {
#pragma omp parallel shared(A, eps, num_of_threads) private(i, j, k) default(none)
    {
        float e;
#pragma omp single
        {
            int per_task = (N - 2) / num_of_threads;
            int num_of_tasks = (N - 2) / per_task;
            if ((N - 2) % per_task != 0) {
                num_of_tasks += 1;
            }
            for (i = 2; i <= N - 3; i++) {
                for (k = 0; k < num_of_tasks; k++) {
#pragma omp task
                    {
                        for (j = k * per_task + 1; j <= Min((k + 1) * per_task, N - 2); j++) {
                            A[i][j] = (A[i - 1][j] + A[i + 1][j] + A[i - 2][j] + A[i + 2][j]) / 4.;
                        }
                    }
                }
#pragma omp taskwait
            }

            for (k = 0; k < num_of_tasks; k++) {
#pragma omp task private(e)
                {
                    for (i = k * per_task + 1; i <= Min((k + 1) * per_task, N - 2); i++) {
                        for (j = 2; j <= N - 3; j++) {
                            float tmp = A[i][j];
                            A[i][j] = (A[i][j - 1] + A[i][j + 1] + A[i][j - 2] + A[i][j + 2]) / 4.;
                            e = Max(e, fabs(tmp - A[i][j]));
                        }
```

```
                    }
#pragma omp critical
                    eps = Max(eps, e);
            }}}}}
```

2.4 Параллельная реализация с помощью MPI

Каждый процесс обрабатывает n-ую строку и n-ый столбец. Таким образом, каждому процессу выделена собственная область. Из-за сложной зависимости данных и важности последовательности действий было два варианта алгоритма: конвейерная реализация или использование коллективных операций. В силу сравнительно простой отладки был выбран алгоритм с коллективными операциями.

Рассмотрим функцию relax(). Первый цикл независим по столбцам, поэтому каждому процессу выделено block строк. После этого цикла процессы обмениваются данными. Так как следующий цикл независим по строкам, то i-ому процессу будет нужна i-ая строка. На данный момент у него есть только кусочек это строки block*block. Поэтому остальные процессы присылают ему недостающие (n-1)*block данных с помощью функции Gather(). Так происходит со всеми процессами.

После цикла независимого по строкам находится максимальный eps среди всех процессов с помощью функции Allreduce() и начинается подготовка к следующему вызову функции relax(). К новому вызову функции relax() i-ому процессу нужно обновить i-ый столбец, так как у него есть только block*block актуальных элементов этого столбца, а остальные устарели. В этот момент у каждого процесса есть block актуальных строк. Поэтому используется операция Scatter(), которая рассылает данные из этих строк соответствующим процессам.

```
int wrank, wsize;
int block, startrow, lastrow;
int main(int argc, char **argv)
{
   double avg_time = 0.;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &wrank);
   MPI_Comm_size(MPI_COMM_WORLD, &wsize);
   MPI_Barrier(MPI_COMM_WORLD);
   for (int rep_num = 0; rep_num < REPETITIONS; rep_num++) {
       block = (N - 2) / wsize;
       startrow = block * wrank + 1;
       if ((wrank == (wsize - 1)) && (N - 2) % wsize != 0) {
           lastrow = startrow + block + (N - 2) % wsize - 1;
       } else {
           lastrow = startrow + block - 1;
       }
       int it;
       init();
       double start, finish;
       if (!wrank) {
           start = MPI_Wtime();
       }
       for (it = 1; it <= itmax; it++) {
           eps = 0.;
```

```
            relax();

            if (eps < maxeps) break;
        }

        if (!wrank) {
            finish = MPI_Wtime();
            avg_time += finish - start;
        }

        verify();
        MPI_Barrier(MPI_COMM_WORLD);
    }
    if (!wrank) {
        avg_time /= REPETITIONS;
        printf("%lf", avg_time);
    }

    MPI_Finalize();

    return 0;
}

void init()
{
    for(i=0; i<=N-1; 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()
{
    for(i=2; i<=N-3; i++)
        for(j=startrow; j<=lastrow; j++)
        {
            A[i][j] = (A[i-1][j]+A[i+1][j]+A[i-2][j]+A[i+2][j])/4.;
        }

    for (int w = 0; w < wsize; w++) {
        for (i = block * w + 1; i <= block * (w + 1); i++) {
            MPI_Gather(&A[i][startrow], block, MPI_FLOAT, &A[i][1], block, MPI_FLOAT, w,
 MPI_COMM_WORLD);
        }
    }

    float local_eps = eps;

    for(i=startrow; i<=lastrow; i++)
        for(j=2; j<=N-3; j++)
        {
            float e=A[i][j];
            A[i][j] = (A[i][j - 1] + A[i][j + 1] + A[i][j - 2] + A[i][j + 2]) / 4.;
            local_eps=Max(local_eps, fabs(e-A[i][j]));
        }
```

