МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Практикум по курсу

"Суперкомпьютеры и параллельная обработка данных"

Разработка параллельной версии программы Sor_3D с использованием технологии MPI.

ОТЧЕТ

о выполненном задании

студента 324 учебной группы факультета ВМК МГУ

Яндутова Алексея Владимровича

Москва, 2019 г.

Оглавление

1 Постановка задачи

Дана последовательная программа Sor_3D, реализующая алгоритм релаксации Якоби. Распараллеливание осуществляется с помощью анализа последовательной программы, аналогично анализу распараллеливающего компилятора, поэтому не предполагается знания указанного алгоритма.

Требуется:

1. Реализовать параллельную версию программы для задачи Sor_3D с использованием технологии MPI.
2. Исследовать масштабируемость полученных программ и построить графики зависимости времени выполнения программ от числа используемых ядер и объёма входных данных.

2 Реализация

2.1 Исходный код программы

Функции исходной программы:

- `init()` инициализирует трехмерную матрицу
- `verify()` высчитывает контрольную сумму итоговой матрицы
- `relax()` выполняет функционал алгоритма релаксации Якоби

Ниже приведен код функции `relax()`, выполняющей основной функционал алгоритма и являющейся узким местом программы.

```
void relax()
{

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

Этот алгоритм является итеративным, на каждой итерации вычисляет элемент матрицы как среднее ее 6 соседей. Сложность алгоритма O(n^3). Цикл имеет зависимость по даннным.

2.2 Код параллельной программы

Ниже приведен код параллельной программы(также код программы приложен к отчету)

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"

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

#define    N   130
double   maxeps = 0.1e-7;
int itmax = 100;
int i,j,k;
double eps;

double (*A) [N][N];
double (*B) [N][N];
double (*C) [N][N];

void relax(int nl, int rank, int ranks);
void init(int nl, int rank, int ranks);
void verify(int nl, int rank, int ranks);

int main(int an, char **as)
{
    double total_time = 0;
    int ranks, rank;
    MPI_Init(&an, &as);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &ranks);
    MPI_Barrier(MPI_COMM_WORLD);

    int num_layers = N / ranks;
    A = malloc(num_layers * N * N * sizeof(A[0][0][0]));
    if (ranks > 1) {
        B = malloc(N * N * sizeof(B[0][0][0]));
        C = malloc(N * N * sizeof(C[0][0][0]));
    }
    init(num_layers, rank, ranks);
    double time = MPI_Wtime();
    for(int it=1; it<=itmax; it++)
    {
        eps = 0;
        relax(num_layers, rank, ranks);
       if (rank == 0) {
            printf( "it=%4i   eps=%f\n", it,eps);
            if (eps < maxeps)
                break;
        }
    }
```

```
void relax(int nl, int rank, int ranks)
{
    double loc = 0.;
    for(i=0; i<nl; ++i) {
        if ((i == 0 && rank == 0) || (i == nl - 1 && rank == ranks - 1)) {
            continue;
        }
        if (ranks > 1 && (rank != 1 || i != 0)) {
            if (!rank) {
                MPI_Send(&A[i][0][0], N * N, MPI_DOUBLE, ranks - 1, 0, MPI_COMM_WORLD);
            } else {
                MPI_Send(&A[i][0][0], N * N, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD);
            }
        }
        if (rank == ranks - 2 && i == nl - 1) {
            for (j = 0; j < N; ++j) {
                for (k = 0; k < N; ++k) {
                    C[0][j][k] = 0;
                }
            }
        } else if (ranks > 1) {
            MPI_Recv(&C[0][0][0], N * N, MPI_DOUBLE, (rank + 1) % ranks, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        for(j=1; j<N-1; ++j) {
            if (ranks > 1 && (rank != 1 || i != 0)) {
                if (!rank) {
                    MPI_Recv(&B[0][j][0], N, MPI_DOUBLE, ranks - 1, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                } else {
                    MPI_Recv(&B[0][j][0], N, MPI_DOUBLE, rank - 1, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                }
            }
            for(k=1; k<N-1; ++k) {
                double e = A[i][j][k];
                if (ranks == 1) {
                    A[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.;
                } else {
                    A[i][j][k] = (B[0][j][k] + C[0][j][k] + A[i][j - 1][k] + A[i][j +
1][k] + A[i][j][k - 1] + A[i][j][k + 1]) / 6.;
                }
                    loc=Max(loc, fabs(e - A[i][j][k]));
            }
            if (ranks > 1 && (rank != ranks - 2 || i != nl - 1)) {
                MPI_Send(&A[i][j][0], N, MPI_DOUBLE, (rank + 1) % ranks, 1,
MPI_COMM_WORLD);
            }
        }
    }
    MPI_Reduce(&loc, &eps, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
}
```

```
void init(int nl, int rank, int ranks)
{
    for(i = 0; i < nl; ++i) {
       for(j = 0; j < N; ++j) {
            for(k = 0; k < N; ++k) {
                if(j==0 || j==N-1 || k==0 || k==N-1 ||(rank + ranks * i==0) ||
(rank + ranks * i==N-1)) {
                  A[i][j][k]= 0;
                } else {
                   A[i][j][k]= 4 + (rank + ranks * i) + j + k;
                }
            }
         }
    }
}



void verify(int nl, int rank, int ranks)
{
    double s = 0.;
    double loc_sum = 0.;
    for(i=0; i<=nl-1; i++) {
       for(j=0; j<=N-1; j++) {
            for(k=0; k<=N-1; k++) {
                loc_sum = loc_sum +
A[i][j][k]*(rank+ranks*i+1)*(j+1)*(k+1)/(N*N*N);
            }
         }
    }
    MPI_Reduce(&loc_sum, &s, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0) {
       printf("  S = %f\n",s);
    }
}
```

