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 MPI.

Отчет

студента 328 группы

факультета ВМК МГУ

Мамаева Павла Вячеславовича

2019 год

Оглавление

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 MPI.
2. Исследовать масштабируемость полученной программы и построить графики зависимости времени её выполнения от числа используемых ядер и объёма входных данных.

Код программы

```
#define MINI_DATASET

#include "gemver.h"

double bench_t_start, bench_t_end;
int nProcs, rank;

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

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

void bench_timer_print() {
    if (rank == 0) {
        printf("Time in seconds = %0.6lf\n", bench_t_end - bench_t_start);
    }
}


int start_index, size;

static
void init_array(int n,
                float *alpha,
                float *beta,
                float A[size][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 (i = 0; i < size; ++i)
        for (j = 0; j < n; j++)
            A[i][j] = (float) ((i + start_index) * j % n) / n;
}

static
void print_array(int n,
                 float w[n]) {
    if (rank != 0) {
        return;
    }
    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[size][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;

  for (i = 0; i < size; i++) {
    for (j = 0; j < n; j++) {
      A[i][j] = A[i][j] + u1[i + start_index] * v1[j] + u2[i + start_index] * v2[j];
    }
  }

  for (i = 0; i < size; i++) {
    for (j = 0; j < n; j++) {
      x[j] = x[j] + beta * A[i][j] * y[i + start_index];
    }
  }

  MPI_Status status;
  if (rank == 0) {
    float (*temp_x)[n];
    temp_x = (float (*)[n]) malloc((n) * sizeof(float));

    for (i = 0; i < n; i++) {
      x[i] = x[i] + z[i];
    }

    for (i = 1; i < nProcs; ++i) {
      MPI_Recv(*temp_x, n, MPI_FLOAT, i, 23, MPI_COMM_WORLD, &status);
      for (j = 0; j < n; j++) {
        x[j] = x[j] + (*temp_x)[j];
      }
    }

    for (i = 1; i < nProcs; i++) {
      MPI_Send(x, n, MPI_FLOAT, i, 2312, MPI_COMM_WORLD);
    }
```

```
        free((void *) temp_x);

    } else {
        MPI_Send(x, n, MPI_FLOAT, 0, 23, MPI_COMM_WORLD);
        MPI_Recv(x, n, MPI_FLOAT, 0, 2312, MPI_COMM_WORLD, &status);

    }

    for (i = 0; i < size; i++) {
        for (j = 0; j < n; j++) {
            w[i + start_index] = w[i + start_index] + alpha * A[i][j] * x[j];
        }
    }

}

int min(int a, int b) {
    return (a < b) ? a : b;
}

int main(int argc, char **argv) {

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int n = N;
    start_index = n / nProcs * rank + min(rank, n % nProcs);
    size = n / nProcs + (rank < n % nProcs);

    float alpha;
    float beta;

    float (*A)[size][n];
    A = (float (*)[size][n]) malloc((size) * (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);

  bench_timer_start();

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

  if (rank == 0) {
     MPI_Status status;
     int j;
     for (j = 1; j < nProcs; j++) {
        int temp_size = n / nProcs + (j < n % nProcs);
        MPI_Recv(&((*w)[n / nProcs * j + min(j, n % nProcs)]), temp_size, MPI_FLOAT, j, 231299,
MPI_COMM_WORLD,
                 &status);
     }
  } else {
     MPI_Send(&((*w)[start_index]), size, MPI_FLOAT, 0, 231299, MPI_COMM_WORLD);
  }

  bench_timer_stop();
```

```
    bench_timer_print();

//    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);

    MPI_Finalize();

    return 0;
}
```

Описание программы

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

Выделим каждому процессу одинаковое количество строк матрицы ***A***, чтобы объём вычислений на каждый процесс был одинаков, таким образом каждый процесс будет просчитывать только те части, которые зависят от выделенных данному процессу строк матрицы ***A***. Последняя итерация в ***kernel_gemver()*** должна работать с изменённым вектором ***x***, но каждый элемент этого вектора зависит от всех строк в матрице ***A***, поэтому объединим промежуточные вычисления в процессе, у которого ***rank==0***. После объединения результатов отправим итоговый вектор ***x*** обратно всем процессам и произведем финальные вычисления.

Верность работы алгоритма была проверена на результатах работы последовательного алгоритма.

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

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 процессов;
- на Bluegene - 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 процессов.

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 3 раз и все результаты были усреднены собственным скриптом.

