Московский государственный университет
имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики

Практикум по курсу
"Суперкомпьютеры и параллельная обработка данных»

Разработка параллельной версии программ, включающей в себя умножение векторов и сложение матриц
MPI версия

ОТЧЕТ
о выполненном задании

Студента 320 учебной группы факультета ВМК МГУ
Попова Алексея Павловича

Москва, 2019

Оглавление

Постановка задачи ----------3

Описание алгоритма ----------4

Простая последовательная версия алгоритма (1) ----------4

Улучшенная последовательная версия алгоритма (2) ----------5

Параллельная версия алгоритма на MPI (3) ----------6

Тестирование программы ----------11

Тестирование программ на Bluegene ----------12

Тестирование программ на Polus ----------16

Анализ полученных результатов ----------18

Выводы ----------19

Постановка задачи

1) Реализовать алгоритм параллельного подсчета ядра *gemver* с использованием *MPI*.
2) Исследовать масштабируемость полученной параллельной программы: построить графики зависимости времени исполнения от числа ядер/процессоров для различного объёма входных данных.
3) Для каждого набора входных данных найти количество ядер/процессоров, при котором время выполнения задачи перестаёт уменьшаться.
4) Определить основные причины недостаточной масштабируемости программы при максимальном числе используемых ядер/процессоров.
5) Построить трехмерные графики зависимости времени выполнения программы от размера входных данных и количества ядер/процессоров.

Описание алгоритма

Простая последовательная версия алгоритма (1)

Начнем описание выполненной работы с определения полученного для улучшения алгоритма. Математически опишем производимые алгоритмом операции.

1) $A = A + u_1 \cdot v_1 + u_2 \cdot v_2$, где A - это матрица размера $n \times n$, а u_1 и u_2 - это столбы высоты n, v_1 и v_2 - это строки длины n.
2) $x = x + \beta \cdot y \cdot A$, где A - это матрица размера $n \times n$, а x и y - это вектор строки длины n, β - вещественный коэффициент.
3) $x = x + z$, где x и z - это вектор столбцы высоты n.
4) $w = w + \alpha \cdot A \cdot x$, где A - это матрица размера $n \times n$, а w и x - это вектор столбцы высоты n, α - вещественный коэффициент.

Приведем основную часть текста программы на языке *c*.

```
int 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];
   }
}
for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
      x[i] = x[i] + beta * A[j][i] * y[j];
   }
}
for (i = 0; i < n; i++) {
   x[i] = x[i] + z[i];
}
for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
      w[i] = w[i] + alpha * A[i][j] * x[j];
   }
}
```

Улучшенная последовательная версия алгоритма (2)

Очевидно, что описанный в предыдущем пункте алгоритм работает верно и выполняет нужные математические операции, но работа его на аппаратном уровне не может считаться оптимальной. Обратим внимание на то, что массив *A* во втором цикле обходится в неправильном порядке, что в свою очередь не позволяет работать КЭШу. Приведем далее текст программы, в которой исправлен этот недочет.

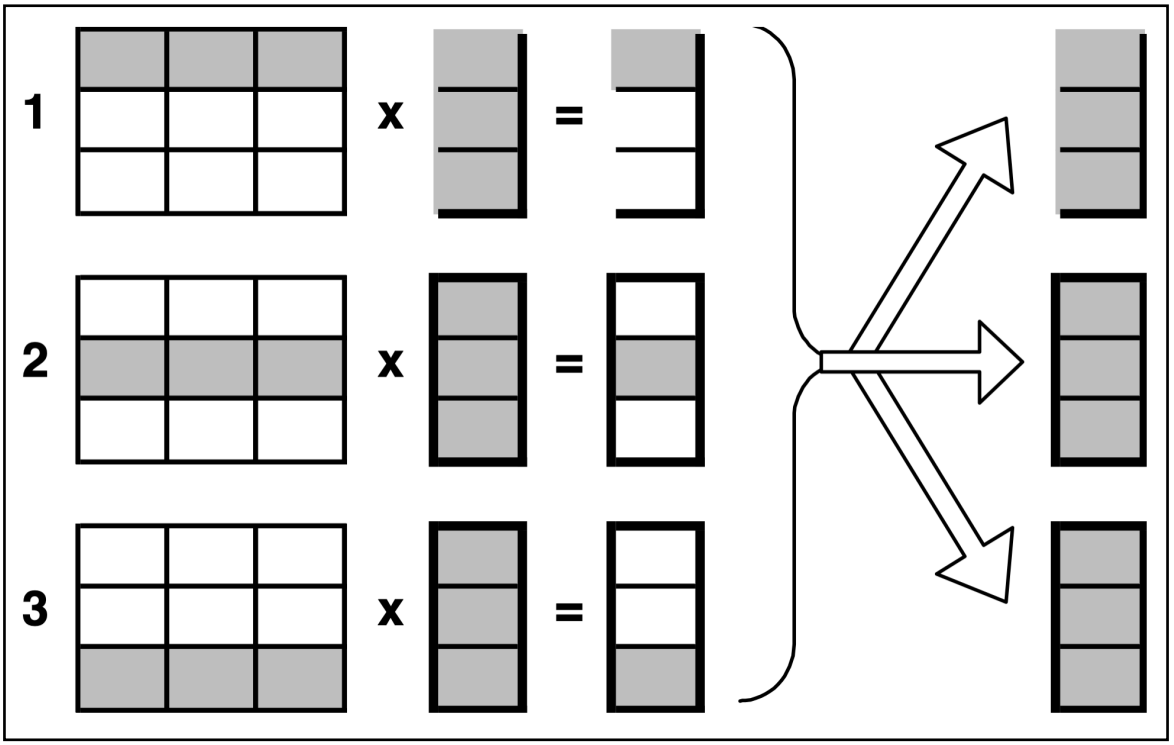
```
for (int i = 0; i < n; ++i) {
   for (int j = 0; j < n; ++j) {
      A[i][j] = A[i][j] + u1[i] * v1[j] + u2[i] * v2[j];
   }
}
for (int i = 0; i < n; ++i) {
   for (int j = 0; j < n; ++j) {
      x[j] = x[j] + beta * A[i][j] * y[i];
   }
}
for (int i = 0; i < n; ++i) {
   x[i] = x[i] + z[i];
}
for (int i = 0; i < n; ++i) {
   for (int j = 0; j < n; ++j) {
      w[i] = w[i] + alpha * A[i][j] * x[j];
   }
}
```