```
    MPI_Allreduce(&local_eps, &eps, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);

    for (int w = 0; w < wsize; w++) {
        for (i = block * w + 1; i <= block * (w + 1); i++) {
            MPI_Scatter(&A[i][1], block, MPI_FLOAT, &A[i][startrow], block, MPI_FLOAT, w,
 MPI_COMM_WORLD);
        }
    }
}

void verify()
{
    float s=0.;
    for(i=0; i<=N-1; i++)
        for(j=startrow; j<=lastrow; j++)
        {
            s=s+A[i][j]*(i+1)*(j+1)/(N*N);
        }

    MPI_Allreduce(&s, &sum, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
}
```

3.1 Таблицы

FOR NO OPTION

size\proc	1	2	3	4	5	6	7	8	9
130	0.289976	0.169953	0.131525	0.112257	0.104066	0.097199	0.097475	0.097183	0.107754
258	1.209739	0.641676	0.505158	0.378651	0.336190	0.318895	0.096380	0.133064	0.282187
514	1.117140	0.503185	1.671929	1.835885	0.836347	1.028376	0.731445	1.636807	0.809230
1026	24.516315	8.643602	8.306271	6.185165	3.594552	3.171339	3.125410	2.018525	2.474451

size\proc	10	20	40	60	80	100	120	140	160
130	0.118393	0.134464	0.191009	0.252577	0.977179	1.017712	0.817251	1.022696	2.506085
258	0.288671	0.321264	0.386535	0.494994	1.094833	1.380274	1.464768	1.699572	1.853386
514	0.777967	0.743887	0.854704	0.981627	1.753120	2.557362	2.952515	3.267394	9.622062
1026	2.324821	1.841239	1.770834	2.216414	3.274205	4.963738	5.922888	6.356124	14.807844

FOR -O3

size\proc	1	2	3	4	5	6	7	8	9
130	0.014038	0.012825	0.020342	0.021962	0.023522	0.041849	0.049522	0.049675	0.059456
258	0.050759	0.040903	0.040307	0.045802	0.056338	0.068300	0.084709	0.078744	0.172008
