МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Практикум по учебному курсу

"Суперкомпьютеры и параллельная обработка данных"

Задание №2:

Разработка параллельной версии программы Jacobi-1D Stencil с использованием технологии MPI.

Отчет

студента 328 группы

факультета ВМК МГУ

Медведева Ивана Владимировича

2019 год

Постановка задачи

Имеется одномерная краевая задача для уравнения теплопроводности, которая решается методом Якоби - для каждой ячейки её значение вычисляется как среднее арифметическое ее значения и значения соседей. Имеется последовательная реализация Jacobi-1D Stencil.

Требуется:

1) Разработать параллельную версию программы для задачи Jacobi-1D Stencil с использованием технологии MPI,
2) Исследовать масштабируемость полученной программы, построить графики зависимости времени её выполнения от числа используемых ядер и объёма входных данных.

Код программы

```
#include <mpi.h>

double bench_t_start, bench_t_end;

int nProcs, id, block, left, right;

MPI_Status status;

static
double rtclock()
{
    struct timeval Tp;
    int stat;
    stat = gettimeofday (&Tp, NULL);
    if (stat != 0)
      printf ("Error return from gettimeofday: %d", stat);
    return (Tp.tv_sec + Tp.tv_usec * 1.0e-6);
}


void bench_timer_start()
{
  bench_t_start = rtclock ();
}


void bench_timer_stop()
{
  bench_t_end = rtclock ();
}

void bench_timer_print()
{
  printf ("Time in seconds = %0.6lf\n", bench_t_end - bench_t_start);
}


static
void init_array (int n,
```

```
   float B[n])
{
    int i;

    for (i = 0; i < n; i++) {
        B[i] = ((float) i+ 2) / n;
    }
}


static
void print_array(int n,
   float A[n])

{
  int i;

  fprintf(stderr, "==BEGIN DUMP_ARRAYS==\n");
  fprintf(stderr, "begin dump: %s", "A");
  for (i = 0; i < n; i++)
    {
      if (i % 20 == 0) fprintf(stderr, "\n");
      fprintf(stderr, "%0.2f ", A[i]);
    }
  fprintf(stderr, "\nend   dump: %s\n", "A");
  fprintf(stderr, "==END   DUMP_ARRAYS==\n");
}

static
void calc(int n, float B[n])
{
    int i;
    for (i = left; i < right; i++) {
        B[i] = 0.33333 * (B[i-1] + B[i] + B[i + 1]);
    }
}

static
void kernel_jacobi_1d(int tsteps,
       int n,
       float B[n])
{
    int t;

    MPI_Status status;
    block = (n - 2) / nProcs;
    if (id == nProcs - 1) {
        left = id * block + 1;
        right = n - 1;
    } else {
        left = id * block + 1;
        right = (id + 1) * block + 1;
    }
    for (t = 0; t < tsteps * 2; t++) {
        calc(n, B);
        if (t < tsteps * 2 - 1) {
            if (id == 0) {
                MPI_Send(&(B[right- 1]),  1, MPI_FLOAT, id + 1, 0, MPI_COMM_WORLD);
                MPI_Recv(&(B[right]),     1, MPI_FLOAT, id + 1, 0, MPI_COMM_WORLD, &status);
```

```
            } else if (id == nProcs - 1) {
                MPI_Recv(&(B[left - 1]),  1, MPI_FLOAT, id - 1, 0, MPI_COMM_WORLD, &status);
                MPI_Send(&(B[left]),      1, MPI_FLOAT, id - 1, 0, MPI_COMM_WORLD);
            } else {
                MPI_Recv(&(B[left - 1]),  1, MPI_FLOAT, id - 1, 0, MPI_COMM_WORLD, &status);
                MPI_Send(&(B[right - 1]), 1, MPI_FLOAT, id + 1, 0, MPI_COMM_WORLD);
                MPI_Recv(&(B[right]),     1, MPI_FLOAT, id + 1, 0, MPI_COMM_WORLD, &status);
                MPI_Send(&(B[left]),      1, MPI_FLOAT, id - 1, 0, MPI_COMM_WORLD);
            }
            MPI_Barrier(MPI_COMM_WORLD);
        } else {
            MPI_Gather(&(B[id * block + 1]), block, MPI_FLOAT, &(B[1]), block, MPI_FLOAT, nProcs -
1, MPI_COMM_WORLD);
        }
    }
}


int main(int argc, char** argv)
{
    int nes[] =      {30, 120, 400, 2000, 4000, 16000, 32000, 64000, 128000};
    int tstepses[] = {20, 40, 100, 500, 1000, 4000, 8000, 16000, 32000};
    int n, tsteps;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    int i;
    for (int i = 0; i < 1; i++) {
        n = nes[i];
        tsteps = tstepses[i];

        float (*B)[n];
        B = (float(*)[n]) malloc ((n) * sizeof(float));

        init_array(n, *B);
        if (id == nProcs - 1) {
            printf("%d, %d\n", n, tsteps);
            bench_timer_start();
        }
        kernel_jacobi_1d(tsteps, n, *B);
        if (id == nProcs - 1) {
            bench_timer_stop();
            bench_timer_print();
            print_array(n, *B);
        }

        free((void*)B);
    }
    MPI_Finalize();
    return 0;
}
```

