МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В.Ломоносова

Факультет вычислительной математики и кибернетики

Компьютерный практикум по учебному курсу

«СУПЕРКОМПЬЮТЕРЫ И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ»

ЗАДАНИЕ.

Реализовать несколько версий параллельных программ с использованием технологии OpenMP и MPI.

ОТЧЕТ

о выполненном задании

студента 321 учебной группы факультета ВМК МГУ

Бутылкина Андрея Сергеевича

гор. Москва

2023 г.

Содержание

OpenMP

Постановка задачи

1) Для предложенного алгоритма реализовать несколько версий параллельных программ с использованием технологии OpenMP.

а) Вариант параллельной программы с распределением витков циклов при помощи директивы for.

б) Вариант параллельной программы с использованием механизма задач (директива task).

2) Исследовать эффективность полученных параллельных программ на суперкомпьютере Polus.

а) Исследовать влияние различных опций оптимизации, которые поддерживаются компиляторами (-O2, -O3)

3) Исследовать масштабируемость полученной параллельной программы: построить графики зависимости времени выполнения параллельной программы от числа используемых ядер для различного объёма входных данных.

4) Определить основные причины недостаточной масштабируемости программы при максимальном числе используемых ядер/процессоров.

Описание Алгоритма

Этот код реализует метод численного решения уравнения Пуассона в трехмерном пространстве методом итерационной релаксации. Процесс состоит из инициализации, итерационной релаксации и верификации решения.

Функция `init` инициализирует трехмерный массив `A` значением (4 + i + j + k) внутри области и нулевыми значениями на границе.

Функция `relax` выполняет итерационную релаксацию внутри области, используя среднее арифметическое соседних точек.

Функция `verify` вычисляет скалярную величину `s`, которая используется для проверки корректности значений в массиве `A` после выполнения всех итераций.

Функция `main` является основной частью программы. Она инициализирует переменные, вызывает функцию `init`, запускает цикл итераций `itmax`, проверяет условие завершения итераций, вызывает функцию `verify` для проверки решения.

Код исходной программы

Использованный язык — C

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

#define  N   (2*2*2*2*2*2+2)
float   maxeps = 0.1e-7;
int itmax = 100;
int i,j,k;

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

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

int main(int an, char **as)
{
    int it;

    init();

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

    verify();

    return 0;
}


void init()
{
    for(k=0; k<=N-1; k++)
    for(j=0; j<=N-1; j++)
    for(i=0; i<=N-1; i++)
    {
        if(i==0 || i==N-1 || j==0 || j==N-1 || k==0 || k==N-1)
        A[i][j][k]= 0.;
        else A[i][j][k]= ( 4. + i + j + k) ;
    }
}

void relax()
{
    for(k=1; k<=N-2; k++)
    for(j=1; j<=N-2; j++)
    for(i=1; i<=N-2; i++)
    {
        A[i][j][k] = (A[i-1][j][k]+A[i+1][j][k])/2.;
    }

    for(k=1; k<=N-2; k++)
    for(j=1; j<=N-2; j++)
    for(i=1; i<=N-2; i++)
    {
        A[i][j][k] =(A[i][j-1][k]+A[i][j+1][k])/2.;
    }

    for(k=1; k<=N-2; k++)
    for(j=1; j<=N-2; j++)
    for(i=1; i<=N-2; i++)
    {
        float e;
        e=A[i][j][k];
        A[i][j][k] = (A[i][j][k-1]+A[i][j][k+1])/2.;
        eps=Max(eps,fabs(e-A[i][j][k]));
    }

}
```

```
77
78 void verify()
79 {
80     float s;
81
82     s=0.;
83     for(k=0; k<=N-1; k++)
84     for(j=0; j<=N-1; j++)
85     for(i=0; i<=N-1; i++)
86     {
87         s=s+A[i][j][k]*(i+1)*(j+1)*(k+1)/(N*N*N);
88     }
89     printf("  S = %f\n",s);
90
91 }
```

Код и описание программы на OpenMP с помощью for

Использованный язык — C

```
1  #include <math.h>
2  #include <omp.h>
3  #include <stdlib.h>
4  #include <stdio.h>
5  #define  Max(a,b) ((a)>(b)?(a):(b))
6
7  #define  N   (2*2*2*2*2*2*2*2+2)
8  double   maxeps = 0.1e-7;
9  int itmax = 100;
10 int i,j,k;
11 double t1, t2;
12 double eps;
13 double A [N][N][N];
14
15 void relax();
16 void init();
17 void verify();
18
19 int main(int an, char **as)
20 {
21     int it;
22     omp_set_num_threads((int)strtol(as[1], NULL, 10));
23
24     t1 = omp_get_wtime();
25
26     init();
27     for(it=1; it<=itmax; it++)
28     {
29         eps = 0.;
30         relax();
31         //printf( "it=%4i   eps=%f\n", it,eps);
32         if (eps < maxeps) break;
33     }
34     verify();
35
36     t2 = omp_get_wtime();
37     printf("%.3lf\n", t2-t1);
38
39     return 0;
40 }
41
42
43 void init()
44 {
45 #pragma omp parallel shared(A) private(i, j, k)
46     {
47 #pragma omp for
48         for(i=0; i<=N-1; i++)
49         for(j=0; j<=N-1; j++)
50         for(k=0; k<=N-1; k++)
51         {
52             if(i==0 || i==N-1 || j==0 || j==N-1 || k==0 || k==N-1)
53             A[i][j][k]= 0.;
54             else A[i][j][k]= ( 4. + i + j + k) ;
55         }
56     }
57 }
```

