МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Практикум по курсу

"Суперкомпьютеры и параллельная обработка данных"

Разработка параллельной версии программы

ОТЧЕТ

о выполненном задании

студента 321 учебной группы факультета ВМК МГУ

Ефанова Михаила Михайловича

Москва, 2024 год

Оглавление

1 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) Реализовать параллельную версию программы с использованием технологии MPI.

1.1 Описание алгоритма

void relax(): Основная функция, которая выполняет релаксацию на трехмерной сетке. Она обновляет значения в массиве B на основе средних значений соседних элементов в массиве A.

void resid(): Вычисляет максимальную разность между элементами матриц A, B с одинаковыми индексами. И заменяет данные в массиве A на данные из массива B.

Тем самым эти две функции реализуют метод релаксации на матрице и подсчитывают изменения на каждом шаге.

void init(): Инициализирует массив A. Устанавливает граничные значения равными 0, а внутренние 4 + i + j.

void verify(): Вычисляет сумму значений в массиве A с некоторыми весами.

Эти функции занимают немного времени, поэтому мы их не распределяем.

Оригинальный код программы (1.1):

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#define  Max(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;
double eps;
double A [N][N][N],  B [N][N][N];

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

int main(int an, char **as)
{
    double start = omp_get_wtime();
    int it;
    init();
    for(it=1; it<=itmax; it++)
    {
        eps = 0.;
        relax();
        resid();
        printf( "it=%4i   eps=%f\n", it,eps);
        if (eps < maxeps) break;
    }
    verify();
    printf("%f", omp_get_wtime()-start);
    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++)
    {
        B[i][j][k]=
               (A[i-1][j][k]+A[i+1][j][k]+
                A[i][j-1][k]+A[i][j+1][k]+
                A[i][j][k-1]+A[i][j][k+1])/6.;
    }
}

void resid()
{
    for(k=1; k<=N-2; k++)
    for(j=1; j<=N-2; j++)
    for(i=1; i<=N-2; i++)
    {
        double e;
        e = fabs(A[i][j][k] - B[i][j][k]);
        A[i][j][k] = B[i][j][k];
        eps = Max(eps,e);
    }
}

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

2 Оптимизация исходной программы

Улучшено обращение к памяти, чтобы оно было более кучным для более частых попаданий в кэш, для этого изменен порядок перебора индексов. Также запись и обращения в матрицы меняются местами для того, чтобы уменьшить количество копирований.

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

#define N 66
double   maxeps = 0.1e-7;
int itmax = 100;
int tred;
int i,j,k,it;
double eps;
double A [N][N][N],  B [N][N][N];

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

int main(int an, char **as)
{
    struct timeval starttime, stoptime;
    gettimeofday(&starttime, NULL);
    init();
    for (it=1; (it<=itmax); it++)
    {
        eps = 0.;
        relax();
        resid();
        printf( "it=%4i   eps=%f\n", it, eps);
        if (eps < maxeps) break;
    }
    verify();

    gettimeofday(&stoptime, NULL);
    long sec = stoptime.tv_sec - starttime.tv_sec;
    long msec  = stoptime.tv_usec - starttime.tv_usec;
    printf("%f\n", sec + msec * 1e-6);
    return 0;
}

void init()
{
    int i, j, k;
    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[k][j][i]= 0.;
        else A[k][j][i] = (4. + i + j + k);
    }
}

void relax()
{
    int i, j, k;
    for(k=1; k<=N-2; k++)
    for(j=1; j<=N-2; j++)
    for(i=1; i<=N-2; i++)
```

```
    {
        if (it % 2 == 1)

B[k][j][i]=(A[k-1][j][i]+A[k+1][j][i]+A[k][j-1][i]+A[k][j+1][i]+A[k][j][i-1]+A[k][j][i+1])/6
.;
        else

A[k][j][i]=(B[k-1][j][i]+B[k+1][j][i]+B[k][j-1][i]+B[k][j+1][i]+B[k][j][i-1]+B[k][j][i+1])/6
.;
    }
}

void resid()
{
    int i, j, k;
    for(k=1; k<=N-2; k++)
    for(j=1; j<=N-2; j++)
    for(i=1; i<=N-2; i++)
    {
        double e;
        e = fabs(A[k][j][i] - B[k][j][i]);
        if (it % 2 == 1)
            A[k][j][i] = B[k][j][i];
        else
            B[k][j][i] = A[k][j][i];
        eps = Max(eps,e);
    }
}

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

3 Построение параллельной версии программы

В код добавлено распараллеливание с использованием OpenMP для повышения производительности программы. Так как большую часть работы программы занимает выполнение функций relax() и resid(), то распараллеливать будем их.

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

В коде добавлены директивы omp parallel for, которые позволяют распараллелить выполнение циклов. Каждый поток может обрабатывать отдельные итерации циклов одновременно, что приводит к более быстрому выполнению.

Использование private(j, i) позволяет каждой задаче иметь свои собственные копии переменных i, j, k, что предотвращает возможные конфликты при доступе к этим переменным из разных потоков.

Для того чтобы вычислять редукционную переменную eps использовалась директива reduction.

Код программы с использованием omp for (4.1):

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

#define NUM_THREADS 4
#define N 66
double   maxeps = 0.1e-7;
int itmax = 100;
int tred;
int i,j,k, it;
double eps;
double A [N][N][N],  B [N][N][N];

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

int main(int an, char **as)
{
    tred = strtol(as[1], 0, 10);

    double start = omp_get_wtime();
    init();
    for (it=1; (it<=itmax); it++)
    {
        eps = 0.;
        relax();
        resid();
        // printf( "it=%4i   eps=%f\n", it, eps);
```

```
        if (eps < maxeps) break;
    }
    verify();

    printf("%f\n", omp_get_wtime()-start);
    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[k][j][i]= 0.;
        else A[k][j][i] = (4. + i + j + k);
    }
}

void relax()
{

    #pragma omp parallel for private(i, j, k) shared(A, B) num_threads(tred)
    for(k=1; k<=N-2; k++)
    for(j=1; j<=N-2; j++)
    for(i=1; i<=N-2; i++)
    {
        if (it % 2 == 1)
            B[k][j][i]=
                (A[k-1][j][i]+A[k+1][j][i]+
                 A[k][j-1][i]+A[k][j+1][i]+
                 A[k][j][i-1]+A[k][j][i+1])/6.;
        else
            A[k][j][i]=
                (B[k-1][j][i]+B[k+1][j][i]+
                 B[k][j-1][i]+B[k][j+1][i]+
                 B[k][j][i-1]+B[k][j][i+1])/6.;  }
}

void resid()
```

```
{
    #pragma omp parallel for private(i, j, k) shared(A, B) num_threads(tred)
                                                          reduction(max: eps)
    for(k=1; k<=N-2; k++)
    for(j=1; j<=N-2; j++)
    for(i=1; i<=N-2; i++)
    {
        double e;
        e = fabs(A[k][j][i] - B[k][j][i]);
        if (it % 2 == 1)
            A[k][j][i] = B[k][j][i];
        else
            B[k][j][i] = A[k][j][i];
        eps = Max(eps,e);
    }
}

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

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

В функции relax директива omp task добавлена перед вложенными циклами, чтобы позволить каждому потоку обрабатывать отдельные элементы массивов A, B в рамках задач, которые могут выполняться параллельно.