Распараллелена программа классическими приёмами, рассмотренными на лекции и в учебном пособии Антонова А. С. «ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ MPI»

Описание программы

После вычисления своих элементов ядро должно переслать значения крайних элементов соседним ядрам, чтобы значения элементов на всех ядрах были в актуальном состоянии.

Результаты замеров времени выполнения

Работа задачи рассмотрена на суперкомпьютерах Polus и Bluegene с различным числом ядер (2 - 64) и различными размерами одномерной матрицы (30 — 128000). Каждое измерение проводилось 3 раза. В таблице и на графиках записаны усредненные результаты времени выполнения.

Таблица с результатами Polus

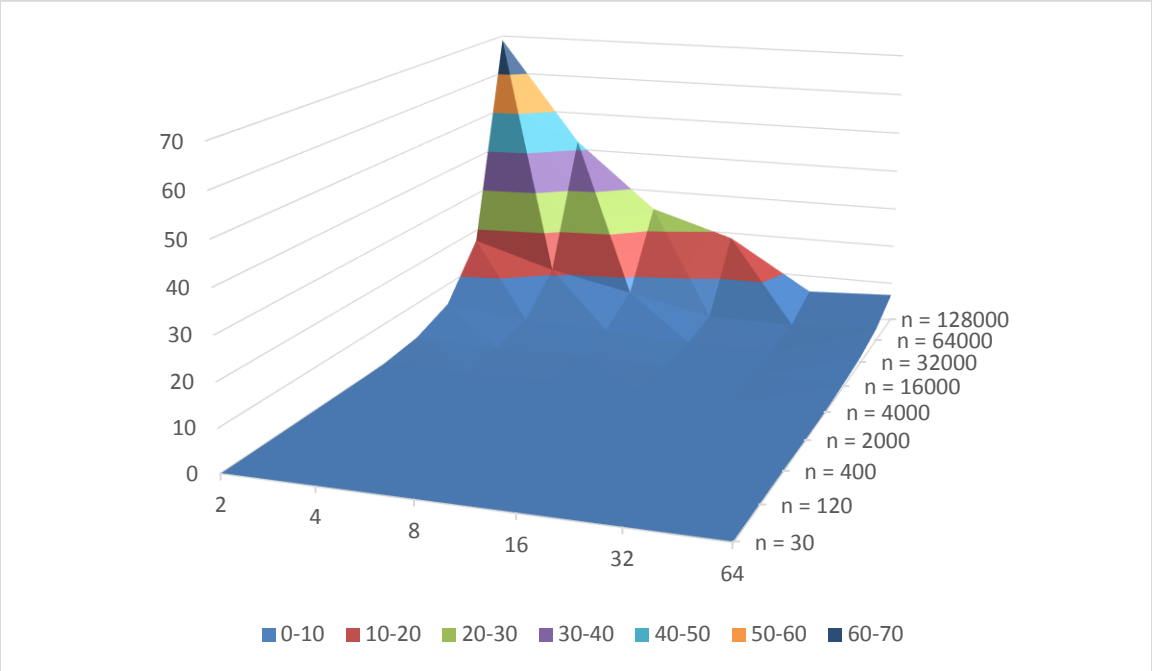
Нити и количество итераций / размер матрицы	n = 30 t = 20	n = 120 t = 40	n = 400 t = 100	n = 2000 t = 500	n = 4000 t = 1000	n = 16000 t = 4000	n = 32000 t = 8000	n = 64000 t = 16000	n = 128000 t = 32000
2	0.000174	0.000240	0.001275	0.015890	0.061476	1.173809	4.648203	17.160382	68.870355
4	0.000459	0.000542	0.001517	0.012900	0.039084	0.522160	2.521657	10.948961	42.161234
8	0.004570	0.001787	0.004109	0.022106	0.052489	0.477835	1.794539	6.618512	25.001374
16	0.002472	0.002383	0.005265	0.028050	0.058639	0.257708	0.702933	2.208911	18.628048
32	0.005835	0.004745	0.010192	0.051895	0.103763	0.380129	0.833827	2.066274	5.766549
64	0.013056	0.012689	0.027882	0.137557	0.189284	0.670890	1.393738	3.015119	6.816907