```
58
59 void relax()
60 {
61 #pragma omp parallel shared(A, eps) private(i, j, k)
62     {
63         for(i=1; i<=N-2; i++)
64 #pragma omp for
65         for(j=1; j<=N-2; j++)
66         for(k=1; k<=N-2; k++)
67         {
68             A[i][j][k] = (A[i-1][j][k]+A[i+1][j][k])/2.;
69         }
70
71 #pragma omp for
72         for(i=1; i<=N-2; i++)
73         for(j=1; j<=N-2; j++)
74         for(k=1; k<=N-2; k++)
75         {
76             A[i][j][k] =(A[i][j-1][k]+A[i][j+1][k])/2.;
77         }
78
79 #pragma omp for reduction(max:eps)
80         for(i=1; i<=N-2; i++)
81         for(j=1; j<=N-2; j++)
82         for(k=1; k<=N-2; k++)
83         {
84             double vr=A[i][j][k];
85             A[i][j][k] = (A[i][j][k-1]+A[i][j][k+1])/2.;
86             eps=Max(eps,fabs(vr-A[i][j][k]));
87         }
88     }
89 }
90
91
92 void verify()
93 {
94     double s;
95
96     s=0.;
97
98 #pragma omp parallel shared(A) private(i, j, k)
99     {
100 #pragma omp for reduction(+:s)
101         for(i=0; i<=N-1; i++)
102         for(j=0; j<=N-1; j++)
103         for(k=0; k<=N-1; k++)
104         {
105             s=s+A[i][j][k]*(i+1)*(j+1)*(k+1)/(N*N*N);
106         }
107     }
108
109     //printf("  SS = %lf\n",s);
110
111 }
```

Описание

В функциях ‘init’, ’relax’, ‘verify’ с помощью директивы parallel создается параллельная область, которая задает общие(shared) и приватные(private) переменные для нитей. Внутри этой области используется директива parallel for. При подсчете eps используется reduction() для корректного вычисления значения. Аналогично reduction() используется для подсчета s.

Код и описание программы на OpenMP с помощью task

Использованный язык — C

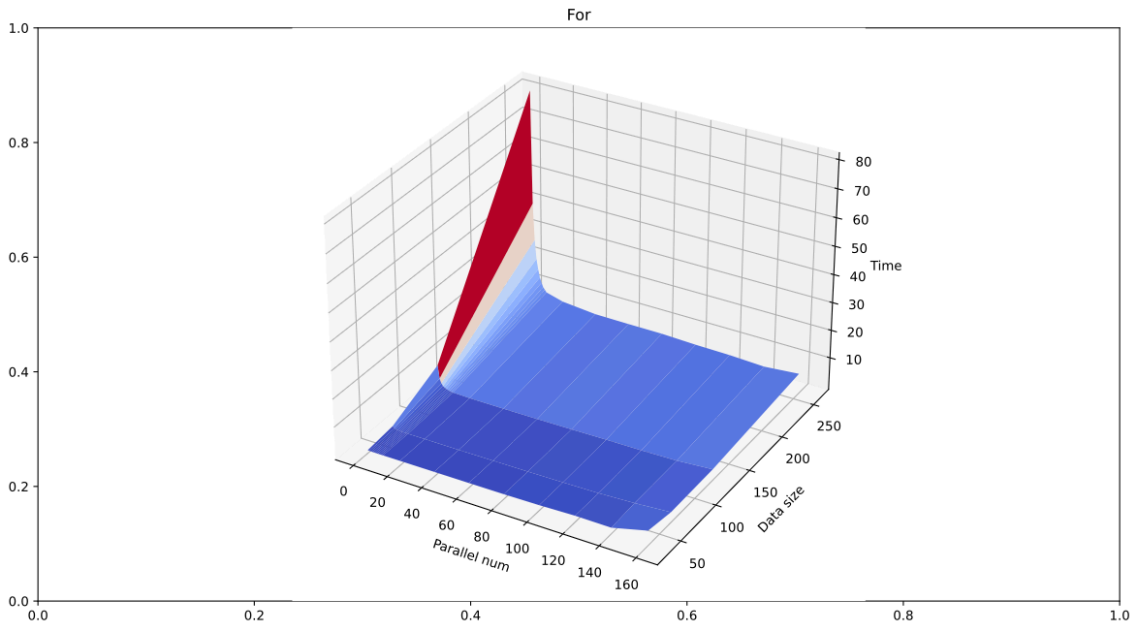
```c
#include <math.h>
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
#define  Max(a,b) ((a)>(b)?(a):(b))
#define  Min(a,b) ((a)<(b)?(a):(b))

#define  N   (2*2*2*2*2*2*2*2+2)

double   maxeps = 0.1e-7;
int itmax = 100;
int i,j,k, z;
double t1, t2;
double eps, s;
double A [N][N][N];
int cub, size, root, *i_s, *i_f;

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

int main(int an, char **as)
{
    int it;

    omp_set_num_threads((int)strtol(as[1], NULL, 10));

    size = (int)strtol(as[1], NULL, 10);

    cub = N / size;

    i_s = (int *) malloc(size * sizeof(int));
    i_f = (int *) malloc(size * sizeof(int));

    for (root = 0; root < size; ++root) {
        i_s[root] = cub * root;
        if (root == size - 1) {
            i_f[root] = N - 1;
        } else {
            i_f[root] = i_s[root] + cub - 1;
        }
    }

    t1 = omp_get_wtime();

    init();

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

    verify();

    t2 = omp_get_wtime();
    printf("%lf\n", t2-t1);

    free(i_s);
    free(i_f);

    return 0;
}


void init()
{
#pragma omp parallel shared(A, i_s, i_f, size) private(i, j, k, z)
    {
#pragma omp single
        {
            for (z = 0; z < size; ++z) {
#pragma omp task
                for(i=i_s[z]; i<=i_f[z]; i++)
                for(j=0; j<=N-1; j++)
                for(k=0; k<=N-1; k++)
                {
                    if(i==0 || i==N-1 || j==0 || j==N-1 || k==0 || k==N-1)
                    A[i][j][k]= 0.;
                    else A[i][j][k]= ( 4. + i + j + k) ;
```

```
                }
            }
        }
    }
}

void relax()
{
#pragma omp parallel shared(A, eps, i_s, i_f, size) private(i, j, k, z)
    {
#pragma omp single
        {
            for(i=1; i<=N-2; i++) {
                for (z = 0; z < size; ++z){
#pragma omp task
                    for(j=Max(i_s[z], 1); j<=Min(i_f[z], N-2); j++)
                    for(k=1; k<=N-2; k++)
                    {
                        A[i][j][k] = (A[i-1][j][k]+A[i+1][j][k])/2.;
                    }
                }
#pragma omp taskwait
            }
        }

#pragma omp single
        {
            for (z = 0; z < size; ++z){
#pragma omp task
                for(i=Max(i_s[z], 1); i<=Min(i_f[z], N-2); i++)
                for(j=1; j<=N-2; j++)
                for(k=1; k<=N-2; k++)
                {
                    A[i][j][k] =(A[i][j-1][k]+A[i][j+1][k])/2.;
                }
            }
        }

#pragma omp single
        {
            for (z = 0; z < size; ++z) {
                double ee;
#pragma omp task private(ee)
                {
                    for(i=Max(i_s[z], 1); i<=Min(i_f[z], N-2); i++)
                    for(j=1; j<=N-2; j++)
                    for(k=1; k<=N-2; k++)
                    {
                        double vr=A[i][j][k];
                        A[i][j][k] = (A[i][j][k-1]+A[i][j][k+1])/2.;
                        ee=Max(ee,fabs(vr-A[i][j][k]));
                    }
#pragma omp critical
                    eps = Max(ee, eps);
                }
            }
        }
    }
}


void verify()
{
#pragma omp parallel shared(A, i_s, i_f, size) private(i, j, k, z)
    {
        double ss = 0;

#pragma omp single
        {
            for(z = 0; z < size; ++z) {
#pragma omp task private(ss)
                {
                    for(i=i_s[z]; i<=i_f[z]; i++)
                    for(j=0; j<=N-1; j++)
                    for(k=0; k<=N-1; k++)
                    {
                        ss=ss+A[i][j][k]*(i+1)*(j+1)*(k+1)/(N*N*N);
                    }

#pragma omp atomic
                    s += ss;
                }
            }
        }
    }