2.3 Описание параллельного алгоритма

Изначально был рассмотрен вариант распараллеливания преобразованного цикла, проходящегося по гиперплоскостям, перпендикулярным вектору (1, 1, 1), который был использован в параллельной программе с использованием OpenMP. Однако из-за неудобств при разбиении матрицы на слои между процессами и прохождению цикла по таким гиперплоскостям, не получилось добиться правильной и более быстрой работы распараллеленного алгоритма по сравнению с последовательным. Поэтому было рассмотрено немного другое решение данной задачи более соответствующего парадигме MPI:

1. Трехмерная матрица была распределена между процессами таким образом, что каждый процесс имеет `N / ranks` слоев, то есть массивов размера `N*N`. Память каждого из процессов имеют следующую структуру: для процесса rank слой с номером `i` является слоем `rank + ranks * i` в изначальном глобальном массиве размера `N*N*N`, то есть слои, обрабатываемые разными процессами перемежаются друг с другом.

   Например, если дана матрица размером 8*8*8 и 4 процесса, то слои сетки будут распределены следующим образом(rank – номер процесса, i – номер слоя в памяти соответствующего процесса)

rank = 0, i = 0	
rank = 1, i = 0	
rank = 2, i = 0	
rank = 3, i = 0	
rank = 0, i = 1	
rank = 1, i = 1	
rank = 2, i= 1	
rank = 3, i= 1	

2. Каждый из процессов rank(кроме процесса, который обрабатывает предпоследний слой, то есть последний негранничный слой, так как его теневой слой заполняется нулями) перед обработкой слоя i получает целый необработанный слой соответствующего номера от процесса (rank + 1) % ranks

3. Перед обработкой строки j слоя i процесс rank получает соответствующую строку соседнего слоя от процесса (rank – 1) (ranks – 1 для rank == 0), опять же кроме процесса, обрабатывающей первый негранничный слой, так как его теневой слой заполнен нулями.

4. После вычисления строки j слоя i процесс rank отправляет ее процессу (rank + 1) % ranks, чтобы тот мог вычислить соответствующую строку своего слоя.

При этом с помощью функции verify() проверено, что сохраняется правильность выполнения программы с возможностью выполнять данную программу параллельно.

2.4 Использованные средства MPI

- MPI_Init(&an, &as) – инициализация параллельного выполнения программы;
- MPI_Comm_rank(MPI_COMM_WORLD, &rank) — получения номера процесса rank в коммуникаторе MPI_COMM_WORLD, содержащем все процессы;
- MPI_Comm_size(MPI_COMM_WORLD, &ranks) — получение общего числа процессов ranks в коммуникаторе MPI_COMM_WORLD;
- MPI_Finalize() — завершение параллельного выполнения;
- MPI_Barrier(MPI_COMM_WORLD) — точка синхронизации всех процессов;
- MPI_Send(), MPI_Recv () — функции посылки/принятия данных, используемые в данной программе для обмена между процессами теневыми зонами;
- MPI_Reduce(&loc_eps, &eps, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD) — операция редукции переменной eps по максимуму;
- MPI_Wtime() — получение астрономического времени в секундах, используется для измерения времени выполнения программы;

3 Результаты замеров времени выполнения

Ниже приведены результаты замеров времени программы на суперкомпьютерах Bluegene и Polus: непосредственно в табличной форме и наглядно на 3D-графиках.