514	0.190075	0.122799	0.110524	0.116797	0.134105	0.155822	0.164212	0.169699	0.425502
1026	0.754672	0.419653	0.340210	0.322034	0.336604	0.340136	0.380151	0.410684	0.844669

2050	2.891907	1.519453	1.135278	1.004542	1.178448	1.125956	1.145137	1.474754	1.889495

size\proc	10	20	40	60	80	100	120	140	160
130	0.060292	0.095045	0.167755	0.329362	0.564723	0.670328	0.804511	0.939615	1.655037
258	0.197304	0.228798	0.318016	0.556491	1.112746	1.317300	1.556096	1.805799	2.786739
514	0.493291	0.503389	0.645688	1.550833	1.862992	2.482585	3.126205	3.163995	5.578332
1026	1.059566	1.003023	1.382319	2.695226	3.208413	4.858796	5.872789	6.173964	8.868546
2050	1.710725	1.869945	2.974727	4.281504	6.772686	9.388751	10.265059	16.672516	35.103463

TASK -O3

size\proc	1	2	3	4	5	6	7	8	9
34	0.002398	0.005303	0.020826	0.016420	0.038806	0.049537	0.083293	0.081782	0.133792
66	0.004420	0.007713	0.027868	0.022964	0.086700	0.104321	0.148201	0.147721	0.215074
130	0.020966	0.038615	0.083010	0.081381	0.143612	0.199549	0.276344	0.339993	0.478998
162	0.021689	0.031692	0.061540	0.070049	0.194806	0.291735	0.416099	0.447883	0.646190

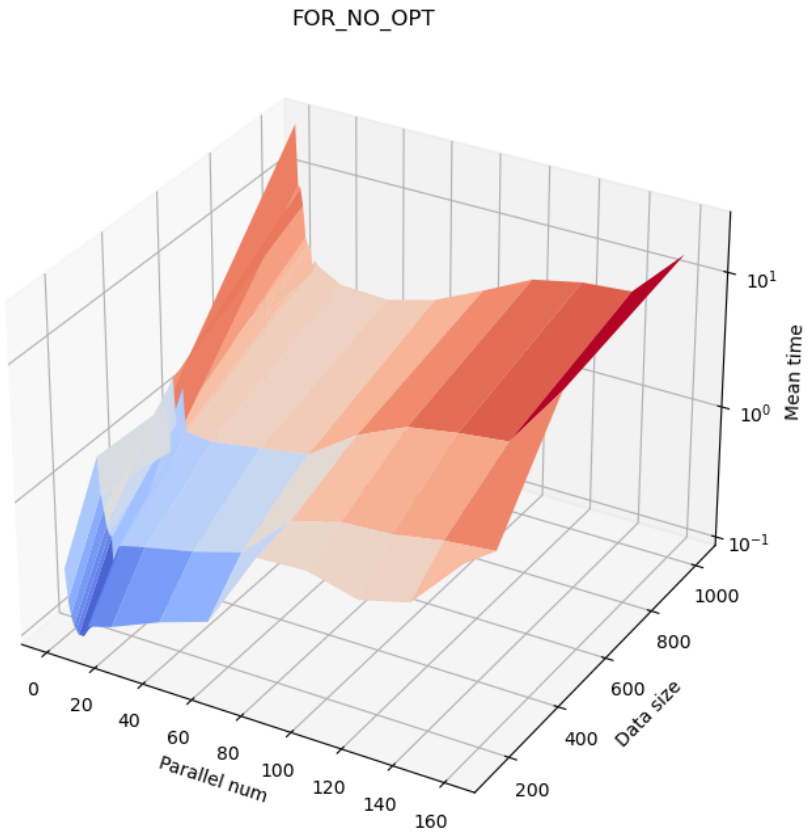
size\proc	10	20	40	60	80	100	120	140	160
34	0.140145	0.918859	0.000992	0.003594	0.003809	0.004553	0.004501	0.004946	0.005953
66	0.300302	1.093999	6.745611	11.082160	0.005213	0.005831	0.006040	0.006728	0.007044
130	0.581838	2.291754	12.125533	29.090573	93.309563	125.8978	145.96857	0.010301	0.010733
162	0.645249	2.537191	10.918511	35.009770	73.314700	99.415538	108.1093	113.9717	308.35377