    //printf("  S = %lf\n",s);
}
```

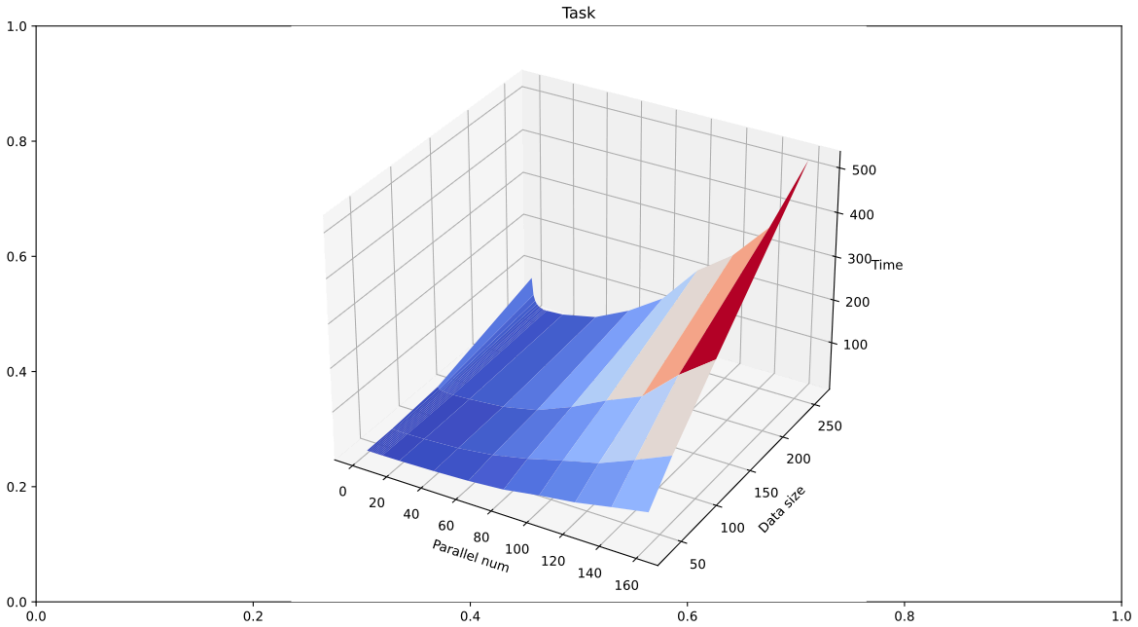
Описание

В функциях ‘init’, ’relax’, ‘verify’ с помощью директивы parallel создается параллельная область, которая задает общие(shared) и приватные(private) переменные для нитей. Внутри этой области используется директива parallel task (для создания задач), omp single нужна для задания задач одной нитью. Omp taskwait используется, чтобы следующие вычисления производились с корректными данными, обработанными предыдущими задачами. С помощью omp critical и omp atomic реализуется аналог reduction() из предыдущей версии. В ‘main’ задается количество задач — size, которое равняется количеству omp процессов.

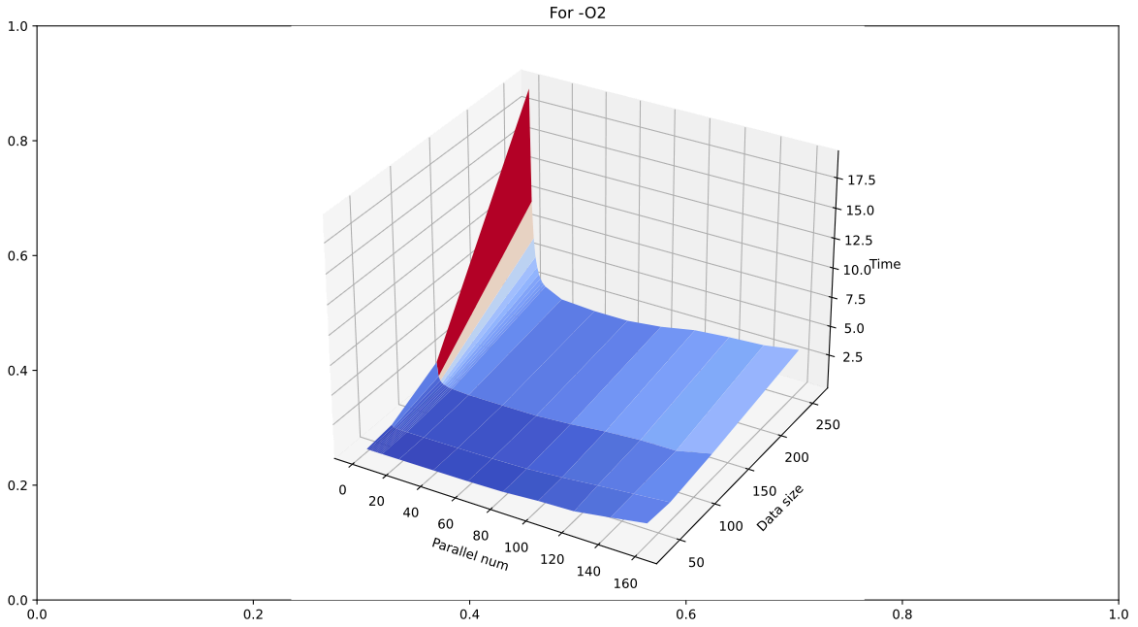
Тесты



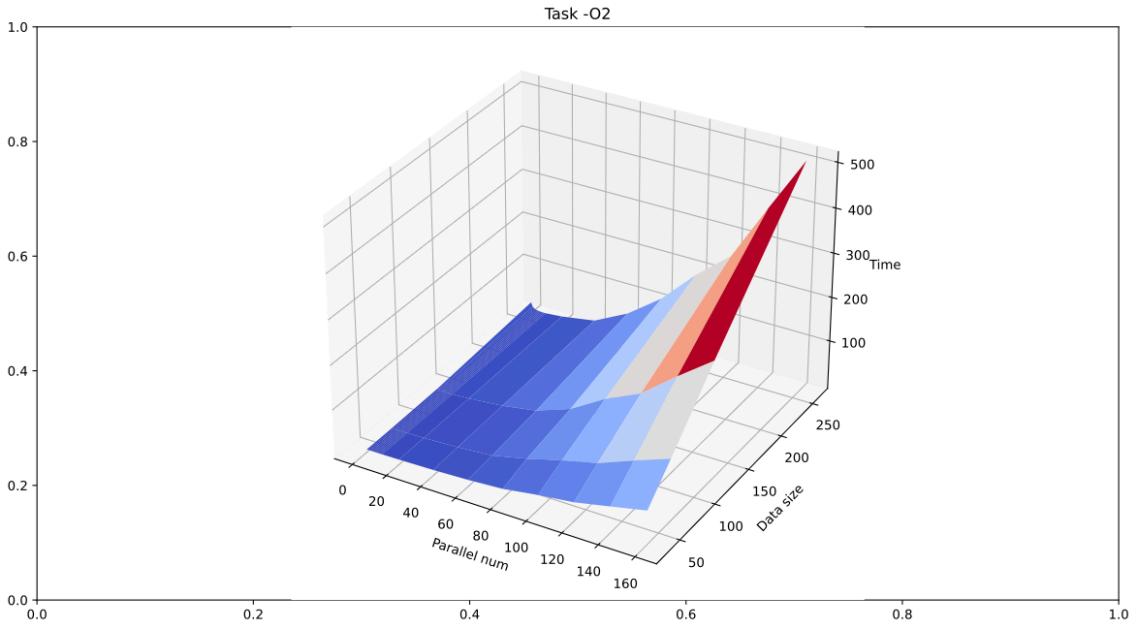
	1	2	3	4	5	6	7	8	9	10	20	40	60	80	100	120	140	160
34	0.154	0.078	0.055	0.042	0.038	0.034	0.031	0.028	0.030	0.031	0.039	0.053	0.073	0.110	0.283	0.648	0.557	3.669
66	1.192	0.602	0.418	0.308	0.254	0.224	0.204	0.169	0.174	0.155	0.182	0.172	0.231	0.285	0.422	0.614	1.159	2.036
130	9.392	4.708	3.176	2.372	1.934	1.645	1.440	1.252	1.177	1.089	0.913	0.876	0.920	1.012	1.173	1.240	1.473	2.338
258	80.676	40.385	27.194	20.274	16.566	13.673	11.885	10.420	9.630	8.817	7.111	5.933	5.820	5.807	5.367	5.174	4.612	5.688



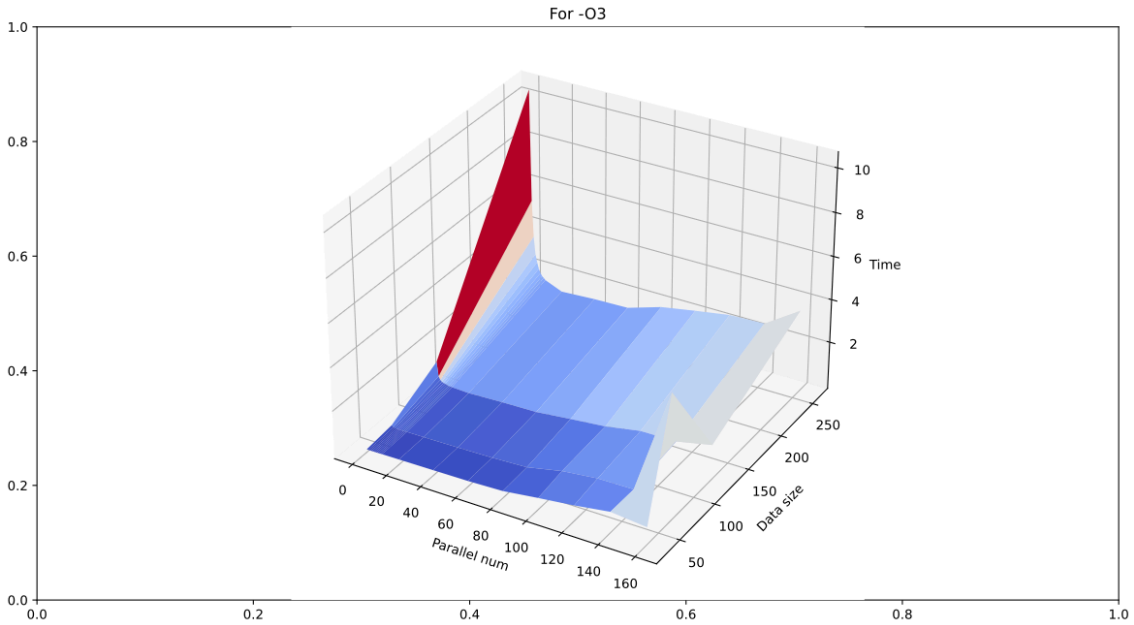
	1	2	3	4	5	6	7	8	9	10	20	40	60	80	100	120	140	160
34	0.154	0.084	0.061	0.056	0.071	0.079	0.090	0.083	0.119	0.124	0.590	2.524	5.761	11.945	24.835	35.463	51.405	66.393
66	1.195	0.611	0.439	0.349	0.289	0.263	0.271	0.261	0.295	0.378	1.225	5.403	15.084	32.069	53.605	73.453	106.639	142.244
130	9.408	4.729	3.204	2.493	2.017	1.901	1.701	1.430	1.488	1.195	3.168	9.813	26.861	59.090	98.997	133.246	206.610	265.450
258	80.736	40.464	27.233	20.652	16.938	13.852	13.269	10.948	10.972	10.892	13.068	30.140	68.883	121.320	210.427	268.276	354.014	525.528



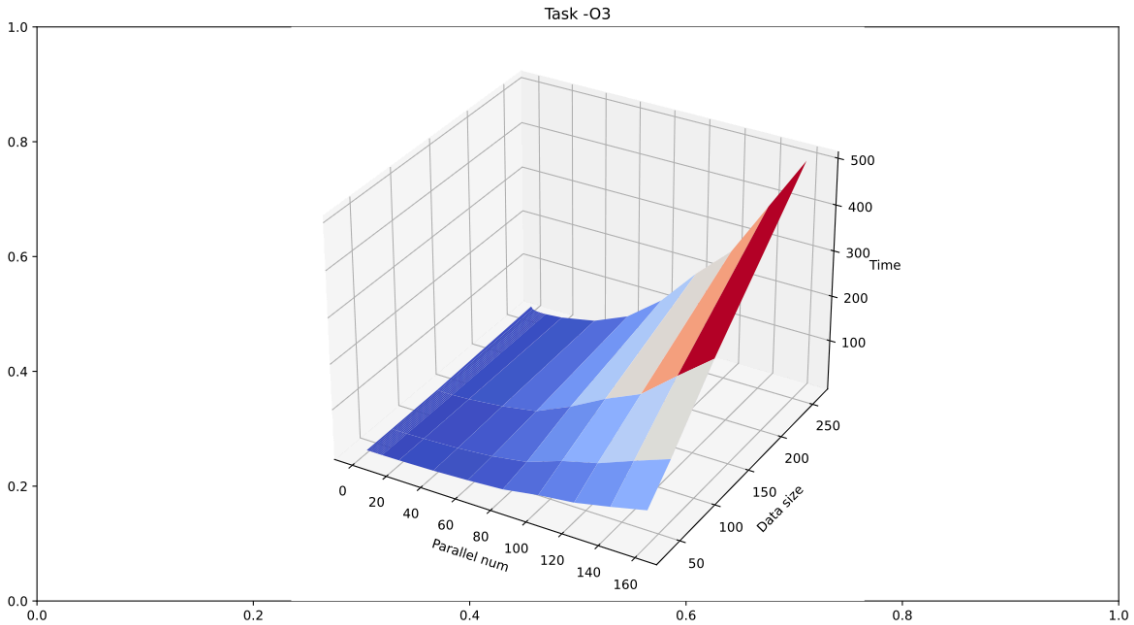
	1	2	3	4	5	6	7	8	9	10	20	40	60	80	100	120	140	160
34	0.0 34	0.0 19	0.0 16	0.0 15	0.0 15	0.0 16	0.0 17	0.0 17	0.0 19	0.0 20	0.0 27	0.0 43	0.0 64	0.1 28	0.3 30	0.4 34	0.8 96	1.3 96
66	0.2 90	0.1 51	0.1 14	0.0 93	0.0 84	0.0 79	0.0 79	0.0 75	0.0 79	0.0 78	0.0 78	0.0 97	0.1 31	0.2 09	0.4 24	0.7 08	0.9 65	1.2 29
130	2.4 01	1.2 07	0.8 72	0.7 15	0.6 35	0.5 91	0.5 39	0.5 00	0.4 89	0.4 69	0.3 82	0.3 83	0.3 96	0.5 92	0.8 79	1.0 71	1.0 96	1.7 86
258	19. 29 9	9.6 95	6.5 23	4.8 94	4.0 29	3.4 35	3.0 84	2.7 94	2.6 39	2.4 92	1.7 66	1.5 67	1.5 58	1.8 83	2.4 42	2.6 92	2.8 62	3.3 28



	1	2	3	4	5	6	7	8	9	10	20	40	60	80	100	120	140	160
34	0.0 34	0.0 23	0.0 28	0.0 35	0.0 45	0.0 57	0.0 64	0.0 76	0.0 97	0.1 14	0.5 77	2.2 64	5.2 76	10. 31 2	22. 28 4	31. 77 9	48. 46 2	65. 56 9
66	0.2 89	0.1 56	0.1 23	0.1 20	0.1 22	0.1 39	0.1 53	0.1 74	0.2 07	0.2 54	1.1 10	4.8 67	10. 70 4	28. 63 1	50. 20 9	69. 117	10 0.1 58	12 9.4 24
130	2.3 97	1.2 25	0.8 98	0.7 28	0.6 32	0.6 29	0.6 00	0.5 68	0.6 26	0.6 42	2.4 80	9.7 78	22. 88 0	50. 97 2	98. 27 3	13 4.5 82	19 6.4 08	25 4.4 30
258	19. 41 3	9.8 17	6.8 79	5.1 69	4.3 39	3.6 85	3.5 84	3.1 30	3.2 38	3.3 06	6.8 03	19. 77 5	59. 35 5	116 .86 3	19 4.5 38	25 9.6 93	38 7.9 48	51 2.3 69