Использование firstprivate(j, i) позволяет каждой задаче иметь свои собственные копии переменных i, j, k, что предотвращает возможные конфликты при доступе к этим переменным из разных потоков.

Директива omp taskwait используется для синхронизации выполнения. Она заставляет текущий поток ждать завершения всех задач, созданных до этой точки. Это важно для того, чтобы убедиться, что все задачи завершены перед переходом к следующим вычислениям.

Для того чтобы вычислять редукционную переменную eps добавлен массив локальных переменных local_eps. Каждый элемент массива сопоставляется одной нити. И потом последовательно вычисляется глобальное значения переменной eps.

Код программы с использованием omp task (4.2):

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

#define N 66
double   maxeps = 0.1e-7;
int itmax = 100;
int tred;
int i,j,k, it;
double eps;
double* local_eps;
double A [N][N][N],  B [N][N][N];

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

int main(int an, char **as)
{
    tred = strtol(as[1], 0, 10);

    double start = omp_get_wtime();
```

```
    local_eps = (double*)malloc(tred*sizeof(double));

    #pragma omp parallel num_threads(tred)
    {
        #pragma omp master
        {
            init();
            for(it=1; it<=itmax; it++)
            {
                eps = 0.;
                #pragma omp taskwait
                relax();
                #pragma omp taskwait
                resid();
                #pragma omp taskwait
                printf( "it=%4i   eps=%f\n", it,eps);
                if (eps < maxeps) break;
            }
        }
    }
    verify();

    free(local_eps);

    printf("%f\n", omp_get_wtime()-start);
    return 0;
}

void init()
{
    int i, j, k;
    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[k][j][i]= 0.;
        else A[k][j][i]= ( 4. + i + j + k) ;
    }
}
```

```
void relax()
{
    int i, j, k;
    for(k=1; k<=N-2; k++)
    #pragma omp task firstprivate(i, j, k)
    {
        for(j=1; j<=N-2; j++)
        for(i=1; i<=N-2; i++)
        {
            if (it % 2 == 1)

B[k][j][i]=(A[k-1][j][i]+A[k+1][j][i]+A[k][j-1][i]+A[k][j+1][i]+A[k][j][i-1]+A[k][j][i+1])/6
.;
            else

A[k][j][i]=(B[k-1][j][i]+B[k+1][j][i]+B[k][j-1][i]+B[k][j+1][i]+B[k][j][i-1]+B[k][j][i+1])/6
.;
        }
    }
}

void resid()
{
    int i, j, k;
    for(k=1; k<=N-2; k++)
    {
        #pragma omp task firstprivate(i, j, k, local_eps)
        {
            int local_index = omp_get_thread_num();
            local_eps[local_index] = 0.;
            for(j=1; j<=N-2; j++)
            for(i=1; i<=N-2; i++)
            {
                double e;
                e = fabs(A[k][j][i] - B[k][j][i]);
                if (it % 2 == 1)
                    A[k][j][i] = B[k][j][i];
                else
                    B[k][j][i] = A[k][j][i];
                local_eps[local_index] = Max(local_eps[local_index], e);
```

```
            }
        }
    }
    #pragma omp taskwait

    for (i = 0; i < tred; ++i)
        eps = Max(eps, local_eps[i]);
}

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

3.3 Вариант параллельной программы с использованием средств межпроцессного взаимодействия MPI.

Матрицы разбиваются на сегменты. Размер сегмента зависит от количества процессов (переменная size) и номера процесса (переменная rank), таким образом, чтобы каждый процесс обрабатывал примерно равное количество данных.

На границе каждого сегмента добавлены теневые слои. В теневом слое хранятся данные из самого близкого слоя к нынешнему сегменту (первый или последний слой).

Процессы пересылают друг другу данные в начале каждой итерации. Причем каждый процесс ждет данные только от своих соседей и сразу продолжает работать, не дожидаясь остальных.

Для того чтобы меньше модифицировать код исходной программы, введены 2 системы индексации в массиве: глобальная (так, как если бы все обрабатывалось в 1 процессе) и локальная (адресация внутри локальных сегментов). Для перехода между системами использовался специальный макрос (INDEX).

Теневые слои расположены таким образом, чтобы обращение к ним было “прозрачным” в программе. Это избавляет от необходимости модифицировать исходный код, но добавляет особенные процессы: нулевой и последний, у которых нет соседей, и они не могут обращаться к своим теневым слоям. Поэтому в таких ситуациях используются дополнительные сдвиги (shift_start, shift_end).