Важно заметить, что полученный алгоритм отлично подходит для дальнейшего распределенного вычисления. Можно заметить, что отличным местом для синхронизации всех процессов является третий цикл. Также несложно заметить, что все внешние циклы идут по строкам матрицы и в некоторых случаях описывают лишь часть векторов, что дает свободу для распределения данных.

Параллельная версия алгоритма на MPI (3)

В процессе анализа полученной программы и идеи распределения данный по вычислительным узлам, было принято решение, распределить матрицу в памяти по строкам.

Основную сложность в распределении MPI программы составляет выбор распределения матриц по вычислительным ядрам. В решении этой проблемы я воспользовался данными представленными на лекции. По моим данным ленточная схема распределения матрицы по строкам (Изображение 1), работает наиболее быстро. Это обосновано расположением матрицы в памяти, и, соответственно, эффективной работой КЭШа при распределении по строкам.

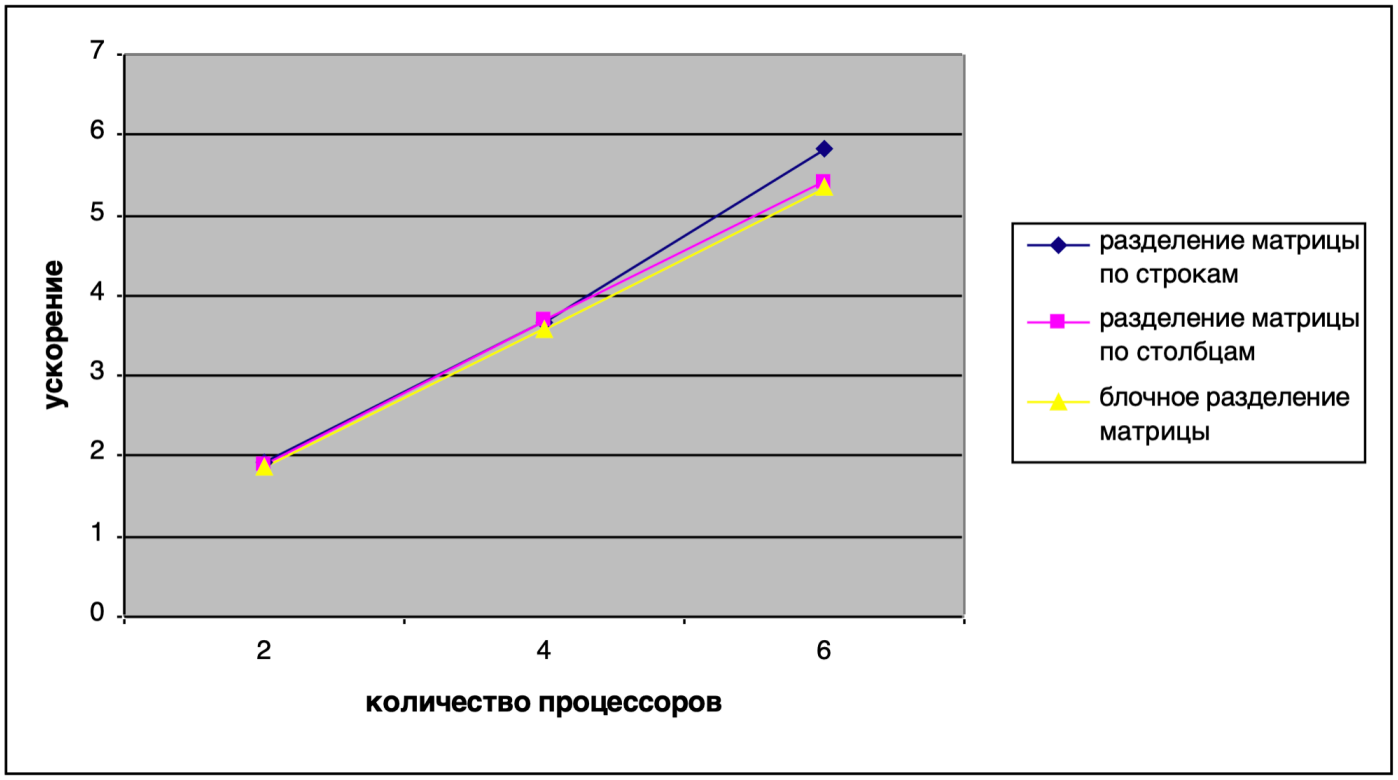


Изображение 1 (Ленточная схема распределения матрицы по строкам)

Далее обоснуем сделаный выбор, в пользу ленточной схемы распределения по строкам, сравнительным анализом, который также представлен в предложенных лекционных материалах (Изображение 2).