	1	2	3	4	5	6	7	8	9	10	20	40	60	80	100	120	140	160
34	0.019	0.012	0.011	0.012	0.013	0.014	0.015	0.016	0.018	0.020	0.028	0.045	0.064	0.121	0.329	0.519	0.759	0.595
66	0.159	0.090	0.073	0.063	0.060	0.059	0.061	0.061	0.065	0.067	0.072	0.096	0.129	0.200	0.541	0.723	0.742	5.621
130	1.346	0.681	0.532	0.466	0.438	0.420	0.390	0.373	0.369	0.357	0.319	0.379	0.412	0.582	0.736	1.028	1.047	1.395
258	10.502	5.341	3.664	2.822	2.455	2.165	1.977	1.850	1.785	1.702	1.369	1.461	1.503	2.002	2.285	2.550	2.610	3.680



	1	2	3	4	5	6	7	8	9	10	20	40	60	80	100	120	140	160
34	0.019	0.016	0.024	0.032	0.042	0.053	0.061	0.074	0.093	0.111	0.564	2.281	5.807	10.771	23.294	32.184	48.508	66.528
66	0.156	0.093	0.081	0.091	0.101	0.120	0.134	0.158	0.191	0.233	1.077	4.306	10.589	22.493	49.466	69.266	99.225	127.837
130	1.325	0.684	0.542	0.474	0.434	0.444	0.448	0.449	0.504	0.538	2.332	10.020	22.732	56.718	97.880	132.012	195.040	255.325
258	10.399	5.337	3.827	3.092	2.632	2.322	2.315	2.134	2.235	2.302	5.378	20.227	52.808	111.689	194.331	265.966	384.677	502.925

Вывод

После реализаций программ и их тестирования можно сделать следующий вывод. Обе директивы (for, task) могут использоваться для решения этой задачи, но использование for предпочтительнее.

Для for при увеличении количества потоков наблюдается спад времени, но после какого-то количества потоков время начинает расти. Это связано с тем, что тратится больше ресурсов на создание параллельной зоны, чем на решение задачи. В зависимости от размера данных значение количества потоков, при которых время начинает увеличиваться, может быть разным.

Для task при увеличении количества потоков наблюдается спад времени, но после какого-то количества нитей время начинает резко расти и достигать существенных значений. Причиной здесь является не только та, что была изложена для for, но и проблема определения на какое количество задач нужно делать разбиение. После многочисленных тестирований можно сделать вывод, что этот параметр надо подбирать вручную в зависимости от размера данных и количество нитей. Можно было наблюдать как спад времени до большого количества нитей(~80) и резкий скачок вверх, так и отсутствие ускорения вовсе. На тестах представленных здесь, задается количество задач равное количество процессов.

Оптимизации компилятора значительно ускоряют программу, однако при большом количестве нитей(>80) не давали ожидаемый результат.

MPI

Постановка задачи

1) Для предложенного алгоритма реализовать параллельную программу с использованием технологии MPI.

2) Исследовать эффективность полученных параллельных программ на суперкомпьютере Polus.

а) Исследовать влияние различных опций оптимизации, которые поддерживаются компиляторами (-O2, -O3)

3) Исследовать масштабируемость полученной параллельной программы: построить графики зависимости времени выполнения параллельной программы от числа используемых ядер для различного объёма входных данных.

4) Определить основные причины недостаточной масштабируемости программы при максимальном числе используемых ядер/процессоров.

Код и описание программы на MPI

Использованный язык — C