Код программы с использованием MPI (4.3):

```c
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
#include <sys/time.h>

#define Max(a,b) ((a)>(b)?(a):(b))
#define Min(a,b) ((a)<(b)?(a):(b))
#define INDEX(i, j, k) (((i + 1 - start) * N + j) * N + k)
#define N 66
double maxeps = 0.1e-7;
int itmax = 100;
int i,j,k;
double eps;
double *A, *B;
int rank, size, segmentsize;
int start, end;

void init();
void relax();
void resid();
void verify();
void synchronize_data();

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

    struct timeval starttime, stoptime;
```

```
    if (rank == 0)
    {
        gettimeofday(&starttime, NULL);
    }

    segmentsize = N % size > rank ? N / size + 1 : N / size;
    A = (double*)malloc(N * N * (segmentsize + 2) * sizeof(double));
    B = (double*)malloc(N * N * (segmentsize + 2) * sizeof(double));

    start = rank * (N / size) + Min(rank, N % size);
    end = start + segmentsize - 1;

    init();

    int it;
    for(it=1; it<=itmax; it++)
    {
        synchronize_data();
        eps = 0.;
        relax();
        resid();

        if (rank == 0)
        {
            printf( "it=%d   eps=%f\n", it,eps);
        }

        if (eps < maxeps)
        {
            break;
        }
    }

    verify();

    if (rank == 0)
    {
        gettimeofday(&stoptime, NULL);
        long sec = stoptime.tv_sec - starttime.tv_sec;
        long msec  = stoptime.tv_usec - starttime.tv_usec;
```

```
        printf("%f\n", sec + msec * 1e-6);
    }

    free(A);
    free(B);

    MPI_Finalize();
    return 0;
}

void init()
{
    for(i = start; i <= end; 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[INDEX(i, j, k)] = 0.;
        }
        else
        {
            A[INDEX(i, j, k)] = (4. + i + j + k);
        }
    }
}

void relax()
{
    int shift_start = rank == 0 ? 1 : 0;
    int shift_end = rank == size - 1 ? -1 : 0;
    for(i = start + shift_start; i <= end + shift_end; i++)
    for(j = 1; j <= N - 2; j++)
    for(k = 1; k <= N - 2; k++)
    {
        B[INDEX(i, j, k)] =
           (A[INDEX(i - 1, j, k)] + A[INDEX(i + 1, j, k)] +
            A[INDEX(i, j - 1, k)] + A[INDEX(i, j + 1, k)] +
            A[INDEX(i, j, k - 1)] + A[INDEX(i, j, k + 1)]) / 6.;
    }
```

```
}

void resid()
{
    double tmp = 0.;
    double e;
    for(i = start; i <= end; i++)
    for(j = 1; j <= N - 2; j++)
    for(k = 1; k <= N - 2; k++)
    {
        e = fabs(A[INDEX(i, j, k)] - B[INDEX(i, j, k)]);
        A[INDEX(i, j, k)] = B[INDEX(i, j, k)];
        tmp = Max(tmp, e);
    }
    MPI_Allreduce(&tmp, &eps, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
}

void verify()
{
    double global_s;
    double s = 0.;
    for(i = start; i <= end; i++)
    for(j = 0; j <= N - 1; j++)
    for(k = 0; k <= N - 1; k++)
    {
        s += A[INDEX(i, j, k)] * (i + 1) * (j + 1) * (k + 1) / (N * N * N);
    }

    MPI_Reduce(&s, &global_s, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0)
    {
        printf("  S = %f\n",global_s);
    }
}

void synchronize_data(){
    MPI_Request request;
    MPI_Status status;
    if (rank != 0)
    {
```

```
        MPI_Isend(A + N * N, N * N, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, &request);
    }
    if (rank != size - 1)
    {
        MPI_Isend(A + segmentsize * N * N, N * N, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD, &request);
    }
    if (rank != 0)
    {
        MPI_Recv(A , N * N, MPI_DOUBLE, rank - 1 , 0 , MPI_COMM_WORLD , &status);
    }
    if (rank != size - 1)
    {
        MPI_Recv(A + (segmentsize + 1) * N * N, N * N, MPI_DOUBLE, rank + 1 , 0 ,
MPI_COMM_WORLD , &status);
    }
}
```

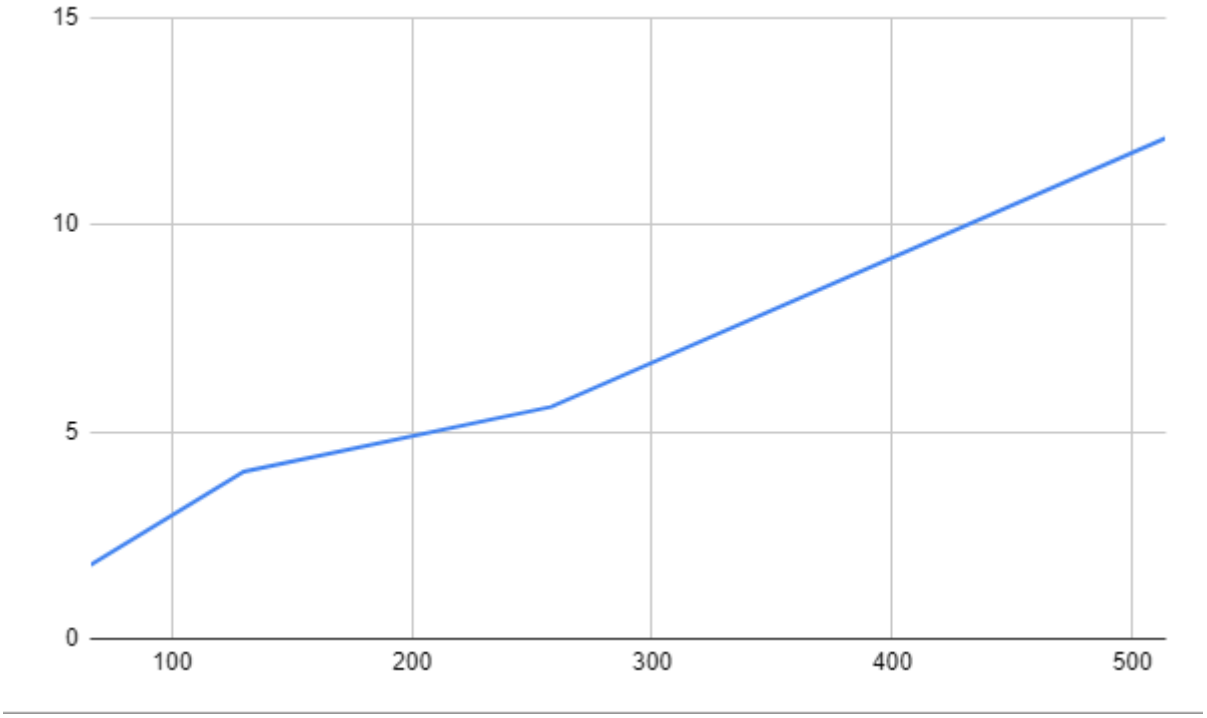
4 Результаты измерений времени выполнения

4.1 Вариант оптимизированной программы, без использования Omp.

Ниже приведены результаты измерений времени работы оригинальной программы (t_{origin}) и оптимизированной ($t_{optimized}$), а также их сравнение.

	origin	optimized	origin / optimized
66	0.609486	0.343138	1.776212486
130	11.365625	2.810645	4.043778207
258	132.228107	23.58569	5.606285294
514	2247.441342	185.679825	12.10385319

Время работы программы (1.1) - origin, (2.1) - optimized. Вертикальная ось - размер данных.



$\frac{t_{origin}}{t_{optimized}}$. Горизонтальная ось - размер данных.

