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 2 Matrix Multiplications с использованием технологии MPI.

Отчет

студента 325 группы

факультета ВМК МГУ

Хайбулаева Глеба Сергеевича

2019 год

Постановка задачи

Распараллелить предложенную реализацию программы 2 Matrix Multiplications с использованием технологии MPI, а затем исследовать масштабируемость полученной программы, построить графики зависимости времени её выполнения от числа используемых ядер и объёма входных данных. Программа получает на вход 4 числа – размеры четырех матриц A, B, C и D и на основе этих данных вычисляет выражение 1.5*A*B*C + 1.2*D.

Код программы

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <math.h>
4.  #include <stdlib.h>
5.  #include <mpi.h>
6.  static
7.  void init_array(int ni, int nj, int nk, int nl,
8.                  double *alpha,
9.                  double *beta,
10.                 double A[ ni][nk],
11.                 double B[ nk][nj],
12.                 double C[ nj][nl],
13.                 double D[ ni][nl],
14.                 double tmp[ ni][nj])
15. {
16.    int i, j;
17.
18.    *alpha = 1.5;
19.    *beta = 1.2;
20.    for (i = 0; i < ni; i++)
21.       for (j = 0; j < nk; j++)
22.          A[i][j] = (double) ((i*j+1) % ni) / ni;
```

```
23.    for (i = 0; i < nk; i++)
24.        for (j = 0; j < nj; j++)
25.            B[i][j] = (double) (i*(j+1) % nj) / nj;
26.    for (i = 0; i < nj; i++)
27.        for (j = 0; j < nl; j++)
28.            C[i][j] = (double) ((i*(j+3)+1) % nl) / nl;
29.    for (i = 0; i < ni; i++)
30.        for (j = 0; j < nl; j++)
31.            D[i][j] = (double) (i*(j+2) % nk) / nk;
32.    for (i = 0; i < ni; i++)
33.        for (j = 0; j < nj; j++)
34.            tmp[i][j] = 0.0;
35. }
36.
37. static
38. void print_array(int ni, int nl,
39.                 double D[ ni][nl]) {
40.    int i, j;
41.    fprintf(stderr, "==BEGIN DUMP_ARRAYS==\n");
42.    fprintf(stderr, "begin dump: %s\n", "D");
43.    for (i = 0; i < ni; i++) {
44.        for (j = 0; j < nl; j++) {
45.            fprintf(stderr, "%0.2lf ", D[i][j]);
46.        }
47.        fprintf(stderr, "\n");
48.    }
49.    fprintf(stderr, "\nend   dump: %s\n", "D");
50.    fprintf(stderr, "==END   DUMP_ARRAYS==\n");
```

```
51. }
52.
53. static
54. void kernel_2mm(int ni, int nj, int nk, int nl,
55.                 double alpha,
56.                 double beta,
57.                 int rank,
58.                 int max_row,
59.                 double tmp[ ni][nj],
60.                 double A[ ni][nk],
61.                 double B[ nk][nj],
62.                 double C[ nj][nl],
63.                 double D[ ni][nl])
64. {
65.     int i, j, k;
66.     for (i = rank * max_row; i < max_row + rank * max_row; i ++) {
67.         for (j = 0; j < nj; j++) {
68.             tmp[i][j] = 0.0;
69.             for (k = 0; k < nk; ++k) {
70.                 tmp[i][j] += alpha * A[i][k] * B[k][j];
71.             }
72.         }
73.     }
74.     MPI_Gather(&tmp[rank*max_row], max_row*nj, MPI_DOUBLE, tmp, max_row*nj, MPI_DOUBLE, 0,
    MPI_COMM_WORLD);
75.     MPI_Barrier(MPI_COMM_WORLD);
76.
77.     for (i = rank * max_row; i < max_row + rank * max_row; i ++) {
```