В программе осуществлено наиболее полное распределение данных, для этого даже векторы были определены кусочками в памяти соответствующих вычислительных узлов. Все вышеперечисленное позволило значительно сэкономить требования программы к объему памяти на узлах. Также в результате отладки и тестирования программы было принято решение - сделать точку синхронизации всех

процессов на шаге 3 математического описания алгоритма (или в перед третьим циклом алгоритма). Это позволило сделать работу алгоритма безошибочной, так как каждый процесс полностью изменяет массив *x*. Эти обмены происходят всего один раз, что хорошо сказывается на производительности программы. Все обмены блокирующие, этот метод был выбран осознанно, так как все возможные проблемы с блокировками были учтены.



Изображение 2 (Сравнительный анализ методов распределения данных)

Немаловажным фактом данной реализации является конечная сборка результата, несмотря на то, что она замедляет программу - это позволяет проверить корректность работы программы и точно замерить время ее исполнения. Понятно, что во многих случаях не требуется итоговая сборка результата, так как каждый программный код, исполняемый на суперкомпьютере является частью какого-то большого вычислительного алгоритма, и данные распределенные на данном этапе будут использованы далее. Так, например, *gemver* является частью большого *benchmark* теста для высокопроизводительных систем и входит во множество пакетов тестирования, и в таком случае он участвует в последовательности выполнении множества операций над матрицами. Очевидно, что в вышеописанном варианте использования реализованных операций не требуется конечная сборка данных, но в учебных целях она просто необходима.

Далее будет приведен код программы в нескольких кусках, инициализация с расчетом смещений (Программный код 1), сам вычислительный участок кода с синхронизацией (Программный код 2) и код основой инициализации (Программный код 3).

```
static
void init_array (int n,
    int size,
    int start,
    int end,
    double *alpha,
    double *beta,
    double A[ size][n],
    double u1[ size],
    double v1[ n],
    double u2[ size],
    double v2[ n],
    double w[ size],
    double x[ n],
    double y[ size],
    double z[ n])
{
    *alpha = 1.5;
    *beta = 1.2;
    double fn = (double)n;
    for (int i = 0; i < n; i++) {
        v1[i] = ((i+1)/fn)/4.0;
        v2[i] = ((i+1)/fn)/6.0;
        z[i] = ((i+1)/fn)/9.0;
        x[i] = 0.0;
    }

    for (int i = start; i < end; i++) {
        u1[i - start] = i;
        u2[i - start] = ((i+1)/fn)/2.0;
        y[i - start] = ((i+1)/fn)/8.0;
        w[i- start] = 0.0;
        for (int j = 0; j < n; j++) {
            A[i - start][j] = (double) (i*j % n) / n;
        }
    }
}
```

Программный код 1 (Алгоритм инициализации)

Считаю важным показать измененную функцию инициализации, так как от нее в значительной степени зависит дальнейшая работа. Так как в первую очередь распределенную память нужно инициализировать.

```
static
void kernel_gemver(int n, int size,
   double alpha,
   double beta,
   double A[ size][n],
   double u1[ size],
   double v1[ n],
   double u2[ size],
   double v2[ n],
   double w[ size],
   double x[ n],
   double y[ size],
   double z[ n])
{
   /**********************/
   MPI_Status status[1];
   double (*cur)[n];
   cur = (double(*)[n])malloc ((n) *
sizeof(double));
   int myrank, ranksize;
   MPI_Comm_rank(MPI_COMM_WORLD,
&myrank);
   MPI_Comm_size(MPI_COMM_WORLD,
&ranksize);
   if (!myrank) {
      for (int j = 0; j < n; j++) {
         x[j] = x[j] + z[j];
      }
      for (int i = 1; i < use_proc; i++) {
         MPI_Recv((*cur), n, MPI_DOUBLE, i, 13,
MPI_COMM_WORLD, &status[0]);
         for (int j = 0; j < n; j++) {
            x[j] = x[j] + (*cur)[j];
         }
      }
      for (int i = 1; i < use_proc; i++) {
         MPI_Send(x, n, MPI_DOUBLE, i, 13,
MPI_COMM_WORLD);
      }
   } else {
      MPI_Send(x, n, MPI_DOUBLE, 0, 13,
MPI_COMM_WORLD);
      MPI_Recv(x, n, MPI_DOUBLE, 0, 13,
MPI_COMM_WORLD, &status[0]);
   }
   /**********************/
}
```

Программный код 2 (Реализация MPI в ядре)