Таблица с результатами Bluegene

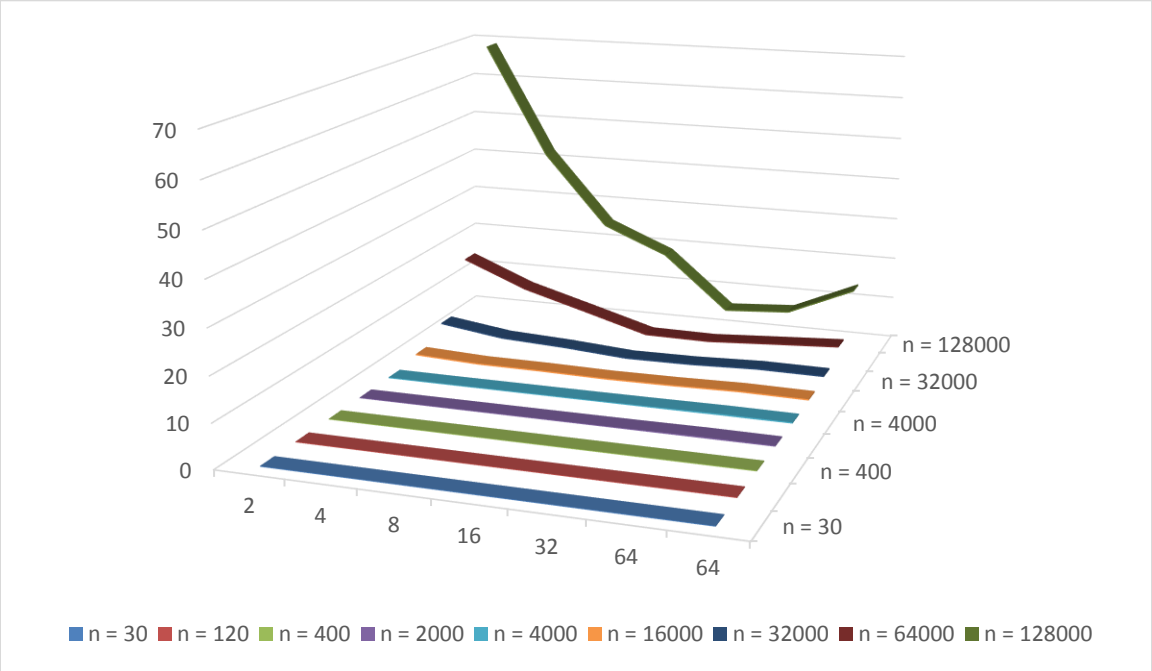
Нити и количество итераций / размер матрицы	n = 30 t = 20	n = 120 t = 40	n = 400 t = 100	n = 2000 t = 500	n = 4000 t = 1000	n = 16000 t = 4000	n = 32000 t = 8000	n = 64000 t = 16000	n = 128000 t = 32000
2	0.000453	0.001045	0.003974	0.060243	0.221860	3.484408	13.733114	54.440204	216.763241
4	0.001018	0.002105	0.005963	0.049917	0.150865	1.826093	7.299267	28.085288	110.128987
8	0.001726	0.003458	0.008982	0.054979	0.135556	1.150632	3.927259	15.409709	57.837071
16	0.003536	0.007107	0.018035	0.095581	0.204018	1.122699	3.054050	9.367682	34.937120
32	0.027287	0.055165	0.138838	0.696596	1.394106	5.623068	11.344840	23.098025	47.851294
64	0.029317	0.058999	0.148974	0.750072	1.503317	6.059333	12.212131	24.843652	51.315895

Графики: время выполнения программы в зависимости от размера матрицы и количества ядер Polus

В виде поверхности:

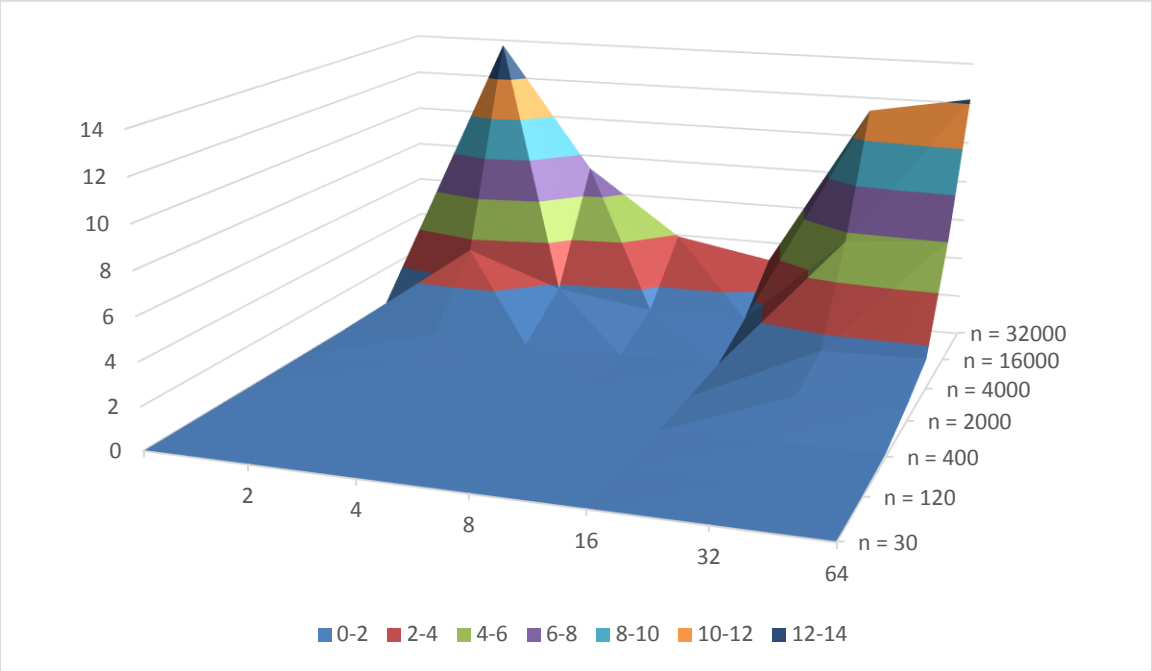


В виде линий в трехмерном пространстве:

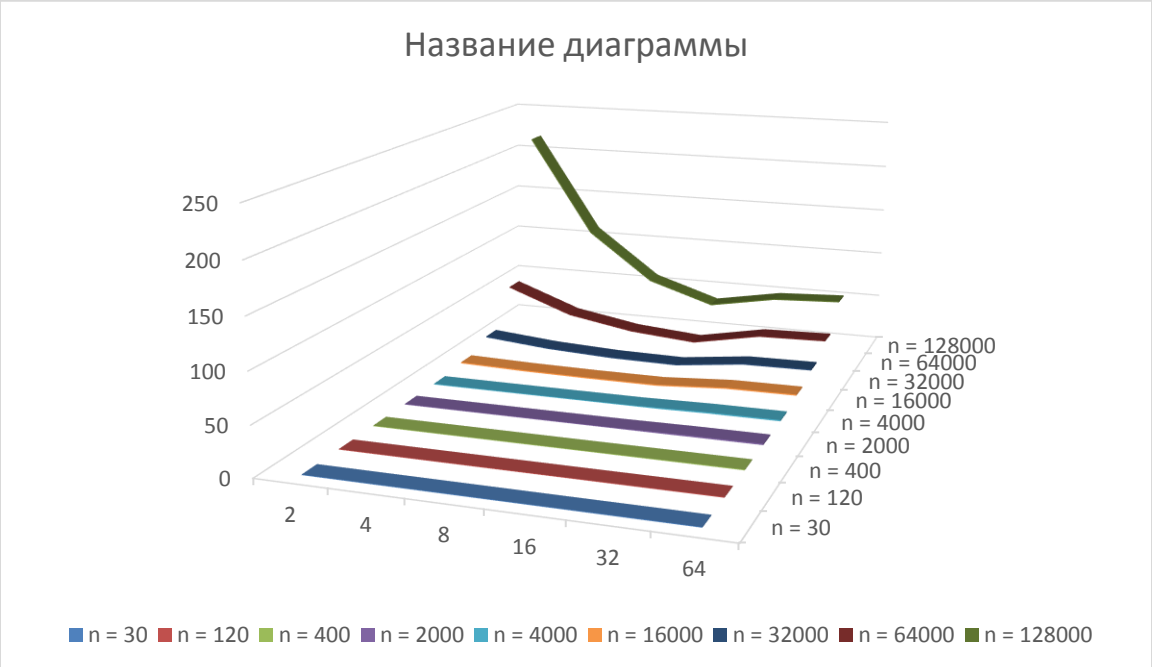


Графики: время выполнения программы в зависимости от размера матрицы и количества ядер Bluegene

В виде поверхности:



В виде линий в трехмерном пространстве:



Вывод:

Задача хорошо поддается распараллеливанию при помощи технологии MPI. Было получено ускорении в 6.35 раз на Bluegene и 13.5 на Polus. При значительном увеличении числа ядер время работы увеличивается. Возможно это связано с затратами на передачу сообщений, большим обменом информации.