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

#define  N   (2*2*2*2*2*2+2)

double   maxeps = 0.1e-7;
int itmax = 100;
int i,j,k;
int rank, size, *i_s, *i_f, buf, cub, bufN, buf0;
int *tag;
int reco1, reco2;

double eps, s;
//double A [N][N][N];
double ***F, ***S;
double **cub_send1, *cub_send2, **cub_rec1, *cub_rec2;
double t1, t2;
MPI_Request *req;
MPI_Status *stat;

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

int main(int an, char **as)
{
    MPI_Init(&an, &as);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if(rank == 0)
        t1 = MPI_Wtime();

    req = (MPI_Request *) malloc(2 * size * sizeof(MPI_Request));
    stat = (MPI_Status *) malloc(2 * size * sizeof(MPI_Status));
    tag = (int *) malloc(2 * size * sizeof(int));

    for (int root = 0; root < size; ++root) {
        tag[2 * root] = root;
        tag[2 * root + 1] = root;
    }
```

```
    i_s = (int *) malloc(size * sizeof(int));
    i_f = (int *) malloc(size * sizeof(int));

    cub = N / size;

    for (int root = 0; root < size; ++root) {
        i_s[root] = cub * root;
        if (root == size - 1) {
            i_f[root] = N - 1;
        } else {
            i_f[root] = i_s[root] + cub - 1;
        }
    }

    buf = i_f[rank] - i_s[rank] + 1;
    bufN = i_f[size - 1] - i_s[size - 1] + 1;
    buf0 = i_f[0] - i_s[0] + 1;
    reco1 = buf0 * buf * N;
    reco2 = bufN * buf * N;

    F = (double ***) malloc(N * sizeof(double **));
    for (i = 0; i < N; ++i) {
        F[i] = (double **) malloc(buf * sizeof(double *));
        for (j = 0; j < buf; ++j)
            F[i][j] = (double *) malloc(N * sizeof(double));
    }

    S = (double ***) malloc(buf * sizeof(double **));
    for  (i = 0; i < buf; ++i) {
        S[i] = (double **) malloc(N * sizeof(double **));
        for (j = 0; j < N; ++j)
            S[i][j] = (double *) malloc(N * sizeof(double));
    }

    cub_send1 = (double **) malloc((size - 1) * sizeof(double *));
    for (int root = 0; root < size - 1; ++root) {
        cub_send1[root] = (double *) malloc(reco1 * sizeof(double));
    }

    cub_send2 = (double *) malloc(reco2 * sizeof(double));

    cub_rec1 = (double **) malloc((size - 1) * sizeof(double *));
    for (int root = 0; root < size - 1; ++root) {
        cub_rec1[root] = (double *) malloc(reco1 * sizeof(double));
    }

    cub_rec2 = (double *) malloc(reco2 * sizeof(double));

    int it;

    init();

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

    verify();

    for (int root = 0; root < size - 1; ++root) {
        free(cub_send1[root]);
    }

    free(cub_send1);
    free(cub_send2);

    for (int root = 0; root < size - 1; ++root) {
        free(cub_rec1[root]);
    }

    free(cub_rec1);
    free(cub_rec2);

    for (i = 0; i < N; ++i) {
        for (j = 0; j < buf; ++j)
            free(F[i][j]);

        free(F[i]);
    }

    free(F);

    for (i = 0; i < buf; ++i) {
        for (j = 0; j < N; ++j)
            free(S[i][j]);

        free(S[i]);



    }

    free(S);

    free(req);
    free(stat);

    free(i_s);
    free(i_f);
    free(tag);
```