Отметим, что в представленном выше алгоритме пропущены части, которые незначительно отличаются от последовательного алгоритма, ознакомиться с подробной версией можно в Приложении 1. Также важно отметить, что единичные Send/Recv в программе можно заменить на групповую операцию Bcast, результаты при использовании этой операции изменяются незначительно, поэтому измерения времени приводить для нее не будем. Часть кода, которая позволяет использовать эту операцию закомментирована и ознакомиться с ней можно также в Приложении 1.

```
int main(int argc, char** argv)
{
   int n = N;
   int myrank, ranksize;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,
&myrank);
   MPI_Comm_size(MPI_COMM_WORLD,
&ranksize);
   use_proc = ranksize;
   int k = n / use_proc;
   int m = n % use_proc;
   if (k == 0) {
      k = 1;
      m = 0;
      use_proc = n;
   }
   for (int i = 0; i < use_proc; i++) {
      if (myrank == i) {
         int start = min(i, m) + i * k;
         int size = k + (i < m);
         int end = start + size;
         MPI_Request req[1];
         MPI_Status status[1];
         double alpha;
         double beta;
         double (*A)[size][n];
         A = (double(*)[size][n])malloc ((size) *
(n) * sizeof(double));
         double (*u1)[size]
         u1 = (double(*)[size])malloc ((size) *
sizeof(double));
         double (*v1)[n];
         v1 = (double(*)[n])malloc ((n) *
sizeof(double));
         double (*u2)[size];
         u2 = (double(*)[size])malloc ((size) *
sizeof(double));
         double (*v2)[n];
         v2 = (double(*)[n])malloc ((n) *
sizeof(double));
         double (*w)[size];
         w = (double(*)[size])malloc ((size) *
sizeof(double));
         double (*x)[n];
         x = (double(*)[n])malloc ((n) *
sizeof(double));
         double (*y)[size];
         y = (double(*)[size])malloc ((size) *
sizeof(double));
         double (*z)[n];
         z = (double(*)[n])malloc ((n) *
sizeof(double));
         init_array (n, size, start, end, &alpha,
&beta, *A, *u1, *v1, *u2, *v2, *w, *x, *y, *z);
         double time_1;
         if (myrank == 0) {
            time_1 = MPI_Wtime();
         }
         kernel_gemver (n, size, alpha, beta, *A, *u1,
*v1, *u2, *v2, *w, *x, *y, *z);
         if (myrank == 0) {
            size = k + (0 < m);
            for (int k = 0; k < size; k++) {
               printf("%lf\n", (*w)[k]);
            }
            double (*cur)[size]; cur = (double(*)
[size])malloc ((size) * sizeof(double));
            for (int j = 1; j < use_proc; j++) {
               size = k + (j < m);
               MPI_Recv(*cur, size, MPI_DOUBLE,
j, 13, MPI_COMM_WORLD, &status[0]);
               for (int k = 0; k < size; k++) {
                  printf("%lf\n", (*cur)[k]);
               }
            }
            printf("MPI --- %lf\n", MPI_Wtime() -
time_1);
         } else {
            MPI_Send(w, size, MPI_DOUBLE, 0, 13,
MPI_COMM_WORLD);
         }
        /* free all local memory */
         break;
      }
   }
   MPI_Finalize();
   return 0;
}
```

Программный код 3 (Создание параллельных областей, сбор и распределение данных)

В целях экономии места в данном разделе предложен не весь код, с полной версией можно ознакомиться в Приложении 1.

Тестирование программы

В этом разделе представлены результаты тестирования, описанных в предыдущем разделе, программ и трехмерные графики зависимостей исходных параметров запуска. Тестирование производилось на компьютере *Bluegene*, *Polus*. В данном разделе на домашнем компьютере с *2.3 GHz Intel Core i5 7360U* тестирование не производилось, так как нельзя получить четкой картины, полезной для отчета.

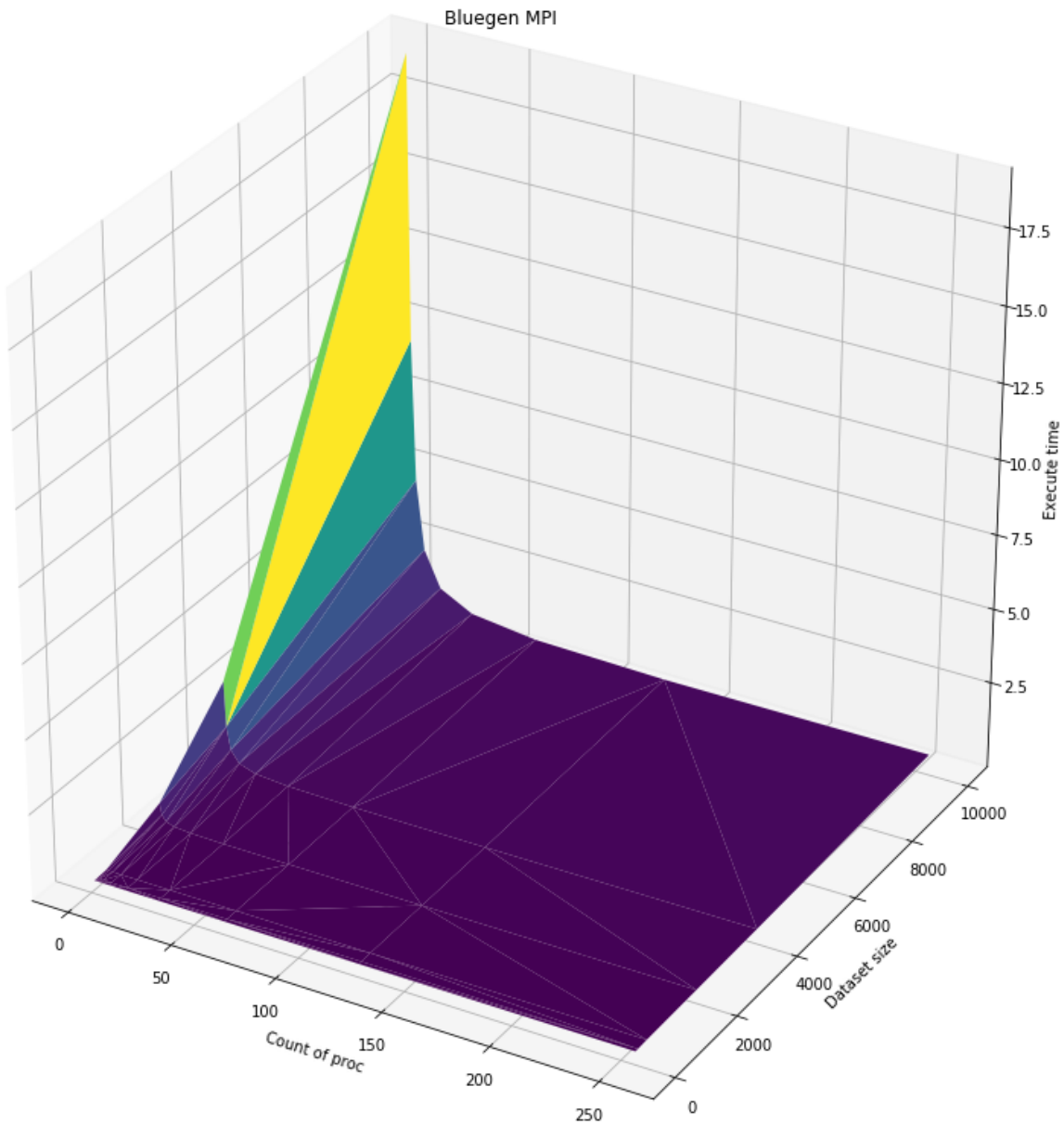
Конфигурации запущенных программ:

- *Bluegene* - 1, 2, 4, 8, 16, 32, 64, 128, 256 узлов.
- *Polus* - 1, 2, 4, 8, 16, 32, 64 узла.

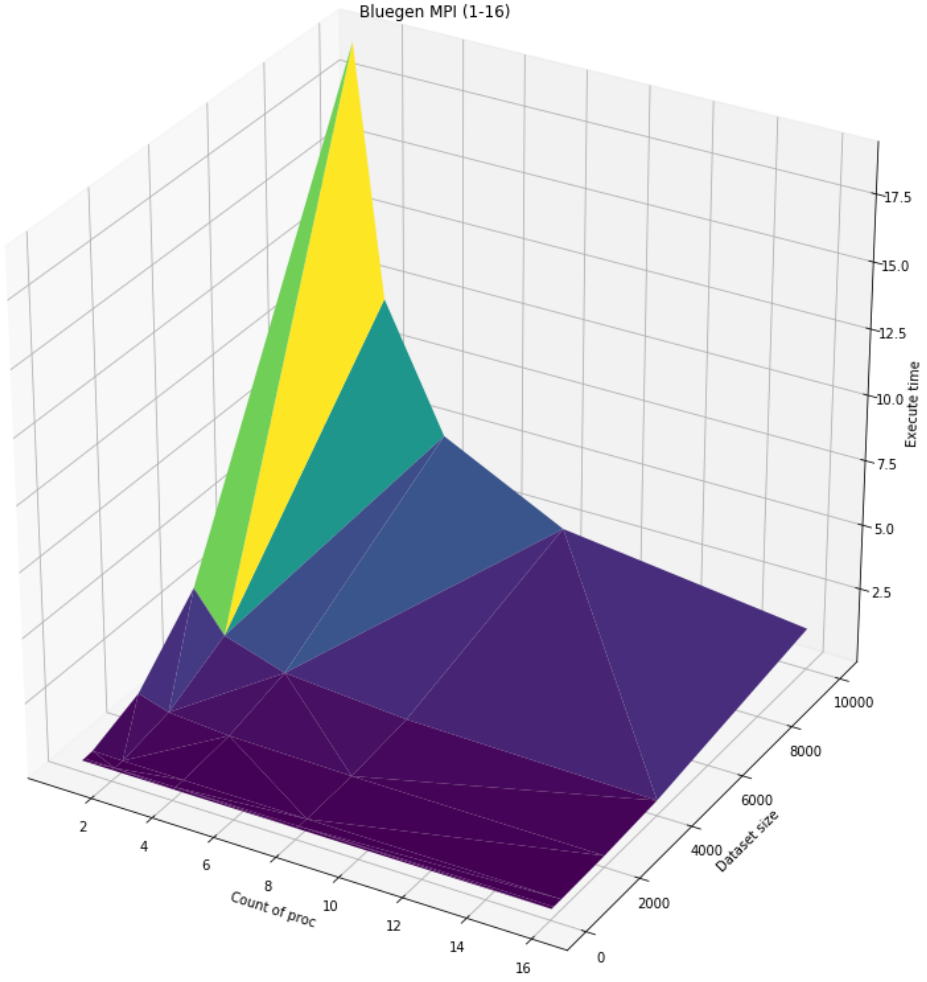
Тестирование программ на *Bluegene*

Тестирование производилось для разных размеров входных данных, в частности для размеров матриц и векторов от 40 до 10000 элементов. В данном отчете было решено добавить размерность 10000, чтобы продемонстрировать прирост от выполнения на нескольких узлах более явно.

Запуск тестов производился с использованием *Bash* файлов для автоматизации работы. Также после тестирования все данные собирались и усреднялись при помощи скрипта на языке *Python* (Приложение 2), что позволило значительно увеличить производительность выполняемой работы.