Результаты тестов:

Bluegene: https://github.com/BinaryDancer/../MPI/BlueGene-results

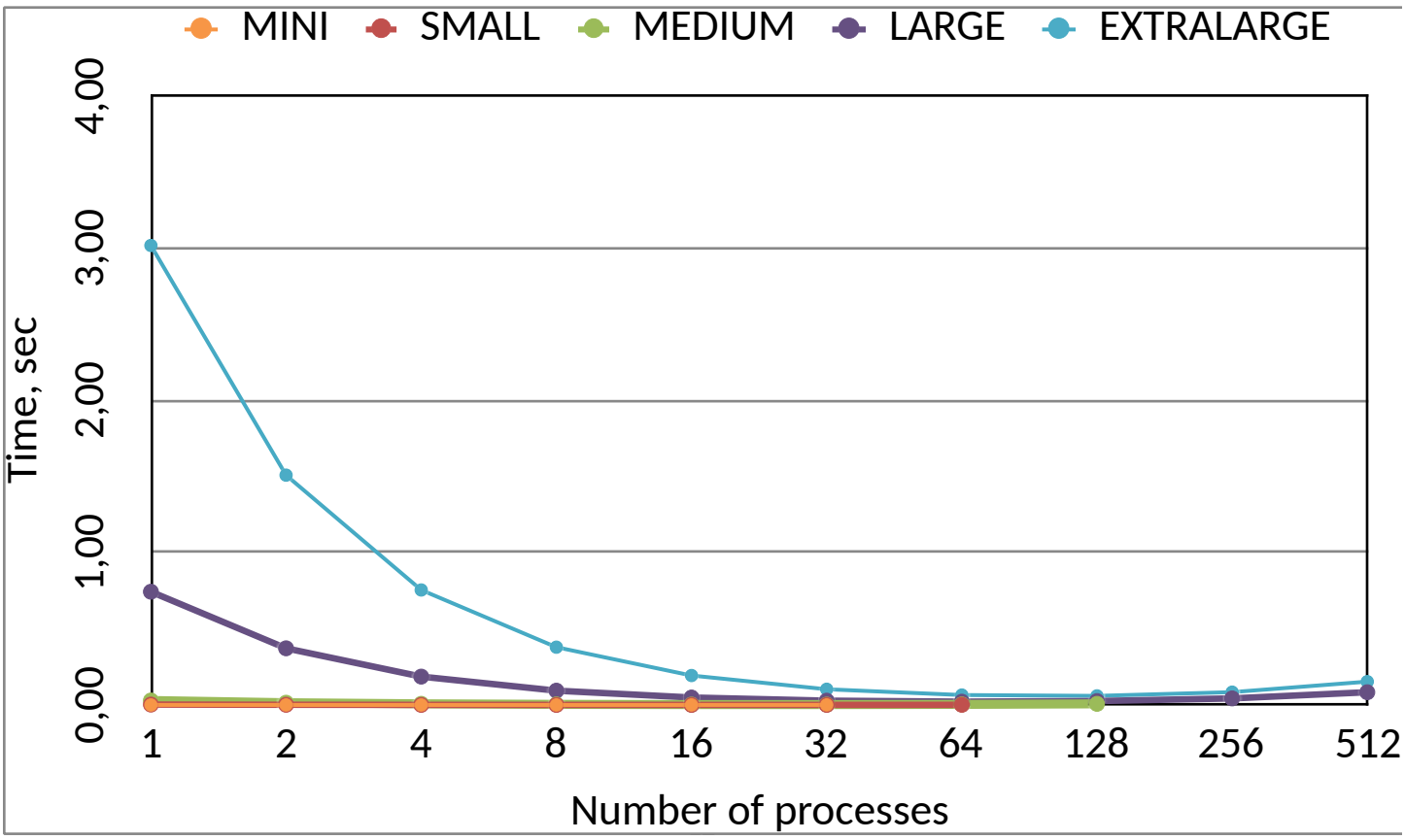
Polus: https://github.com/BinaryDancer/../MPI/Polus-results