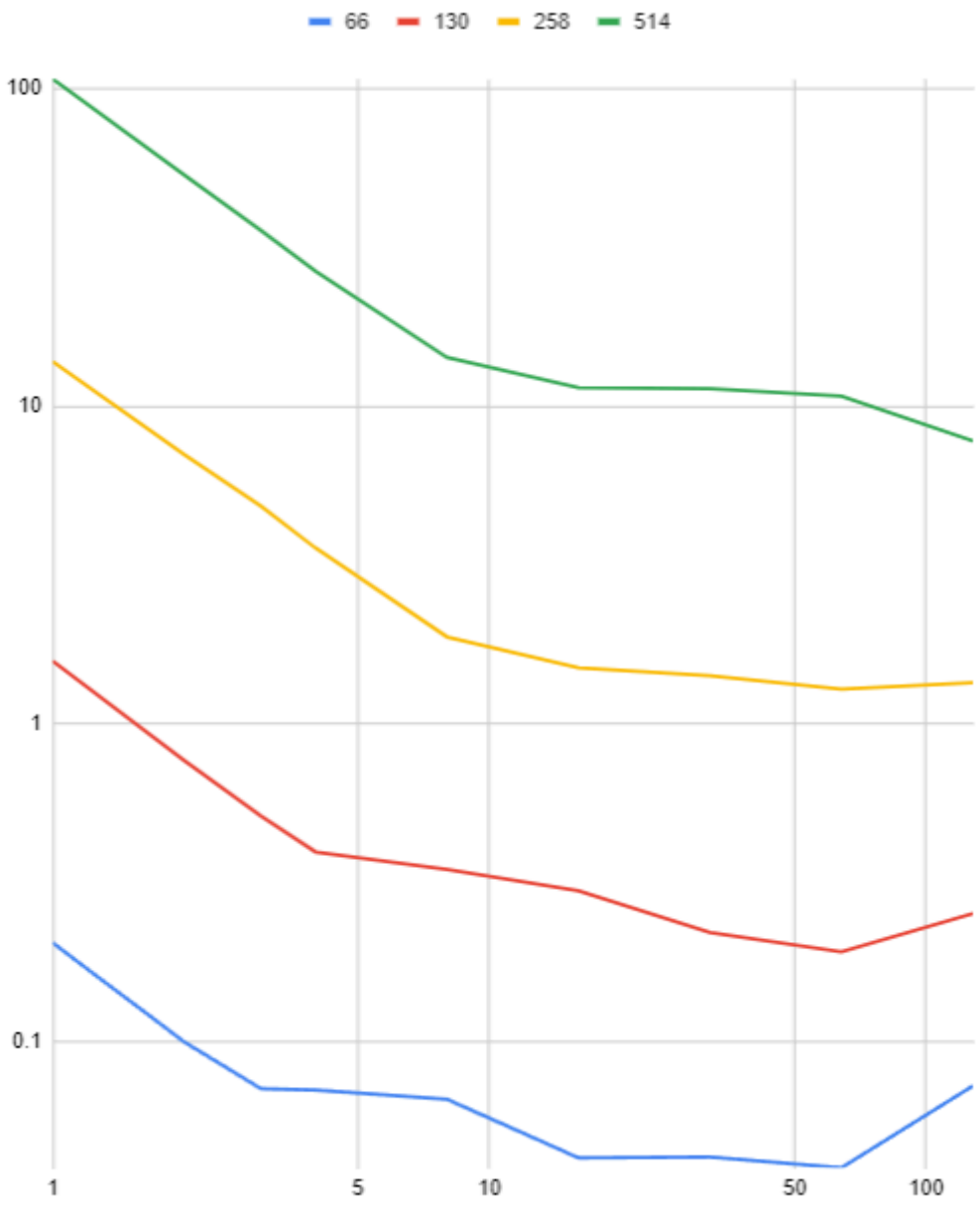
Из графика видно, что написания программы с учётом работы механизма кэш памяти дает существенное ускорение. Это говорит о том, что перед тем, как пробовать распределять вычисления, стоит провести рефакторинг кода. Полученный результат хорош еще и тем, что программа не использует никаких дополнительных ресурсов, в отличие от распределенных вычислений.

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

Для общей информации приведены измерения времени работы программы.

	66	130	258	514
1	0.205435	1.582098	13.84427	107.339183
2	0.100224	0.770834	7.070163	53.601417
3	0.071382	0.514742	4.859891	35.776563
4	0.07071	0.396752	3.58744	26.687327
8	0.066131	0.34926	1.884672	14.27516
16	0.043252	0.299714	1.505719	11.463017
32	0.043517	0.221455	1.42237	11.393325
64	0.040261	0.192498	1.290662	10.79261
128	0.072867	0.25392	1.351611	7.796548

Время работы программы (4.1) на компьютере “Полюс”. Горизонтальная ось - размер данных, вертикальная - количество нитей.



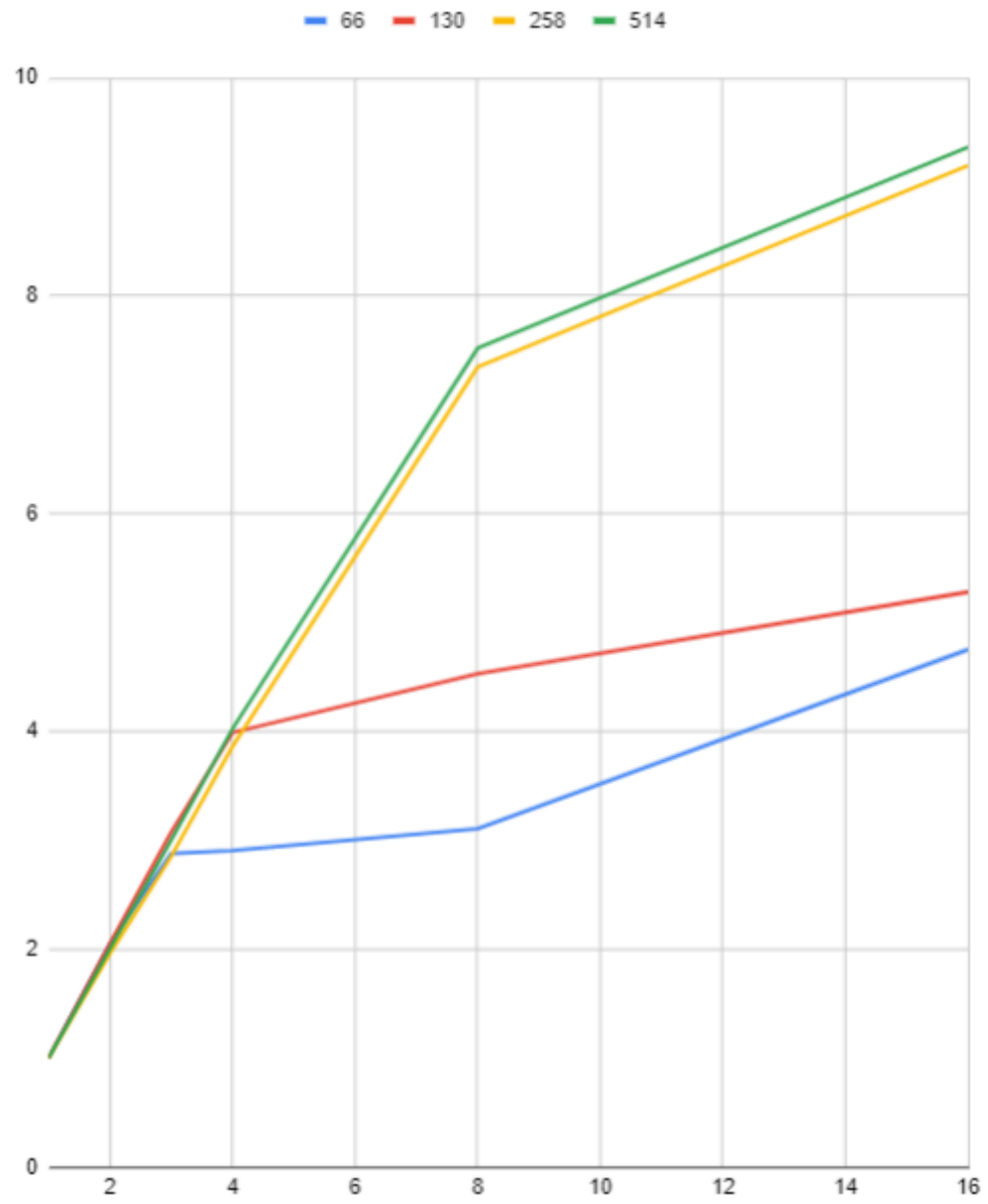
Время работы программы (4.2) на компьютере “Полюс”. Горизонтальная ось - количество нитей, вертикальная - время работы. Маркировки для линий - размер данных.