Изображение 3 (Полный график тестирования на *Bluegene*)

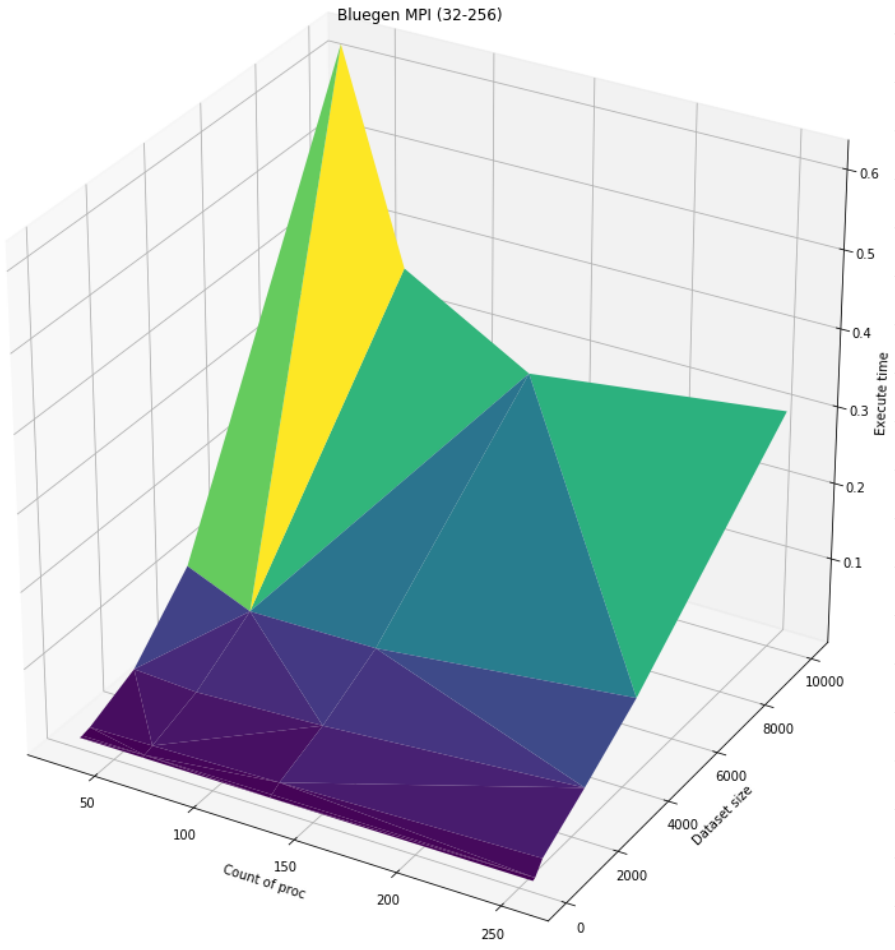


Count of threads	Dataset size	Execute time
1.0000	40.0000	0.0003
1.0000	120.0000	0.0026
1.0000	400.0000	0.0288
1.0000	2000.0000	0.7514
1.0000	4000.0000	3.0434
1.0000	10000.0000	19.0160
2.0000	40.0000	0.0002
2.0000	120.0000	0.0013
2.0000	400.0000	0.0144
2.0000	2000.0000	0.3756
2.0000	4000.0000	1.5224
2.0000	10000.0000	9.5094
4.0000	40.0000	0.0001
4.0000	120.0000	0.0007
4.0000	400.0000	0.0074
4.0000	2000.0000	0.1884
4.0000	4000.0000	0.7621
4.0000	10000.0000	4.7563
8.0000	40.0000	0.0001
8.0000	120.0000	0.0005
8.0000	400.0000	0.0041
8.0000	2000.0000	0.0964
8.0000	4000.0000	0.3833
8.0000	10000.0000	2.3979
16.0000	40.0000	0.0003
16.0000	120.0000	0.0005
16.0000	400.0000	0.0029
16.0000	2000.0000	0.0504
16.0000	4000.0000	0.1960
16.0000	10000.0000	1.2020

Изображение 4 (Тестирование с количеством узлов 1-16 на *Bluegene*)

В данном этапе отчета нет сравнения программы с разными оптимизациями, так как основная часть сравнений была произведена в предыдущей части. Для более наглядной иллюстрации времени работы программ в зависимости от начальных данных было принято разделить график на несколько частей (Изображение 3, 4, 5).

На Изображении 3 представлен график выполнения параллельной версии программы. В свою очередь на Изображении 4, для большей наглядности, представлен график и таблица для построения приведенного графика для 1-16 узлов, а на изображении 5 для 32-256 узлов.



