МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Практикум по учебному курсу

"Суперкомпьютеры и параллельная обработка данных"

Разработка параллельной версии программы для решения задачи Vector Multiplication and Matrix Addition с помощью технологии OpenMP.

Отчет

студента 328 группы

факультета ВМК МГУ

Мамаева Павла Вячеславовича

2019 год

Оглавление

Постановка задачи 3

Код программы 4

Описание программы 8

Результаты замеров времени выполнения.....9

Таблица с результатами и графики 10

Анализ результатов 13

Выводы 14

Постановка задачи

Имеется код алгоритма *Vector Multiplication and Matrix Addition*, работающий с квадратной матрицей и векторами. Ставится задача разработки параллельной версии этого алгоритма

Требуется:

1. Разработать параллельную версию программы для задачи Vector Multiplication and Matrix Addition с использованием технологии OpenMP.
2. Исследовать масштабируемость полученной программы и построить графики зависимости времени её выполнения от числа используемых ядер и объёма входных данных.

Код программы

```
#define EXTRALARGE_DATASET

#include "gemver.h"

double bench_t_start, bench_t_end;

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 = omp_get_wtime();
}

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

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

static
void init_array(int n,
                float *alpha,
                float *beta,
                float A[n][n],
                float u1[n],
                float v1[n],
                float u2[n],
                float v2[n],
                float w[n],
                float x[n],
                float y[n],
                float z[n]) {
    int i, j;
    *alpha = 1.5;
    *beta = 1.2;
```

```
    float fn = (float) n;

    for (i = 0; i < n; i++) {
        u1[i] = i;
        u2[i] = ((i + 1) / fn) / 2.0;
        v1[i] = ((i + 1) / fn) / 4.0;
        v2[i] = ((i + 1) / fn) / 6.0;
        y[i] = ((i + 1) / fn) / 8.0;
        z[i] = ((i + 1) / fn) / 9.0;
        x[i] = 0.0;
        w[i] = 0.0;
        for (j = 0; j < n; j++)
            A[i][j] = (float) (i * j % n) / n;
    }
}

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

static
void kernel_gemver(int n,
                   float alpha,
                   float beta,
                   float A[n][n],
                   float u1[n],
                   float v1[n],
                   float u2[n],
                   float v2[n],
                   float w[n],
                   float x[n],
                   float y[n],
                   float z[n]) {
    int i, j;
```

```
#pragma omp parallel for private(i, j)
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         A[i][j] = A[i][j] + u1[i] * v1[j] + u2[i] * v2[j];

#pragma omp parallel for private(i, j)
   for (j = 0; j < n; j++)
      for (i = 0; i < n; i++)
         x[i] = x[i] + beta * A[j][i] * y[j];

#pragma omp parallel for private(i)
   for (i = 0; i < n; i++)
      x[i] = x[i] + z[i];

#pragma omp parallel for private(i, j)
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         w[i] = w[i] + alpha * A[i][j] * x[j];

}

int main(int argc, char **argv) {
   int n = N;
   float alpha;
   float beta;
   float (*A)[n][n];
   A = (float (*)[n][n]) malloc((n) * (n) * sizeof(float));
   float (*u1)[n];
   u1 = (float (*)[n]) malloc((n) * sizeof(float));
   float (*v1)[n];
   v1 = (float (*)[n]) malloc((n) * sizeof(float));
   float (*u2)[n];
   u2 = (float (*)[n]) malloc((n) * sizeof(float));
   float (*v2)[n];
   v2 = (float (*)[n]) malloc((n) * sizeof(float));
   float (*w)[n];
   w = (float (*)[n]) malloc((n) * sizeof(float));
   float (*x)[n];
   x = (float (*)[n]) malloc((n) * sizeof(float));
   float (*y)[n];
   y = (float (*)[n]) malloc((n) * sizeof(float));
   float (*z)[n];
   z = (float (*)[n]) malloc((n) * sizeof(float));
```

```
    init_array(n, &alpha, &beta,
               *A,
               *u1,
               *v1,
               *u2,
               *v2,
               *w,
               *x,
               *y,
               *z);

    print_array(n, *w);
    bench_timer_start();

    kernel_gemver(n, alpha, beta,
                  *A,
                  *u1,
                  *v1,
                  *u2,
                  *v2,
                  *w,
                  *x,
                  *y,
                  *z);

    bench_timer_stop();
    bench_timer_print();

    print_array(n, *w);
    if (argc > 42 && !strcmp(argv[0], ""))
        print_array(n, *w);

    free((void *) A);
    free((void *) u1);
    free((void *) v1);
    free((void *) u2);
    free((void *) v2);
    free((void *) w);
    free((void *) x);
    free((void *) y);
    free((void *) z);

    return 0;
}
```