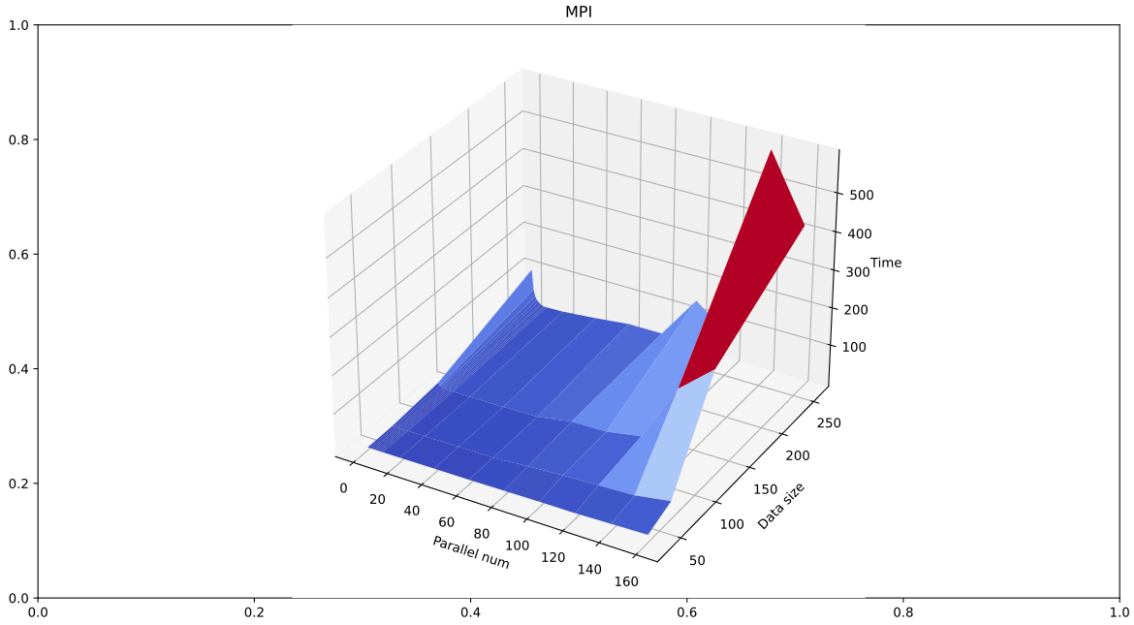
```
148
149     MPI_Finalize();
150
151     if(rank == 0) {
152         t2 = MPI_Wtime();
153         printf("%.3lf\n", t2 - t1);
154     }
155
156     return 0;
157 }
158
159
160 void init()
161 {
162     for(i=0; i<=N-1; i++)
163     for(j=i_s[rank]; j<=i_f[rank]; j++)
164     for(k=0; k<=N-1; k++)
165     {
166         if(i==0 || i==N-1 || j==0 || j==N-1 || k==0 || k==N-1)
167         F[i][j - i_s[rank]][k]= 0.;
168         else F[i][j - i_s[rank]][k]= ( 4. + i + j + k) ;
169     }
170
171     for(i=i_s[rank]; i<=i_f[rank]; i++)
172     for(j=0; j<=N-1; j++)
173     for(k=0; k<=N-1; k++)
174     {
175         if(i==0 || i==N-1 || j==0 || j==N-1 || k==0 || k==N-1)
176         S[i - i_s[rank]][j][k]= 0.;
177         else S[i - i_s[rank]][j][k]= ( 4. + i + j + k) ;
178     }
179
180 }
181
182 void relax()
183 {
184     for(i=1; i<=N-2; i++)
185     for(j=Max(i_s[rank], 1); j<=Min(i_f[rank], N-2); j++)
186     for(k=1; k<=N-2; k++)
187     {
188         F[i][j - i_s[rank]][k] = (F[i-1][j - i_s[rank]][k]+F[i+1][j - i_s[rank]][k])/2.;
189     }
190
191     for (int root = 0; root < size - 1; ++root) {
192         for (i = i_s[root]; i <= i_f[root]; ++i) {
193             for (j = i_s[rank]; j <= i_f[rank]; ++j) {
194                 for (k = 0; k < N; ++k) {
195                     cub_send1[root][(i - i_s[root]) * buf * N + (j - i_s[rank]) * N + k] = F[i][j - i_s[rank]][k];
196                 }
197             }
198         }
199     }
200
201     for (i = i_s[size - 1]; i <= i_f[size - 1]; ++i) {
202         for (j = i_s[rank]; j <= i_f[rank]; ++j) {
203             for (k = 0; k < N; ++k) {
204                 cub_send2[(i - i_s[size - 1]) * buf * N + (j - i_s[rank]) * N + k] = F[i][j - i_s[rank]][k];
205             }
206         }
207     }
208
209     for (int root = 0; root < size - 1; ++root) {
210         MPI_Isend(&cub_send1[root][0], reco1, MPI_DOUBLE, root, tag[2 * rank], MPI_COMM_WORLD, &req[2 * root]);
211     }
212
213     MPI_Isend(&cub_send2[0], reco2, MPI_DOUBLE, size - 1, tag[2 * rank], MPI_COMM_WORLD, &req[2 * (size - 1)]);
214
215     for (int root = 0; root < size - 1; ++root) {
216         MPI_Irecv(&cub_rec1[root][0], reco1, MPI_DOUBLE, root, tag[2 * root + 1], MPI_COMM_WORLD, &req[2 * root + 1]);
217     }
218
219     MPI_Irecv(&cub_rec2[0], reco2, MPI_DOUBLE, size - 1, tag[2 * (size - 1) + 1], MPI_COMM_WORLD, &req[2 * (size - 1) + 1]);
220
221     MPI_Waitall(2 * size, req, stat);
222
223     for (int root = 0; root < size - 1; ++root) {
224         for (i = i_s[rank]; i <= i_f[rank]; ++i) {
225             for (j = i_s[root]; j <= i_f[root]; ++j) {
226                 for (k = 0; k < N; ++k) {
227                     S[i - i_s[rank]][j][k] = cub_rec1[root][(i - i_s[rank]) * buf0 * N + (j - i_s[root]) * N + k];
228                 }
229             }
230         }
231     }
232
233     for (i = i_s[rank]; i <= i_f[rank]; ++i) {
234         for (j = i_s[size - 1]; j <= i_f[size - 1]; ++j) {
235             for (k = 0; k < N; ++k) {
236                 S[i - i_s[rank]][j][k] = cub_rec2[(i - i_s[rank]) * bufN * N + (j - i_s[size - 1]) * N + k];
237             }
238         }
```

```
241
242     for(i=Max(i_s[rank], 1); i<=Min(i_f[rank], N-2); i++)