Count of threads	Dataset size	Execute time
32.0000	40.0000	0.0005
32.0000	120.0000	0.0007
32.0000	400.0000	0.0031
32.0000	2000.0000	0.0303
32.0000	4000.0000	0.1069
32.0000	10000.0000	0.6232
64.0000	40.0000	0.0005
64.0000	120.0000	0.0011
64.0000	400.0000	0.0047
64.0000	2000.0000	0.0253
64.0000	4000.0000	0.0713
64.0000	10000.0000	0.3549
128.0000	40.0000	0.0005
128.0000	120.0000	0.0020
128.0000	400.0000	0.0088
128.0000	2000.0000	0.0325
128.0000	4000.0000	0.0724
128.0000	10000.0000	0.2609
256.0000	40.0000	0.0005
256.0000	120.0000	0.0020
256.0000	400.0000	0.0179
256.0000	2000.0000	0.0567
256.0000	4000.0000	0.1093
256.0000	10000.0000	0.3027

Изображение 5 (Тестирование с количеством узлов 32-256 на *Bluegene*)

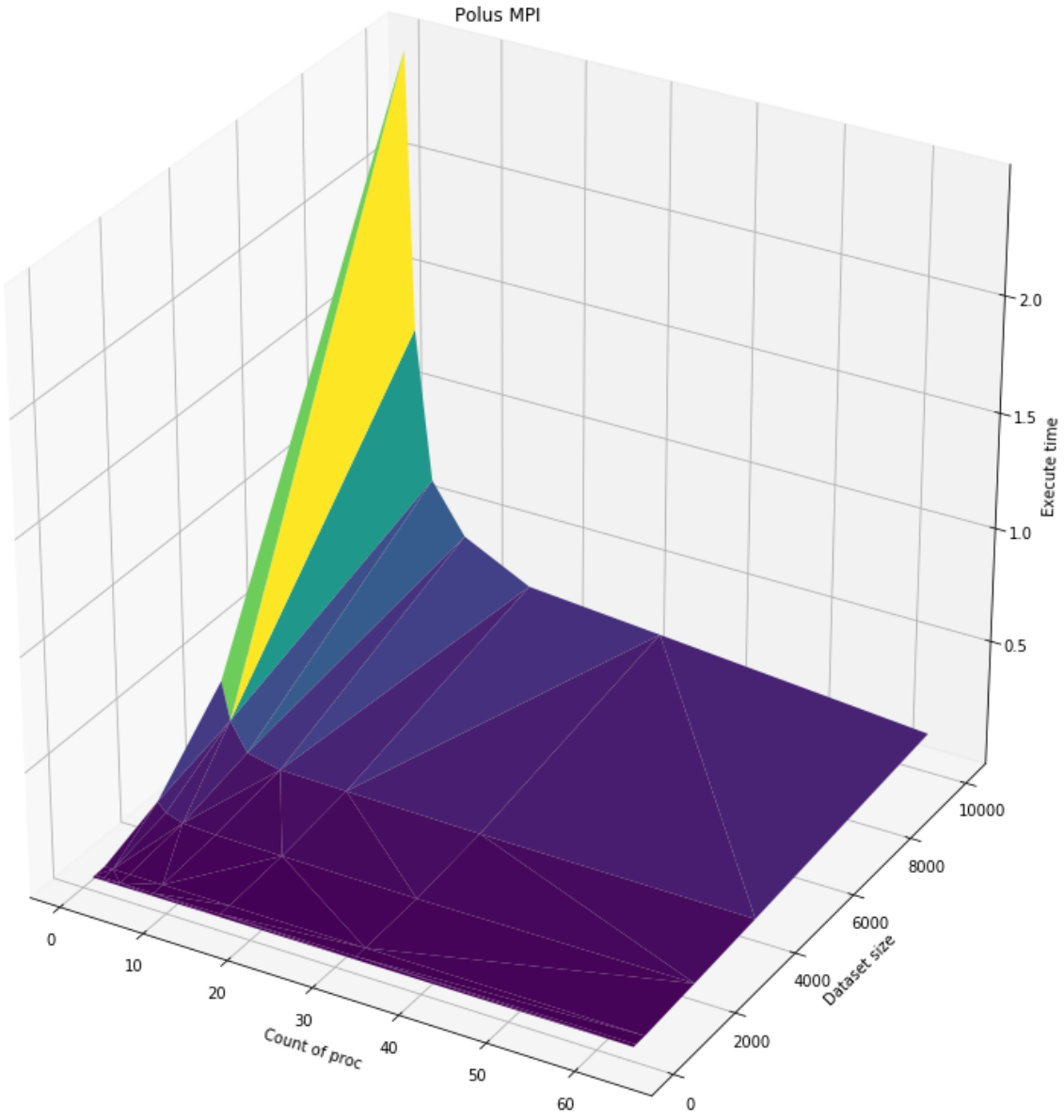
Сделаем некоторые выводы из приведенных выше результатов тестирования. Хорошо видно, что использование технологии MPI позволяет значительно уменьшить время выполнения программы. Так для размера данных в 10000 ускорение составляет около 80 раз при выполнении на 128 узлах. Очевидно, что при дальнейшем увеличении числа узлов будет достигнут порог масштабируемости программы, так как накладные расходы на создание процессов превысят выигрыш за счет увеличения их числа, также значительно увеличится число пересылок данных между процессами, что также можно считать накладными расходами. Важно заметить, что исходя из простого факта, заключающегося в том, что начальное время выполнения программы для каждого размера данных различно, вытекает следствие - для каждого размера данных порог насыщения свой. Так например для 10000 он составил 128 узлов, а для 4000 уже

64. Эти данные легко получаются, если пересечь трехмерный график плоскостью с фиксированным размером данных и посмотреть на сечение трехмерной поверхности, которая определит зависимость времени от количества узлов для данного размера входных данных.

Осуществим подсчет количества запусков программы на *Bluegene*, всего было произведено 270 запусков. Построение графиков осуществлялось при помощи языка *Python* (Приложение 3).