Из таблицы и графика выше видно, что при большом объеме данных, использование параллельной обработки дает заметное ускорение. Однако на малом объеме данных это может не только не ускорить работу, но даже замедлить ее, причем конечное время работы будет даже больше чем без оптимизации.

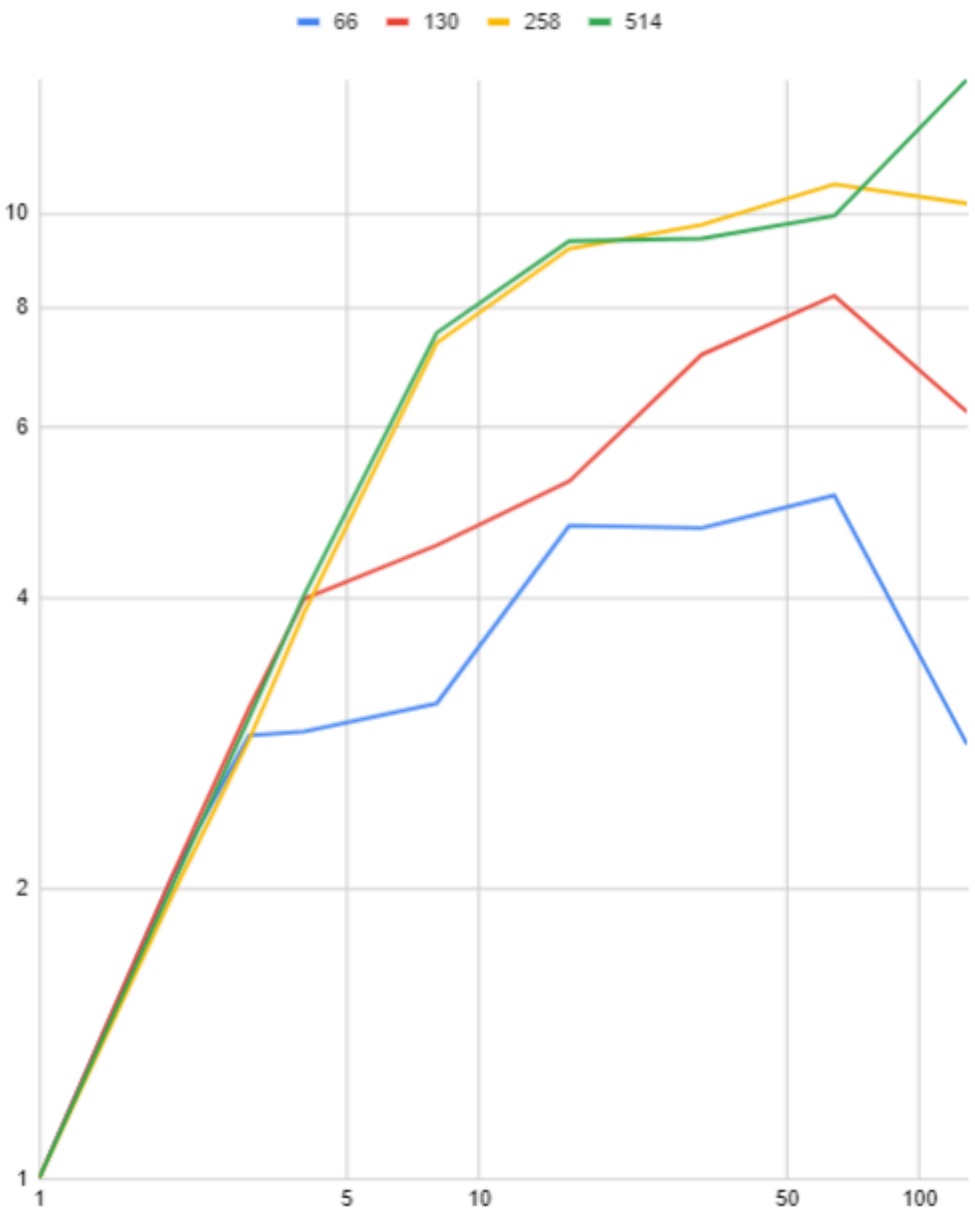
Для оценки эффективности масштабирования приведены отношения времени работы программы без использования дополнительных ресурсов (t_{origin}) к времени с использованием доп. ресурсов $(t_{parallel})$.

	66	130	258	514
1	1	1	1	1
2	2.049758541	2.052449684	1.958126001	2.002543757
3	2.877966434	3.073574723	2.848679116	3.000265369
4	2.905317494	3.987624511	3.859094508	4.022103188
8	3.106485612	4.529857413	7.345718512	7.519298067
16	4.749722556	5.278692353	9.19445793	9.363955667
32	4.720798768	7.144106026	9.733240999	9.421234188
64	5.102580661	8.218776299	10.72648765	9.945618622
128	2.819314642	6.230694707	10.24279175	13.76752673

$\frac{t_{origin}}{t_{parallel}}$. Горизонтальная ось - размер данных, вертикальная - количество нитей.



$\frac{t_{origin}}{t_{parallel}}$. Горизонтальная ось - количество нитей, вертикальная - время работы. Маркировки для линий - размер данных.