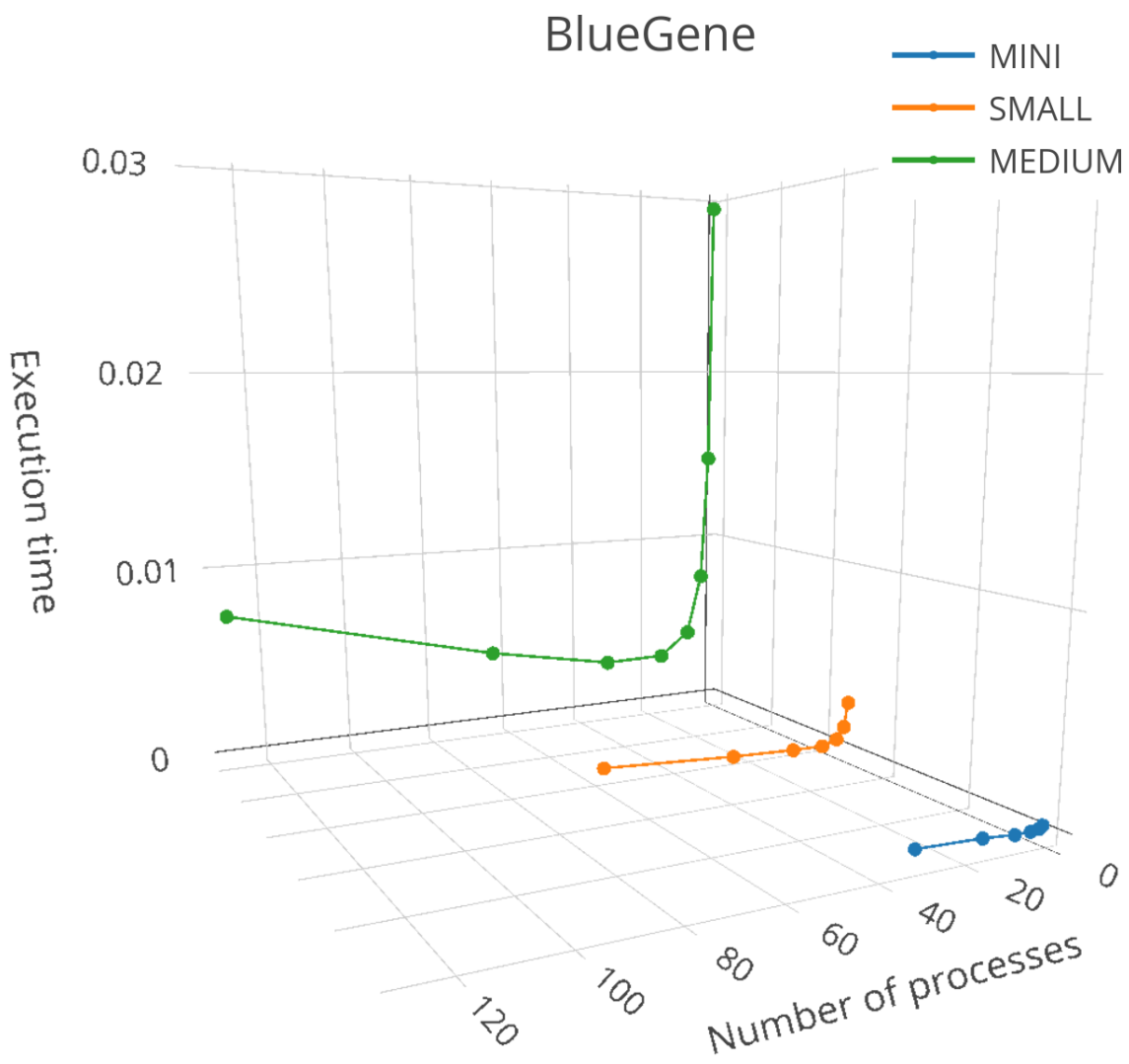
Таблица с результатами и графики

На 3D графиках были оставлены датасеты MINI, SMALL и MEDIUM для сохранения информативности графиков, так как при добавлении результатов по LARGE теряется информативность из-за резкого изменения масштаба.

BlueGene:

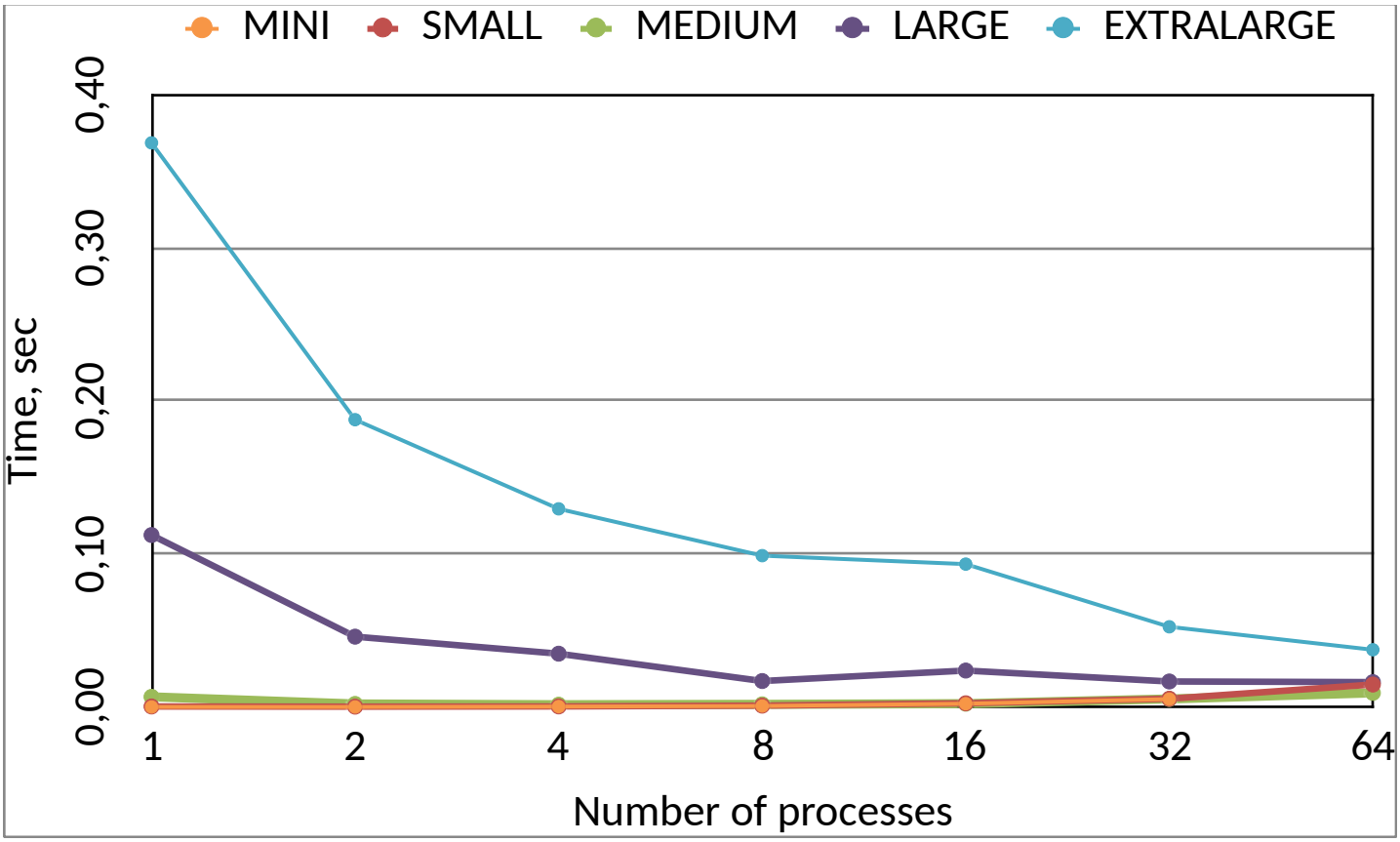
	MINI	SMALL	MEDIUM	LARGE	EXTRALARGE
1	0,000296	0,002659	0,029432	0,743683	3,017237
2	0,000168	0,001353	0,014822	0,372192	1,509140
4	0,000115	0,000720	0,007582	0,186340	0,755184
8	0,000117	0,000453	0,004120	0,094020	0,379537
16	0,000268	0,000491	0,002797	0,048959	0,193430
32	0,000410	0,000636	0,002749	0,028288	0,103298
64		0,001062	0,004055	0,021752	0,065350
128			0,007482	0,025762	0,059846
256				0,043185	0,084302
512				0,083778	0,153312