Тестирование программ на *Polus*

В данном пункте верны все тезисы предыдущего пункта, поэтому сразу перейдем к приведению результатов тестирования. Для получения удобной формы данных и построения графиков использовался скрипт на языке *Python* (Приложение 2).



Изображение 6 (Тестирование на *Polus*)

По данному графику видно (Изображение 6), что на системе *Polus* не получается достичь порог масштабируемости для больших входных данных. Но важно заметить, что в программе есть условие, что при малом количестве данных и большом кол-ве узлов используется столько узлов каково кол-во данных (Приложение 2). Для подкрепления графических данных предлагаем рассмотреть таблицу (Изображение 7).

Count of threads	Dataset size	Execute time	Count of threads	Dataset size	Execute time
1.0000	40.0000	0.0000	**8.0000**	2000.0000	0.0224
1.0000	120.0000	0.0004	**8.0000**	4000.0000	0.0664
1.0000	400.0000	0.0028	**8.0000**	10000.0000	0.4170
1.0000	2000.0000	0.0920	**16.0000**	40.0000	0.0024
1.0000	4000.0000	0.3889	**16.0000**	120.0000	0.0021
1.0000	10000.0000	2.4951	**16.0000**	400.0000	0.0035
2.0000	40.0000	0.0001	**16.0000**	2000.0000	0.0203
2.0000	120.0000	0.0002	**16.0000**	4000.0000	0.0606
2.0000	400.0000	0.0022	**16.0000**	10000.0000	0.2709
2.0000	2000.0000	0.0533	**32.0000**	40.0000	0.0034
2.0000	4000.0000	0.2216	**32.0000**	120.0000	0.0038
2.0000	10000.0000	1.2896	**32.0000**	400.0000	0.0053
4.0000	40.0000	0.0005	**32.0000**	2000.0000	0.0161
4.0000	120.0000	0.0005	**32.0000**	4000.0000	0.0513
4.0000	400.0000	0.0019	**32.0000**	10000.0000	0.2216
4.0000	2000.0000	0.0290	**64.0000**	40.0000	0.0053
4.0000	4000.0000	0.0983	**64.0000**	120.0000	0.0060
4.0000	10000.0000	0.6323	**64.0000**	400.0000	0.0059
8.0000	40.0000	0.0014	**64.0000**	2000.0000	0.0223
8.0000	120.0000	0.0007	**64.0000**	4000.0000	0.0447
8.0000	400.0000	0.0027	**64.0000**	10000.0000	0.1207

Изображение 7 (Тестирование на *Polus*)

По таблице с Изображения 7 и графику с Изображения 6 легко видеть пределы масштабирования программ для разных размеров данных, используя тезисы описанные в предыдущем пункте рассмотрим лишь частный пример: для размера данных 40 предел масштабируемости - это 1 узел, для 120 - это 2, для 400 - это 4. Причина этого очевидна и описана в предыдущем пункте, но поскольку она важна в контексте данной работы, опишем ее еще раз. Существование предела масштабируемости программы определяется тем, что накладные расходы могут превышать выигрыш за счет увеличения количества узлов, этот предел непосредственно зависит от количества входных данных.

Общее число тестирований составило 210 раз.

Анализ полученных результатов

Из проведенных исследований результаты которых приведены в предыдущем разделе можно сделать множество выводов.

Скорость выполнения программы перестает линейно возрастать в зависимости от количества процессов на которых выполняется программа по причине того, что накладные расходы на создание порции процессов значительно увеличиваются, и проанализировав результаты выполнения можно заметить, что при меньших размерах данных программа перестает ускоряться при меньшем числе потоков. Это подтверждает идею о том, что накладные расходы на создание процессов и на произведение рассылки данных между процессами - основная причина недостаточной масщтабируемости программы.

Анализируя графики и находя минимумы времени в проекции трехмерного графика на плоскость с соответствующим размером данных можно установить оптимальное число потоков для данного размера данных.

Важным результатом проведенных исследований можно считать, что технология MPI позволяет без больших временных затрат получить значительный прирост производительности программы, при имеющихся вычислительных ресурсах. Так лучший результат был получен на размере входных дынных в 10000, и он составил ускорите в 80 на 128 процессах (узлах) машины *Bluegene*.

Выводы

Работа по улучшению и разработке параллельной версии программы при помощи средств *MPI* позволила значительно ускорить полученную версию программы и провести ряд экспериментов с такими компьютерами как *Polus* и *Bluegene.*

MPI - это крайне удобная в использовании технология, которая позволяет быстро получить качественный результат и значительный прирост производительности на различных системах.

В сравнении с OpenMP данная технология позволяет использовать более сложные в устройстве машины, которые имею большую распределенность и вычислительную мощность, но очевидно, что эти технологии предназначены для разного уровня распределения программы, по вычислительным процессам или по нитям. Из вышеперечисленного следует, что достаточно просто использовать данные технологии вместе, что даст отличный прирост производительности. Сравнить данные методы численно, по метрикам затраченного времени на выполнения программы трудно, так как наибольшую эффективность они показывают на разных системах, с разным устройством и архитектурой.

Работа с *MPI* позволяет зародить интерес к параллельному программированию и изучению новых технологий из этой сферы деятельности.

Приложение 1

Файл с программой приложен к отчету *(gemver.c, gemver.h)*.

Приложение 2

Файлы скриптов построения результатов *(make_res.py, parse.py)*.

Приложение 3

Файл ноутбука с таблицами всех замеров и построением графиков приложен к отчету *(pad_MPI.ipynb)*.