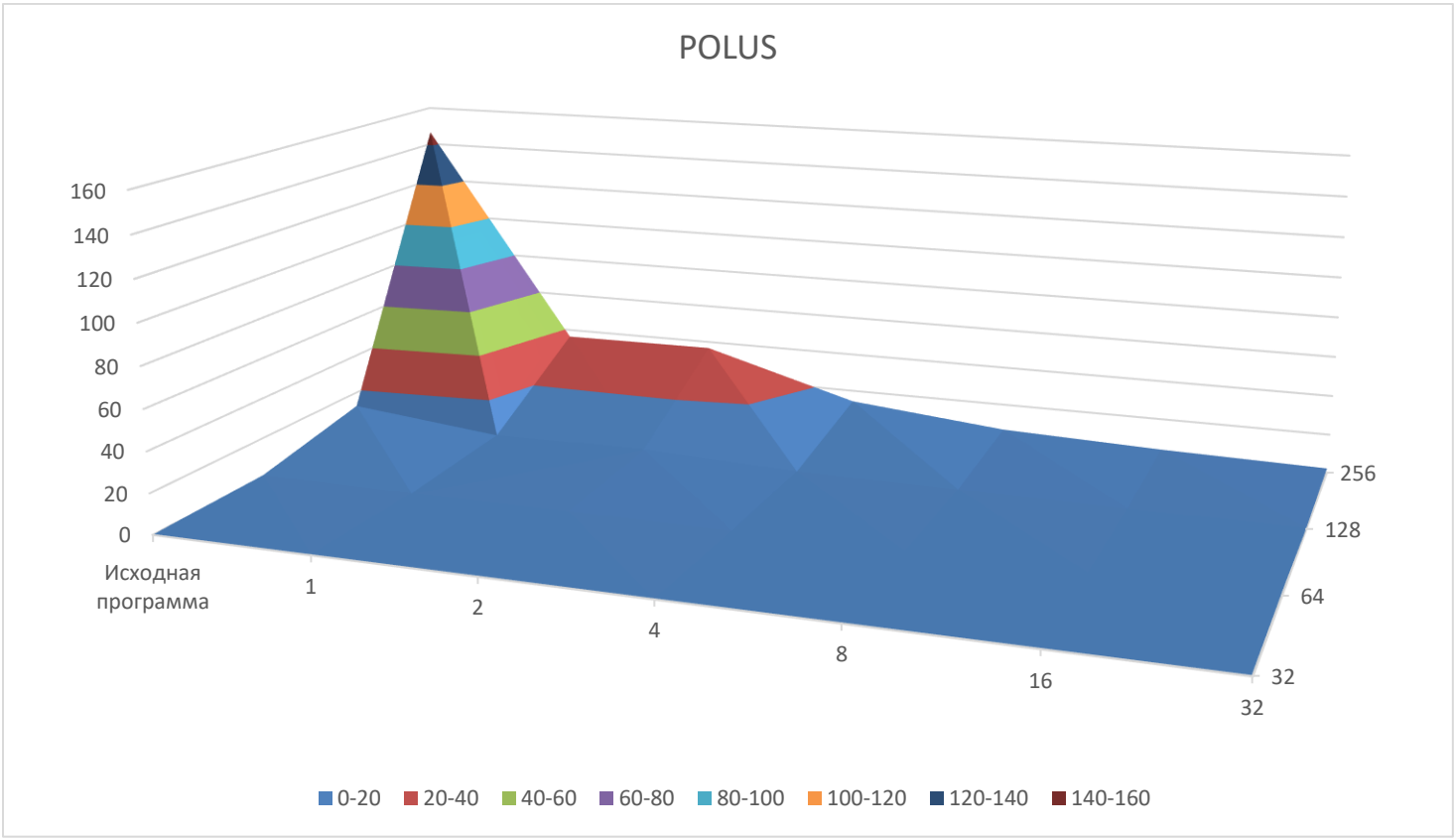
Программа была запущена на Polus на 1, 2, 4, 8, 16, 32, процессах на различных размерах матрицы.