Описание программы

Последовательный алгоритм предполагает последовательное вычисление значений матрицы ***A***, вектора ***x*** и вектора ***w***. Каждый последующий элемент зависит от предыдущего.

Заметим, что вычисление матрицы и двух векторов можно распараллелить, но нельзя нарушать последовательность вычисления, так как вектор ***x*** зависит от изменённой матрицы ***A***, а вектор ***w*** зависит от изменённого вектора ***x***.

Осуществим распараллеливание следующим образом:

- матрица ***A***: сделали так, чтобы каждая нить работала со своей строкой, что обеспечит наибольшую эффективность от распараллеливания
- вектор ***x***: видим, что также можем распараллелить внешний цикл, но в данном случае это не обеспечит нам максимального прироста скорости выполнения. Заметим, что при изменении индексов в матрице ***A*** можем добиться большего прироста скорости, поэтому изменили порядок обхода, для работы со строками
- вектор ***w***: здесь уже был реализован обход по строкам, поэтому просто предоставим каждой нити вычисления зависящие от одной строки в матрице ***A***

Верность работы алгоритма была проверена на результатах работы последовательного алгоритма.

GitHub: https://github.com/BinaryDancer/Supercomputer-practice/tree/master/OpenMP

Результаты замеров времени выполнения

Ниже приведены результаты замеров времени работы программ на суперкомпьютерах Bluegene и Polus: непосредственно в табличной форме и наглядно на графиках.

Программа была запущена в конфигурациях:

- на Polus - 1, 2, 4, 8, 16, 32, 64, 128 потоков;
- на Bluegene - 1, 2, 4 потоков.

Измерения проводились на различных объемах входных данных.

Название фдатасета	Значение N
MINI_DATASET	40
SMALL_DATASET	120
MEDIUM_DATASET	400
LARGE_DATASET	2000
EXTRALARGE_DATASET	4000

Каждая конфигурация запускалась 20 раз и все результаты были усреднены собственным скриптом.

Тесты:

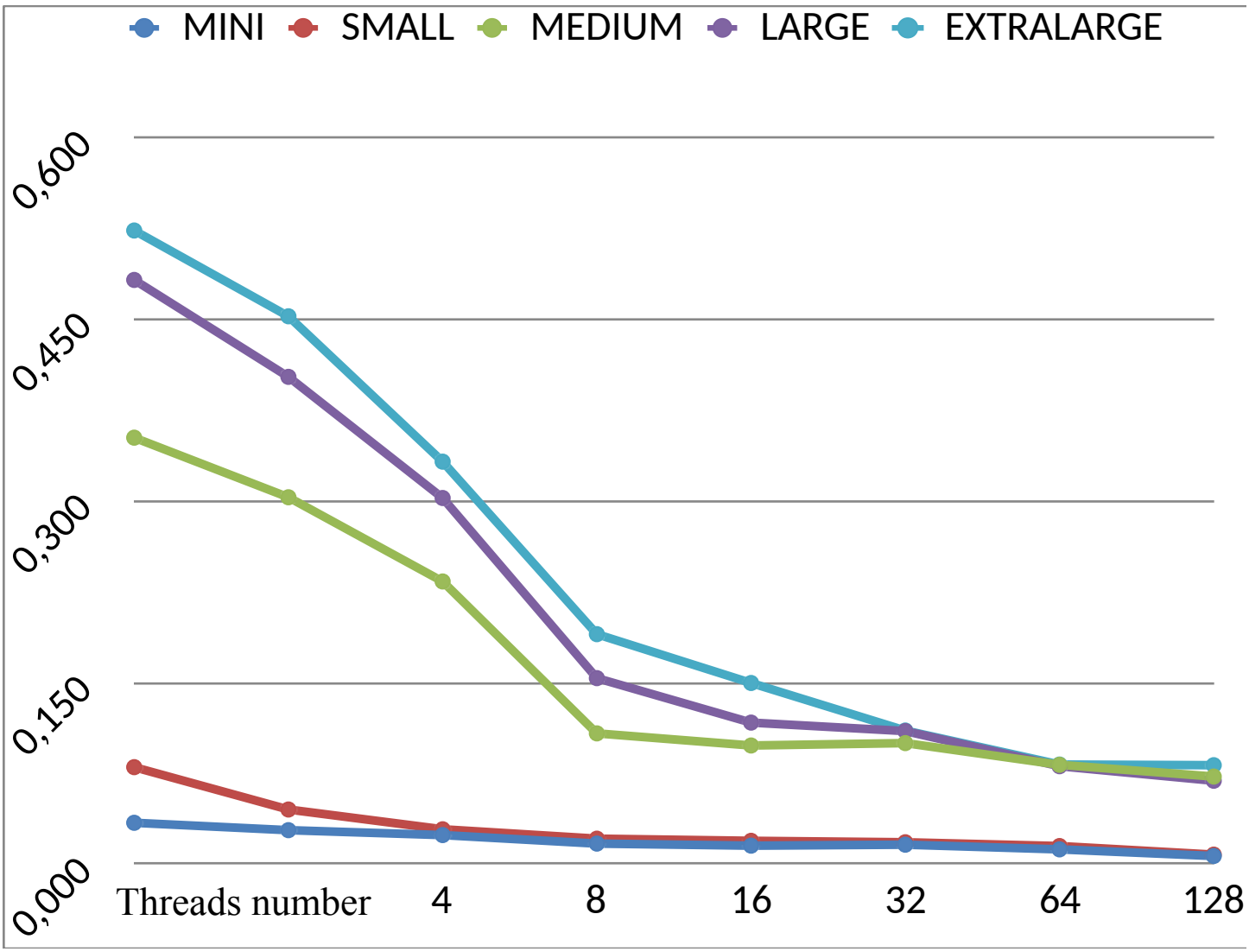
Bluegene: https://github.com/BinaryDancer/Supercomputer-practice/tree/master/OpenMP/BlueGene

Polus: https://github.com/BinaryDancer/Supercomputer-practice/tree/master/OpenMP/Polus