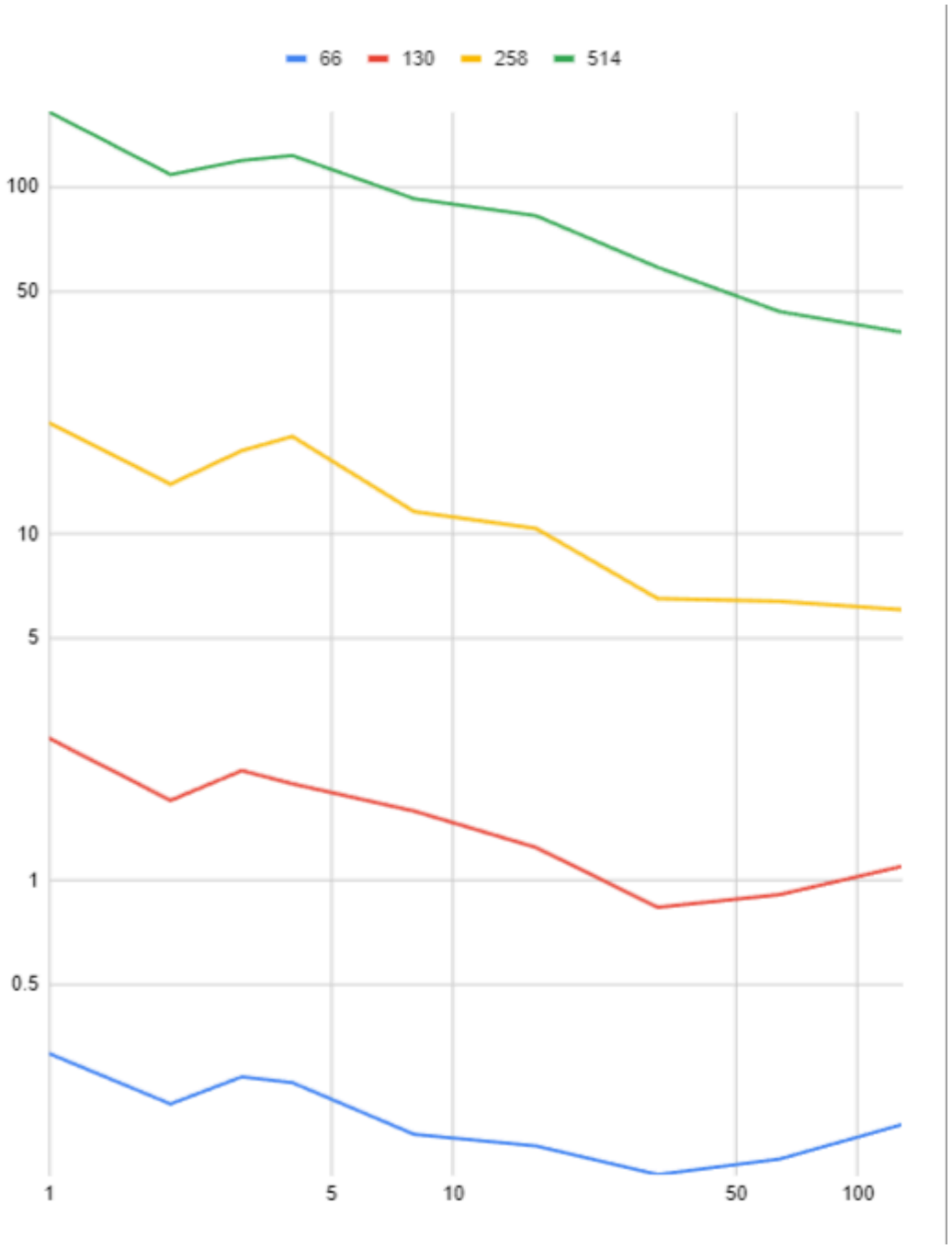
$\frac{t_{origin}}{t_{parallel}}$. Горизонтальная ось - количество нитей, вертикальная - время работы. Маркировки для линий - размер данных. Количество нитей 1-16 для лучшей наглядности.

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

Для общей информации приведены измерения времени работы программы.

	66	130	258	514
1	0.316679	2.573447	20.904236	165.1658
2	0.22629	1.701213	13.90625	108.775922
3	0.271045	2.074033	17.378148	119.505173
4	0.260713	1.899913	19.092462	123.460826
8	0.184937	1.584412	11.587892	92.650689
16	0.171194	1.244459	10.361268	82.756108
32	0.141536	0.83606	6.50116	58.73047
64	0.156894	0.909539	6.385389	43.777646
128	0.197038	1.096503	6.039909	38.17689

Время работы программы (4.2) на компьютере “Полюс”. Горизонтальная ось - размер данных, вертикальная - количество нитей.



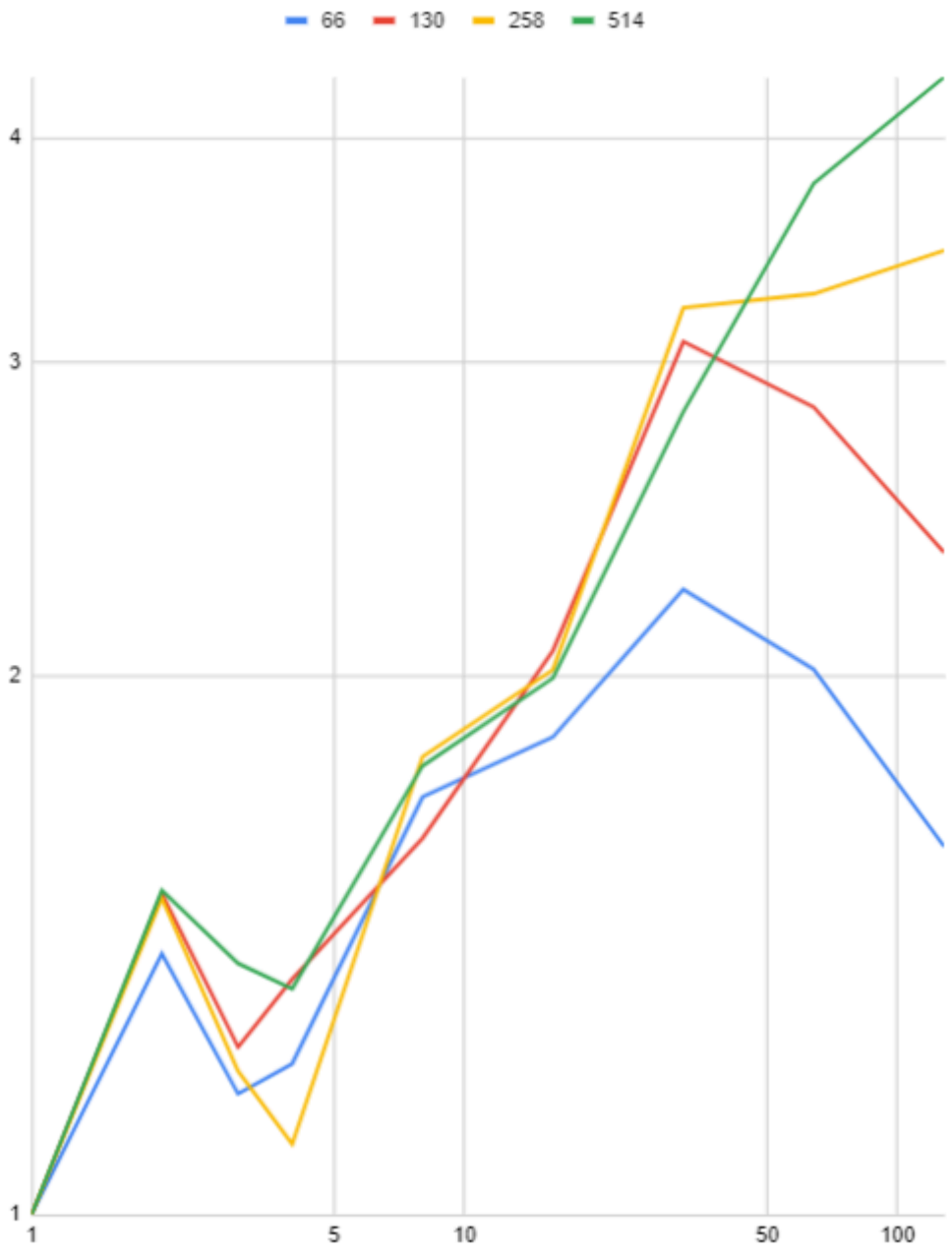
Время работы программы (4.2) на компьютере “Полюс”. Горизонтальная ось - количество нитей, вертикальная - время работы. Маркировки для линий - размер данных.