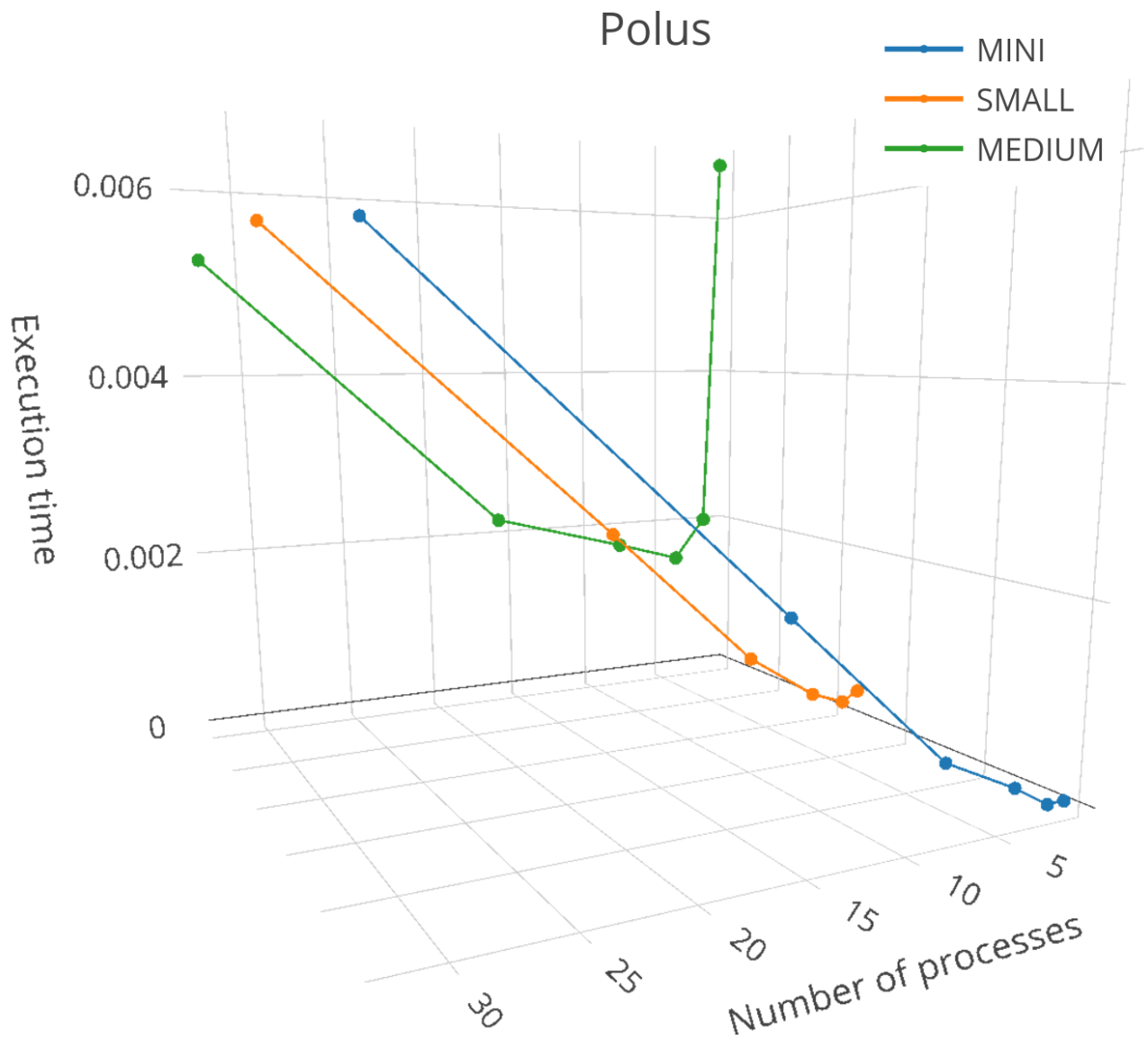



BlueGene
MINI
SMALL
MEDIUM
Execution time
0.03
0.02
0.01
0
0
20
40
60
80
100
120
Number of processes

Polus:

	MINI	SMALL	MEDIUM	LARGE	EXTRALARGE
1	0,0000447	0,0003387	0,0066443	0,1123800	0,3687110
2	0,0000447	0,0002337	0,0020023	0,0459753	0,1877723
4	0,0002777	0,0003800	0,0014983	0,0348370	0,1295033
8	0,0006573	0,0009000	0,0017357	0,0169697	0,0989137
16	0,0021627	0,0024217	0,0021777	0,0238143	0,0933797
32	0,0051537	0,0053963	0,0052513	0,0166743	0,0524147
64		0,0146890	0,0092150	0,0162697	0,0374123




Polus
MINI
SMALL
MEDIUM
Execution time
0.006
0.004
0.002
0
30
25
20
15
10
5
Number of processes

Анализ результатов

На основе полученных результатов видно, что задача поддалась распараллеливанию и зависимость скорости работы программы от числа процессов близка гиперболической.

Рассмотрев результаты работы программы на Bluegene, видим, система хорошо работает с многопроцесными программами. При увеличении числа процессов идет значительный прирост производительности, прирост замедляется, а потом сходит на нет, когда количество процессов начинает причинять вред времени выполнения программы из-за большого числа обменов информации между ними.

На машине Polus изменение времени работы программы гораздо хуже. На относительно малых размерах датасетов видим, что максимальный прирост производительности - это уменьшение времени работы всего в 2 раза. Вероятно, это связано со структурой вычислительного ресурса.

В итоге, максимальный прирост производительности на машине Polus уменьшил время работы примерно в 10 раз на больших данных, а на машине BlueGene время выполнения изменилось в 50 раз в лучшую сторону.

Выводы

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

Технология MPI предназначена для использования в многопроцессорных системах, виден значительный прирост производительности при увеличении числа процессов. На больших объёмах данных смогли уменьшить время выполнения программы в 50 раз.