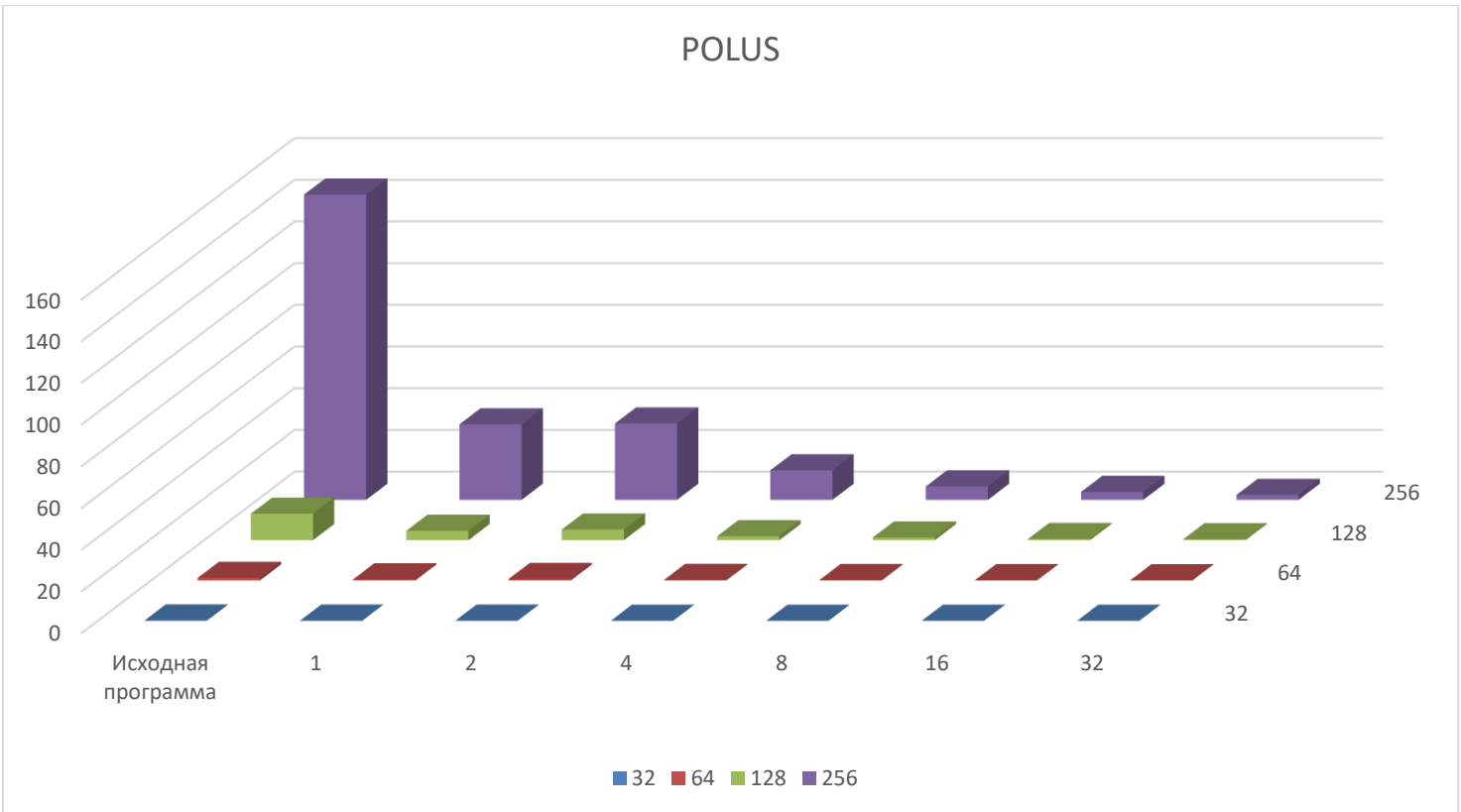
Каждая запуск был произведен 3 раза, ниже приведена таблица с усредненными значениями времени.