MPI

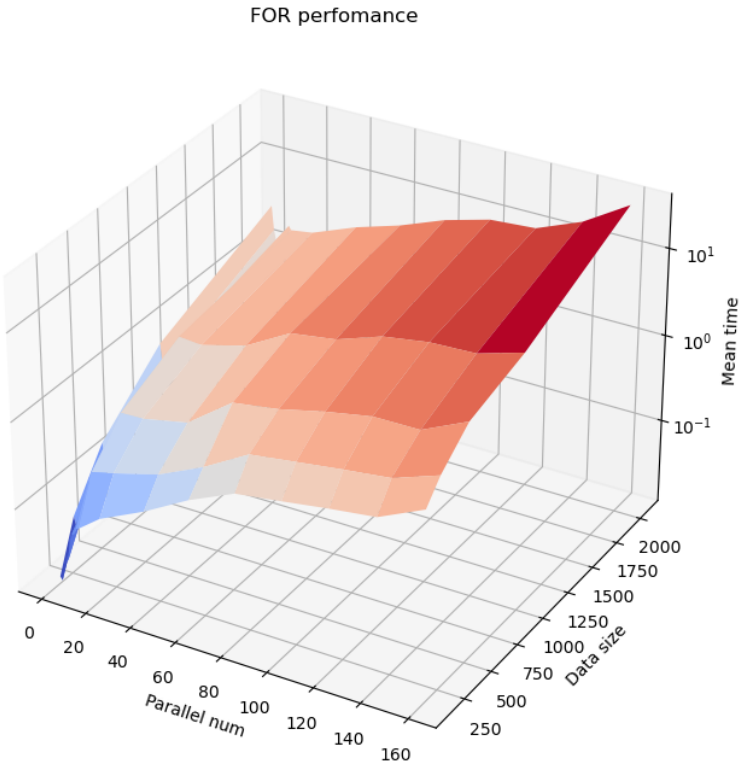
size\proc	1	2	3	4	5	6	7	8	9
130	0.043000	0.039411	0.056257	0.054416	0.048029	0.056946	0.053424	0.084297	0.071434
258	0.165551	0.124015	0.112442	0.118123	0.109130	0.147990	0.132125	0.160544	0.13016
514	0.649629	0.408738	0.334672	0.320389	0.291608	0.307341	0.295647	0.316136	0.311963
1026	2.573645	1.515802	1.533169	1.003978	0.876092	0.857425	0.798288	0.821499	0.809037

size\proc	10	20	40	60	80	100	120	140	160
130	0.083667	0.172974	0.190343	0.257110	0.149381	0.227807	0.270471	0.175601	0.301260
258	0.165816	0.292433	0.429342	0.668701	1.018983	1.101245	1.567107	0.656390	1.274695
514	0.337429	0.495766	0.410832	0.656343	2.151109	3.054524	3.927508	4.555432	8.478411
1026	0.852987	1.094037	2.017290	3.275252	4.921305	7.094722	9.382395	12.398588	20.086456

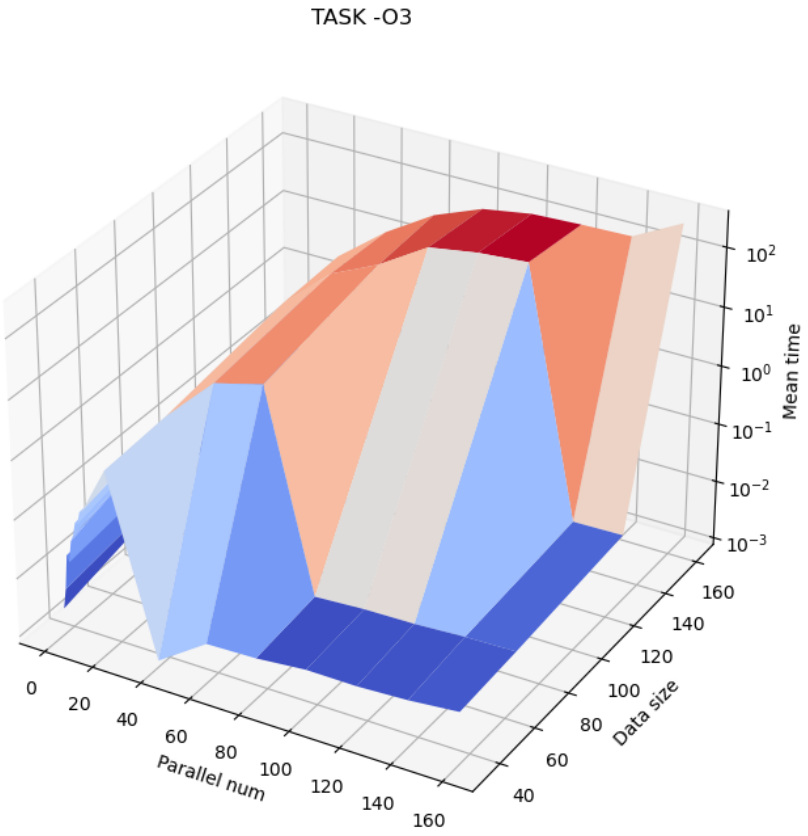
3.2 Графики



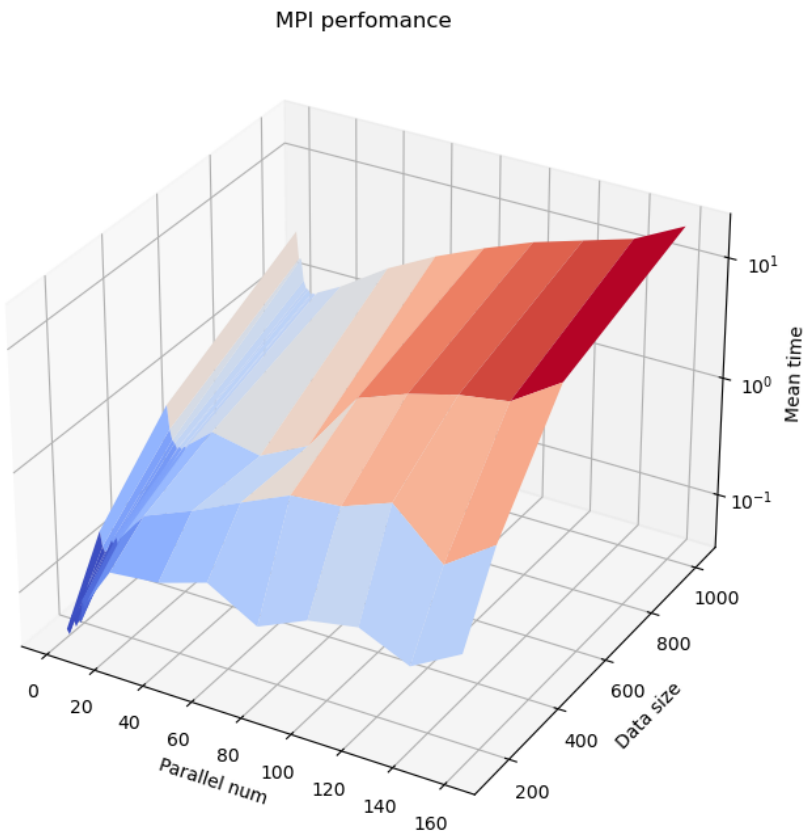
директива for OpenMP без опций оптимизации



директива for OpenMP -O3



директива task OpenMP



MPI

4 Описание результатов

Удалось успешно написать две версии программы на OpenMP с использованием директивы for и task, а также версию программы на MPI. По полученным графика для for можно сделать вывод, что опция -O3 ускоряет однопоточную версию, но сильно замедляет распараллеленную, а также за счет распараллеливания программа ускоряется до числа потоков = 8 и далее замедляется из-за накладных расходов связанных с выделением большего числа нитей независимо от объема входных данных.