243     for(j=1; j<=N-2; j++)
244     for(k=1; k<=N-2; k++)
245     {
246         S[i - i_s[rank]][j][k] =(S[i - i_s[rank]][j-1][k]+S[i - i_s[rank]][j+1][k])/2.;
247     }
248
249     double local_eps = eps;
250
251     for(i=Max(i_s[rank], 1); i<=Min(i_f[rank], N-2); i++)
252     for(j=1; j<=N-2; j++)
253     for(k=1; k<=N-2; k++)
254     {
255         double e;
256         e=S[i - i_s[rank]][j][k];
257         S[i - i_s[rank]][j][k] = (S[i - i_s[rank]][j][k-1]+S[i - i_s[rank]][j][k+1])/2.;
258         local_eps=Max(local_eps,fabs(e-S[i - i_s[rank]][j][k]));
259     }
260
261     for (int root = 0; root < size - 1; ++root) {
262         for (i = i_s[rank]; i <= i_f[rank]; ++i) {
263             for (j = i_s[root]; j <= i_f[root]; ++j) {
264                 for (k = 0; k < N; ++k) {
265                     cub_send1[root][(i - i_s[rank]) * buf0 * N + (j - i_s[root]) * N + k] = S[i - i_s[rank]][j][k];
266                 }
267             }
268         }
269     }
270
271     for (i = i_s[rank]; i <= i_f[rank]; ++i) {
272         for (j = i_s[size - 1]; j <= i_f[size - 1]; ++j) {
273             for (k = 0; k < N; ++k) {
274                 cub_send2[(i - i_s[rank]) * bufN * N + (j - i_s[size - 1]) * N + k] = S[i - i_s[rank]][j][k];
275             }
276         }
277     }
278
279     for (int root = 0; root < size - 1; ++root) {
280         MPI_Isend(&cub_send1[root][0], reco1, MPI_DOUBLE, root, tag[2 * rank], MPI_COMM_WORLD, &req[2 * root]);
281     }
282
283     MPI_Isend(&cub_send2[0], reco2, MPI_DOUBLE, size - 1, tag[2 * rank], MPI_COMM_WORLD, &req[2 * (size - 1)]);
284
285     for (int root = 0; root < size - 1; ++root) {
286         MPI_Irecv(&cub_rec1[root][0], reco1, MPI_DOUBLE, root, tag[2 * root + 1], MPI_COMM_WORLD, &req[2 * root + 1]);
287     }
288
289     MPI_Irecv(&cub_rec2[0], reco2, MPI_DOUBLE, size - 1, tag[2 * (size - 1) + 1], MPI_COMM_WORLD, &req[2 * (size - 1) + 1]);
290
291     MPI_Waitall(2 * size, &req[0], &stat[0]);
292
293     for (int root = 0; root < size - 1; ++root) {
294         for (i = i_s[root]; i <= i_f[root]; ++i) {
295             for (j = i_s[rank]; j <= i_f[rank]; ++j) {
296                 for (k = 0; k < N; ++k) {
297                     F[i][j - i_s[rank]][k] = cub_rec1[root][(i - i_s[root]) * buf * N + (j - i_s[rank]) * N + k];
298                 }
299             }
300         }
301     }
302
303     for (i = i_s[size - 1]; i <= i_f[size - 1]; ++i) {
304         for (j = i_s[rank]; j <= i_f[rank]; ++j) {
305             for (k = 0; k < N; ++k) {
306                 F[i][j - i_s[rank]][k] = cub_rec2[(i - i_s[size - 1]) * buf * N + (j - i_s[rank]) * N + k];
307             }
308         }
309     }
310
311     MPI_Allreduce(&local_eps, &eps, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
312
313 }
314
315 void verify()
316 {
317     double ss;
318
319     ss=0.;
320     for(i=i_s[rank]; i<=i_f[rank]; i++)
321     for(j=0; j<=N-1; j++)
322     for(k=0; k<=N-1; k++)
323     {
324         ss=ss+S[i - i_s[rank]][j][k]*(i+1)*(j+1)*(k+1)/(N*N*N);
325     }
326
327     MPI_Allreduce(&ss, &s, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
328
329     if(rank == 0)
330         printf("  S = %f\n",s);
331 }
```