3.1 Polus

Значения времени выполнения в таблице приведены в секундах

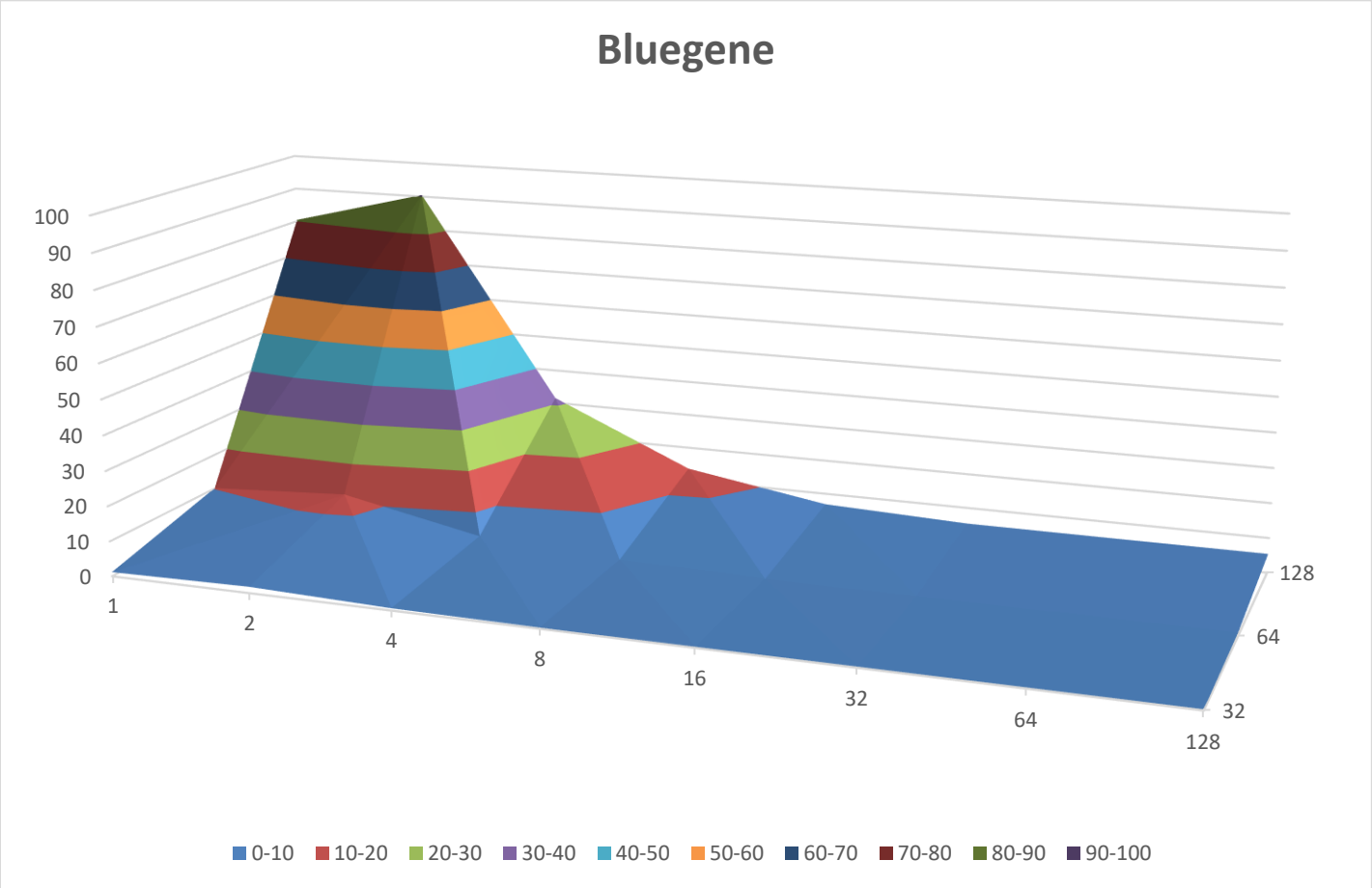
Размер сетки *Количество процессов*	**32*32*32**	**64*64*64**	**128*128*128**	**256*256*256**
1	0,071	0,564	4,484	36,196
2	0,107	0,707	5,082	36,599
4	0,047	0,273	1,885	14,142
8	0,041	0,231	1,237	6,430
16	0,031	0,135	0,579	3,818
32	0,024	0,110	0,474	2,355
Исходная программа	0,15	1,24	12,67	146,42