Из таблицы и графика выше видно, что при большом объеме данных, использование параллельной обработки дает заметное ускорение. Однако на малом объеме данных это может не только не ускорить работу, но даже замедлить ее, причем конечное время работы будет даже больше чем без оптимизации.

Для оценки эффективности масштабирования приведены отношения времени работы программы без использования дополнительных ресурсов (t_{origin}) к времени с использованием доп. ресурсов ($t_{parallel}$).

	66	130	258	514
1	1	1	1	1
2	1.399438773	1.512712988	1.50322596	1.518404045
3	1.168363187	1.240793661	1.202903555	1.382080757
4	1.214665168	1.354507812	1.094894729	1.337799247
8	1.712361507	1.62422842	1.803972284	1.782672118
16	1.849825344	2.067924295	2.017536464	1.995814013
32	2.23744489	3.078064971	3.215462471	2.812267636
64	2.018426454	2.82939709	3.273760769	3.772834199
128	1.607197596	2.346958467	3.46101837	4.326329358

$\frac{t_{origin}}{t_{parallel}}$. Горизонтальная ось - размер данных, вертикальная - количество нитей.



$\frac{t_{origin}}{t_{parallel}}$. Горизонтальная ось - количество нитей, вертикальная - время работы. Маркировки для линий - размер данных.

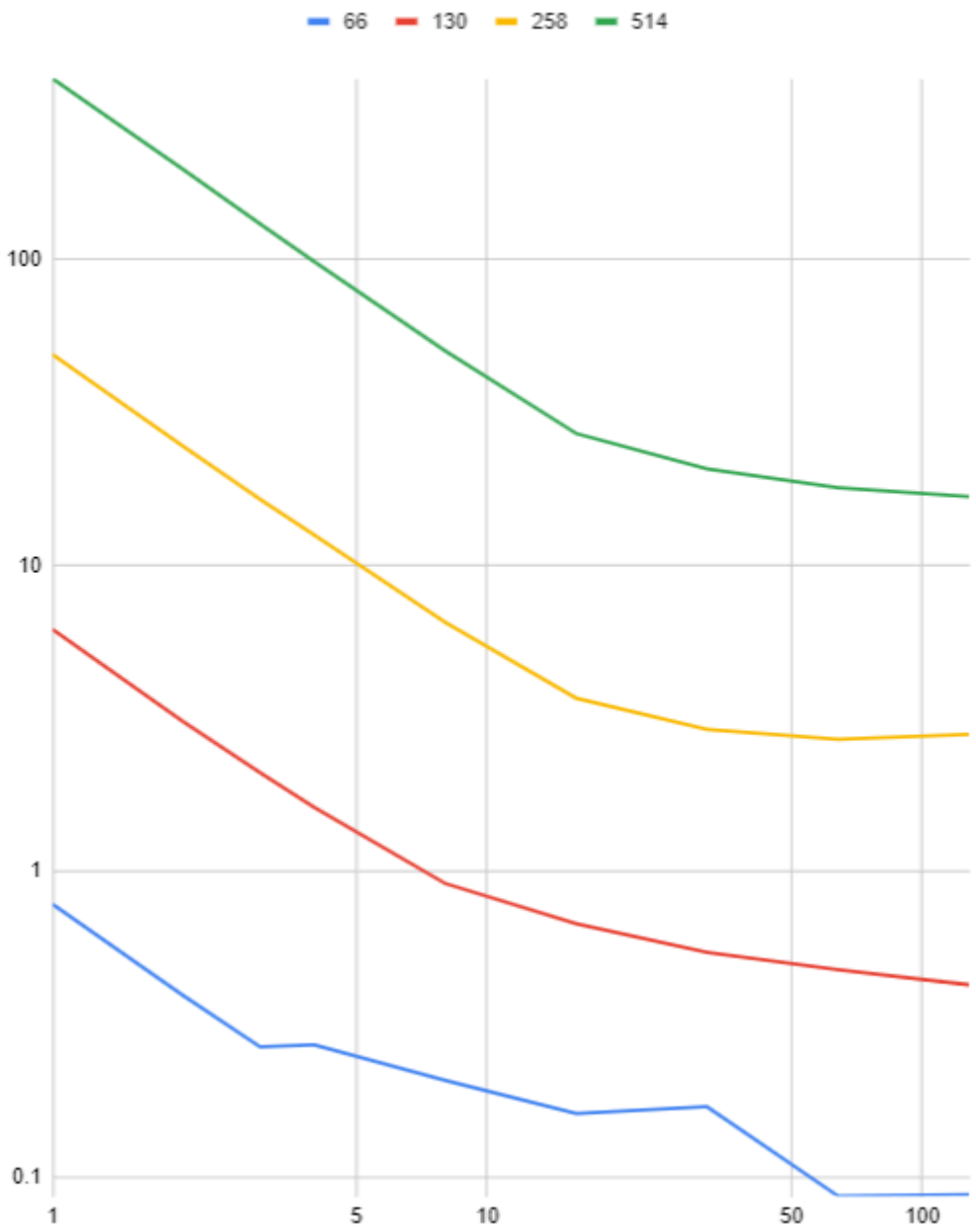
Из графиков выше более наглядно видно ускорение, полученное с использованием параллельной обработки данных. На небольшом количестве потоков, заметен быстрый линейный рост скорости работы. Однако с ростом количества потоков скорость перестает расти, и даже заметно падает.

4.4 Вариант параллельной программы с использованием средств межпроцессорного взаимодействия MPI.

Для общей информации приведены измерения времени работы программы.