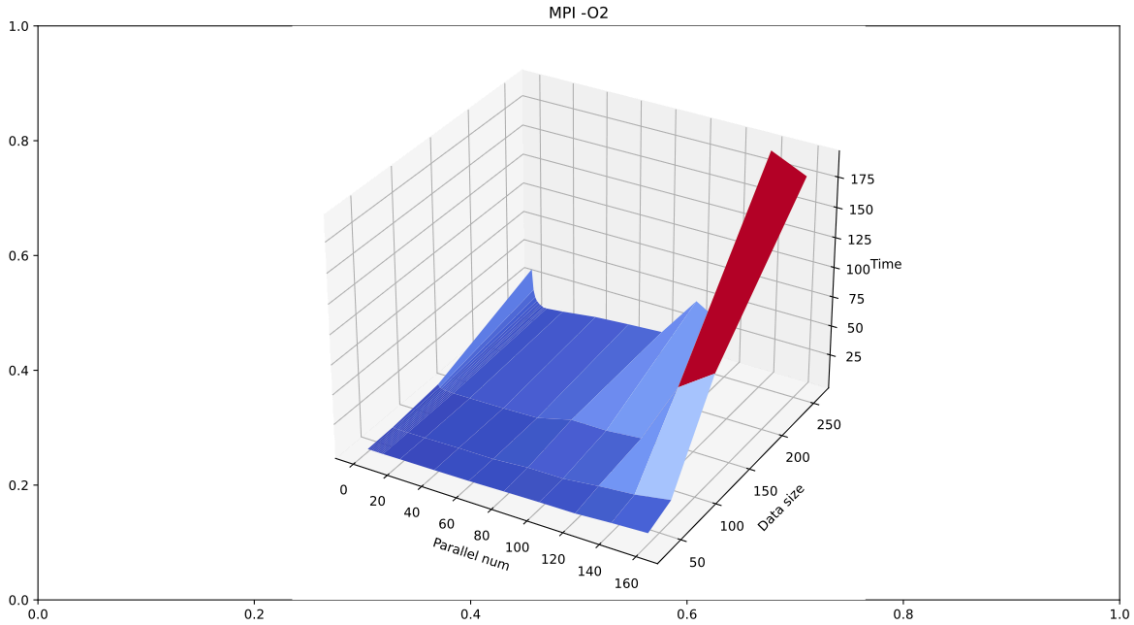
Описание

MPI_Comm_rank() выдает номер текущего процесса. MPI_Comm_size() выдает общее количество процессов. Каждый процесс создает массив i_s, i_f, которые сообщают о размере массива в каждом процессе. Задаются размеры передаваемых блоков между процессами. Создаются сами блоки, с которыми работает процесс — F, S. Далее идет их инициализация. После чего следуют вызовы функции релаксации, в которой каждый блок S и F после манипуляций с ними, ‘режется на блоки’ и передается в другие процессы с помощью MPI_Isend, и сам процесс принимает эти блоки от всех процессов. Дальше происходит ожидание всех пересылок с помощью MPI_Waitall. С помощью MPI_Allreduce процессы обмениваются своими eps. После запускается функция верификации и процессы суммируют свои s, с помощью MPI_Allreduce. Далее идет освобождение динамической памяти в каждом процессе и завершение по средствам MPI_Finalize().

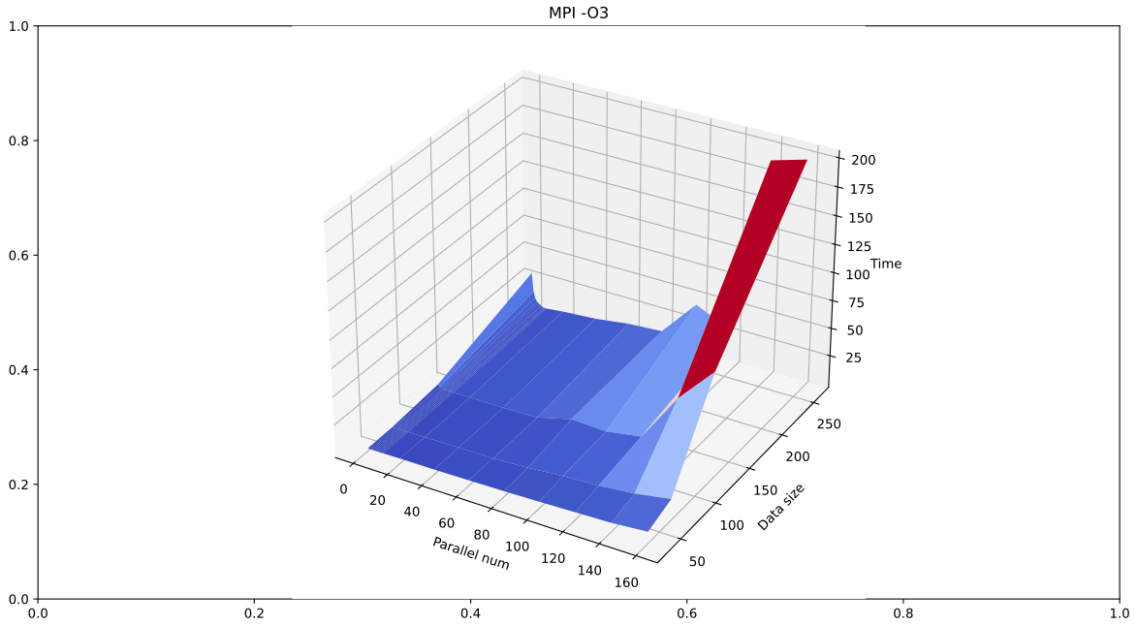
Тесты



	1	2	3	4	5	6	7	8	9	10	20	40	60	80	100	120	140	160
34	0.248	0.128	0.096	0.086	0.086	0.079	0.091	0.088	0.094	0.095	0.205	0.848	1.023	2.006	3.046	3.377	5.770	8.185
66	1.786	0.890	0.627	0.535	0.465	0.398	0.426	0.334	0.373	0.482	0.469	1.356	1.384	8.177	13.429	16.883	20.629	35.698
130	13.761	6.886	4.700	3.623	2.956	2.789	2.517	2.054	2.107	1.679	2.153	2.797	4.544	20.089	21.811	38.397	191.408	269.610
258	108.096	53.775	36.009	27.690	23.477	18.578	18.418	14.541	15.152	14.769	16.080	24.340	34.775	35.166	151.696	77.905	597.108	426.791



	1	2	3	4	5	6	7	8	9	10	20	40	60	80	100	120	140	160
34	0.080	0.044	0.036	0.043	0.036	0.051	0.043	0.052	0.046	0.045	0.220	0.433	0.748	1.485	2.043	2.326	4.195	5.554
66	0.575	0.292	0.228	0.212	0.161	0.170	0.160	0.155	0.133	0.197	0.293	0.788	0.765	2.899	3.067	6.312	8.839	13.909
130	4.549	2.314	1.568	1.311	1.069	1.021	0.917	0.815	0.817	0.716	0.849	1.145	1.967	9.443	9.384	12.062	64.214	84.678
258	36.463	18.432	12.266	9.395	8.014	6.546	6.344	5.101	5.262	5.097	6.493	10.186	12.225	13.950	49.603	31.952	192.273	178.833



	1	2	3	4	5	6	7	8	9	10	20	40	60	80	100	120	140	160
34	0.080	0.044	0.036	0.043	0.036	0.051	0.043	0.052	0.046	0.045	0.220	0.433	0.748	1.485	2.043	2.326	4.195	5.554
66	0.575	0.292	0.228	0.212	0.161	0.170	0.160	0.155	0.133	0.197	0.293	0.788	0.765	2.899	3.067	6.312	8.839	13.909
130	4.549	2.314	1.568	1.311	1.069	1.021	0.917	0.815	0.817	0.716	0.849	1.145	1.967	9.443	9.384	12.062	64.214	84.678
258	36.463	18.432	12.266	9.395	8.014	6.546	6.344	5.101	5.262	5.097	6.493	10.186	12.225	13.950	49.603	31.952	192.273	178.833

Вывод

После реализаций программ и их тестирования можно сделать следующий вывод.

Как и ранее при увеличении количества процессов наблюдается спад времени, но после 8 процессов время начинает расти. Это связано с тем, что тратится больше времени на общение между процессами, чем на решение задачи. Как можно видеть из графика: после 60 процессов происходит резкое увеличение времени выполнения.

Оптимизации компилятора значительно ускоряют программу, можно наблюдать ускорение до 3.1 раз. И даже с оптимизацией аналогичные программы на for с той же оптимизацией дадут результат лучше.

В результате тестирование можно прийти к тому, что для решения этой задачи лучше использовать OpenMP, чем MPI.