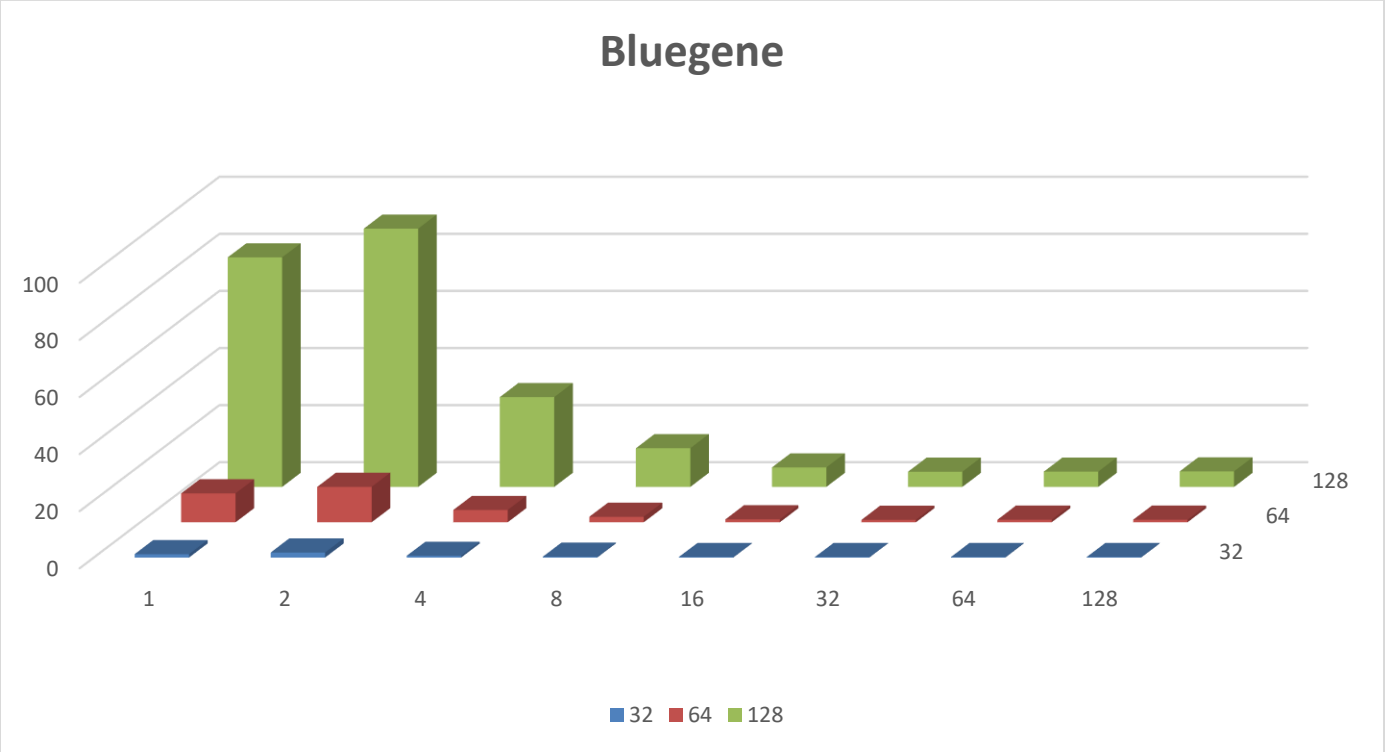

POLUS
160
140
120
100
80
60
40
20
0
Исходная программа
1
2
4
8
16
32
256
128
64
32
0-20
20-40
40-60
60-80
80-100
100-120
120-140
140-160


POLUS
160
140
120
100
80
60
40
20
0
Исходная программа
1
2
4
8
16
32
256
128
64
32
32
64
128
256

3.2 Bluegene

Размер сетки / *Количество процессов*	**32*32*32**	**64*64*64**	**128*128*128**
1	1,181	10,111	80,396
2	1,764	12,392	90,458
4	0,610	4,290	31,468
8	0,295	1,911	13,552
16	0,180	1,019	6,824
32	0,186	0,848	5,331
64	0,186	0,886	5,360
128	0,186	0,886	5,433





Видим, что распараллеленная программа показала себя лучше на суперкомпьютере Polus.

Можно заметить, что при переходе от одного процесса к двум время выполнения немного увеличивается, а затем при выполнении четырьмя процессами сильно падает.

4 Выводы

Выполнена работа по разработке параллельной версии алгоритма релаксации Якоби. Изучены технологии написания параллельных алгоритмов MPI. Представлено время выполнения распараллеленной программы на различных количествах процессоров Polus и Bluegene.