	66	130	258	514
1	0.782457	6.177382	48.998873	390.337232
2	0.39253	3.089094	24.513171	196.774424
3	0.26712	2.105002	16.443418	130.631396
4	0.271187	1.623095	12.597678	98.529829
8	0.207704	0.914721	6.5229	50.31351
16	0.161671	0.675741	3.686128	26.981427
32	0.170422	0.543483	2.910644	20.650475
64	0.086897	0.477966	2.703746	17.952866
128	0.088012	0.426457	2.804909	16.818195

Время работы программы (4.3) на компьютере “Полюс”. Горизонтальная ось - размер данных, вертикальная - количество процессов.

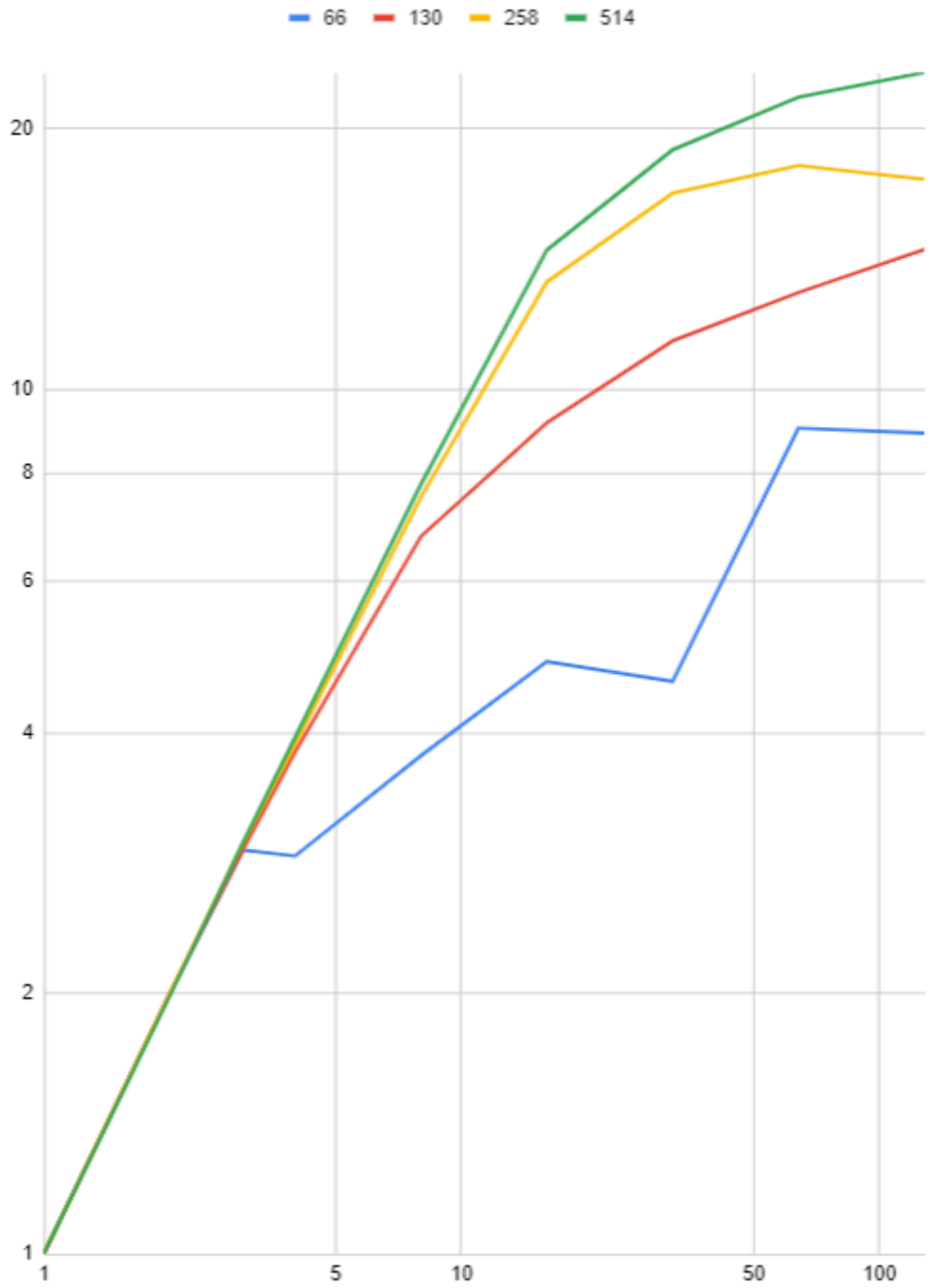


Время работы программы (4.3) на компьютере “Полюс”. Горизонтальная ось - количество процессов, вертикальная - время работы. Маркировки для линий - размер данных.

Для оценки эффективности масштабирования приведены отношения времени работы программы без использования дополнительных ресурсов (t_{origin}) к времени с использованием доп. ресурсов ($t_{parallel}$).

	66	130	258	514
1	1	1	1	1
2	1.99336866	1.999739082	1.998879419	1.983678692
3	2.929234052	2.93462049	2.979847195	2.988081303
4	2.885304237	3.805927564	3.889516227	3.961614832
8	3.767173478	6.75329636	7.511823422	7.758099803
16	4.839810479	9.141641546	13.29277578	14.46688613
32	4.591291031	11.36628377	16.83437514	18.9020946
64	9.004419025	12.92431261	18.12258733	21.74233529
128	8.890344498	14.48535726	17.46897065	23.20922263

$\frac{t_{origin}}{t_{parallel}}$. Горизонтальная ось - размер данных, вертикальная - количество процессов.



$\frac{t_{origin}}{t_{parallel}}$. Горизонтальная ось - количество процессов, вертикальная - время работы. Маркировки для линий - размер данных.

4 Анализ результатов

Были написаны два варианта параллельной программы на языке Си с помощью библиотеки <omp.h>. Программы были запущены на вычислительном комплексе «Полюс». По результатам видно, что количество ресурсов для работы программы нужно подбирать исходя из объема данных,

которые нужно обработать. Так, например, слишком “маленькая” задача будет работать еще дольше, при попытке распределить ее, так как создаются дополнительные расходы на выделение ресурсов. Также, перед использованием механизмов параллельной обработки данных, необходимо удостовериться, что программа использует максимум возможностей и в последовательной версии проектирования.

Также была написана программа на языке Си с использование библиотеки <mpi.h>. По результатам видно, что при правильной организации обмена данных между процессами, объем прикладных расходов растет очень медленно.

5 Выводы

Параллельные вычисления позволяют значительно увеличить производительность вычислений на многопоточных системах, однако необходимо выбирать оптимальное количество потоков для достижения наилучших результатов.

Основными причинами недостаточной масштабируемости программы при максимальном числе используемых ядер/процессоров могут являться недостаточно эффективная синхронизация или не лучший выбор распределения нагрузки и ресурсов.