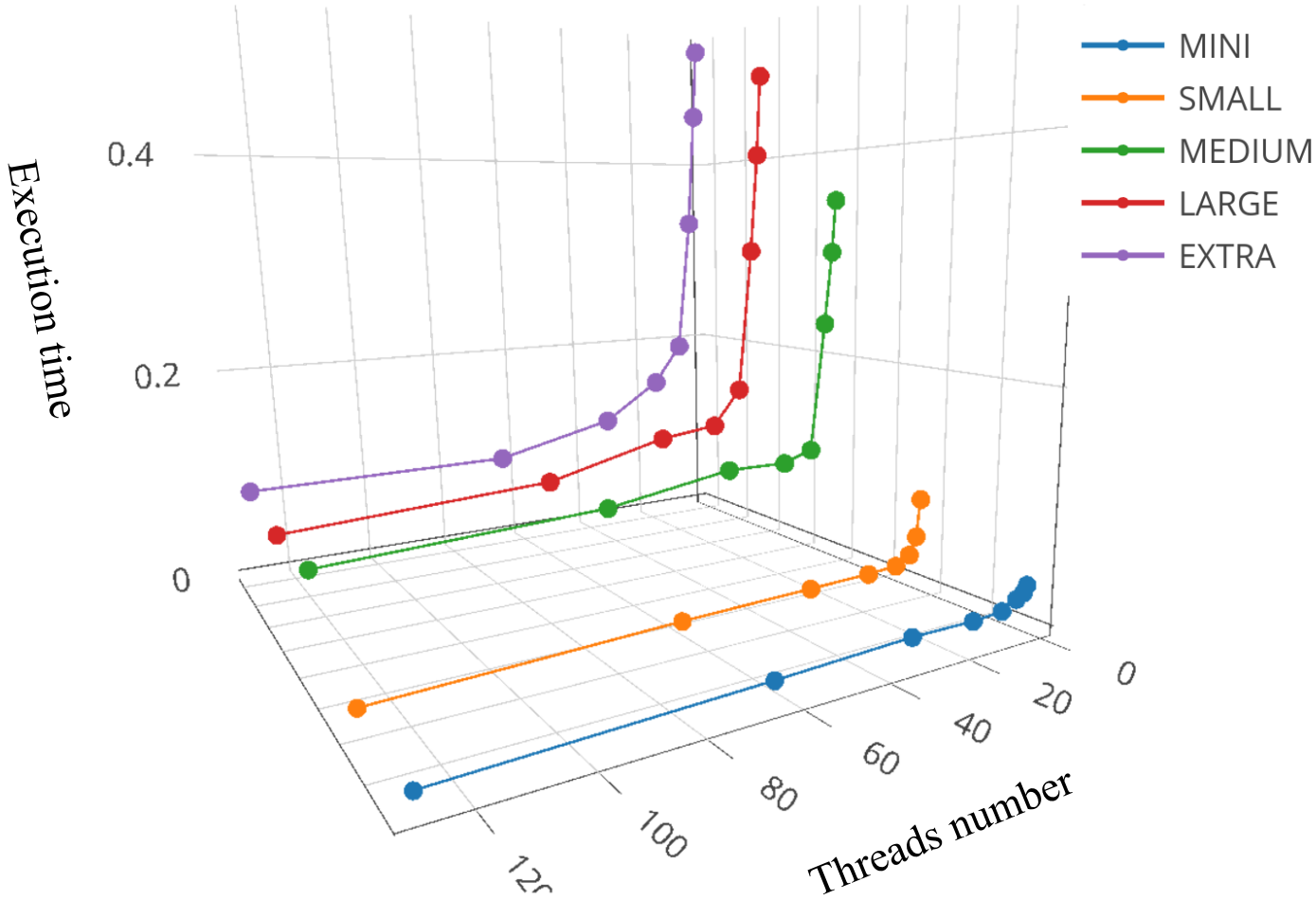
Таблица с результатами и графики

Polus:

	MINI	SMALL	MEDIUM	LARGE	EXTRALARGE
1	0.03420400	0.08012456	0.35211240	0.48232356	0.5231234
2	0.02810600	0.04516784	0.30291208	0.40231234	0.45231234
4	0.02405330	0.02884815	0.23348224	0.30235431	0.33235431
8	0.01702665	0.02109240	0.10802000	0.15361855	0.1899940
16	0.01532146	0.01931358	0.09859515	0.11694185	0.1496027
32	0.01612278	0.01812441	0.09998354	0.10997885	0.1103492
64	0.01226804	0.01571712	0.08205795	0.08094020	0.0822697
128	0.00679574	0.00791844	0.07236040	0.06891760	0.0817438

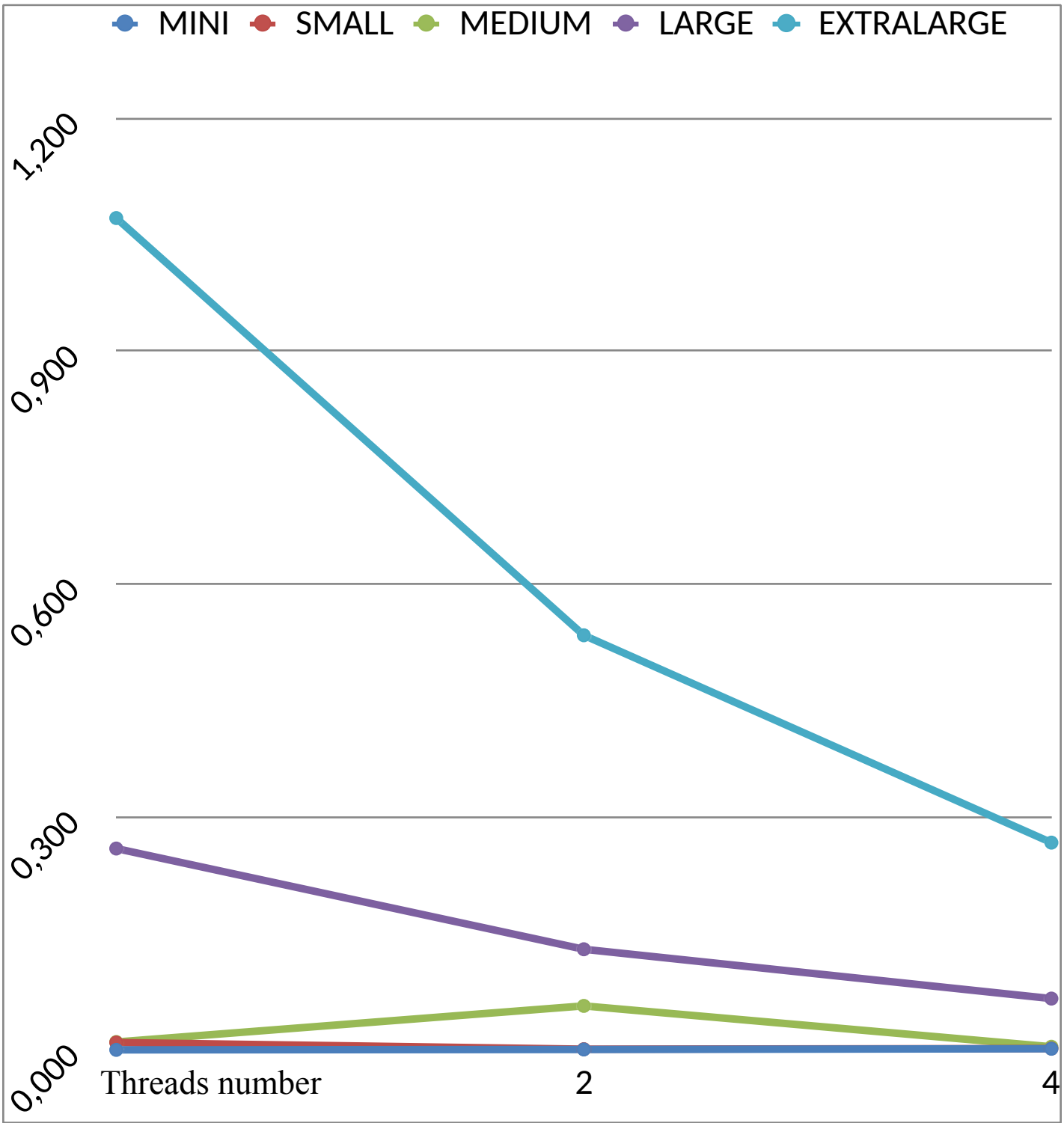


Polus


Execution time
0.4
0.2
0
0
20
40
60
80
100
Threads number
MINI
SMALL
MEDIUM
LARGE
EXTRA

Bluegene:

	MINI	SMALL	MEDIUM	LARGE	EXTRALARGE
1	0.000124	0.000958	0.0102540	0.259415	1.071306
2	0.000632	0.00104	0.05676	0.129681	0.534007
4	0.001572	0.001815	0.004137	0.066176	0.267007



Анализ результатов

На основе полученных результатов видно, что задача хорошо поддалась распараллеливанию и зависимость скорости работы программы от числа нитей близка к линейной.

Рассмотрев результаты работы программы на Bluegene, видим, что время работы на малых объемах данных распределено неравномерно. Это происходит из-за того, что система заточена не под многопоточные вычисления, а под многопроцессорные .

Из графиков видно, что время вычислений на машине Polus с увеличением числа нитей время работы уменьшается, но непропорционально, потому что при увеличении количества нитей увеличиваются накладные расходы на создание и взаимодействия потоков. Из-за этого и происходит такое незначительное уменьшение времени.

Выводы

Выполнена работа по разработке параллельной версии алгоритма *Vector Multiplication and Matrix Addition*. Изучена технология написания параллельных алгоритмов OpenMP. Проанализировано время выполнения алгоритмов на различных вычислительных системах.

Технология OpenMP удобна и проста в использовании, причем дает значительный прирост производительности на системах с многопоточными вычислениями. На больших объёмах данных получили время выполнения программы меньшее последовательного в 6 раз.