```
78.       for (j = 0; j < nl; j++) {
79.           D[i][j] *= beta;
80.           for (k = 0; k < nj; ++k)
81.               D[i][j] += tmp[i][k] * C[k][j];
82.       }
83.   }
84.   MPI_Gather(&D[rank*max_row], max_row*nl, MPI_DOUBLE, D, max_row*nl, MPI_DOUBLE, 0,
      MPI_COMM_WORLD);
85.   MPI_Barrier(MPI_COMM_WORLD);
86. }
87.
88. int main(int argc, char** argv)
89. {
90.   int err = MPI_Init(&argc, &argv);
91.   if (err != MPI_SUCCESS)
92.   {
93.       fprintf(stderr, "Error while starting! \n");
94.       MPI_Abort(MPI_COMM_WORLD, err);
95.   }
96.
97.   int size, rank = 0;
98.   MPI_Comm_size(MPI_COMM_WORLD, &size);
99.   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
100.
101.              if (!rank) {
102.                  printf("Number of threads: %d\n", size);
103.              }
104.
```

```
105.        int nis[5] = {16, 40, 180, 800, 1600};
106.        int njs[5] = {18, 50, 190, 900, 1800};
107.        int nks[5] = {22, 70, 210, 1100, 2200};
108.        int nls[5] = {24, 80, 220, 1200, 2400};
109.        char *names[5] = {"MINI", "SMALL", "MEDIUM", "LARGE", "EXTRALARGE"};
110.        int ni;
111.        int nj;
112.        int nk;
113.        int nl;
114.        MPI_Barrier(MPI_COMM_WORLD);
115.
116.        for (int i = 0; i < 5; i++) {
117.            ni = nis[i];
118.            nk = nks[i];
119.            nj = njs[i];
120.            nl = nls[i];
121.
122.            double alpha;
123.            double beta;
124.            double (*tmp)[ni][nj] = NULL;
125.            double (*A)[ni][nk] = NULL;
126.            double (*B)[nk][nj] = NULL;
127.            double (*C)[nj][nl] = NULL;
128.            double (*D)[ni][nl] = NULL;
129.            tmp = (double (*)[ni][nj]) malloc((ni) * (nj) * sizeof(double));
130.            A = (double (*)[ni][nk]) malloc((ni) * (nk) * sizeof(double));
131.            B = (double (*)[nk][nj]) malloc((nk) * (nj) * sizeof(double));
132.            C = (double (*)[nj][nl]) malloc((nj) * (nl) * sizeof(double));
```

```
133.        D = (double (*)[ni][nl]) malloc((ni) * (nl) * sizeof(double));
134.
135.        init_array(ni, nj, nk, nl, &alpha, &beta,
136.              *A,
137.              *B,
138.              *C,
139.              *D,
140.              *tmp);
141.        double start = MPI_Wtime();
142.        int max_row = ni/size;
143.        kernel_2mm(ni, nj, nk, nl,
144.              alpha, beta,
145.              rank, max_row,
146.              *tmp,
147.              *A,
148.              *B,
149.              *C,
150.              *D);
151.
152.        double end = MPI_Wtime();
153.        if (rank == 0){
154.           printf("Dataset %s:\nTime = %fs\n", names[i], end-start);
155.           fflush(stdin);
156.        }
157.
158.        if (argc > 42 && !strcmp(argv[0], "")) print_array(ni, nl, *D);
159.        MPI_Barrier(MPI_COMM_WORLD);
160.        if (tmp != NULL)
```

```
161.                free((void *) tmp);
162.              tmp = NULL;
163.              if (A != NULL)
164.                free((void *) A);
165.              A = NULL;
166.              if (B != NULL)
167.                free((void *) B);
168.              B = NULL;
169.              if (C != NULL)
170.                free((void *) C);
171.              C = NULL;
172.              if (D != NULL)
173.                free((void *) D);
174.              D = NULL;
175.          }
176.          MPI_Finalize();
177.          return 0;
178.      }
```

Распараллелена программа классическими приёмами, рассмотренными на лекции.

Результаты замеров времени выполнения

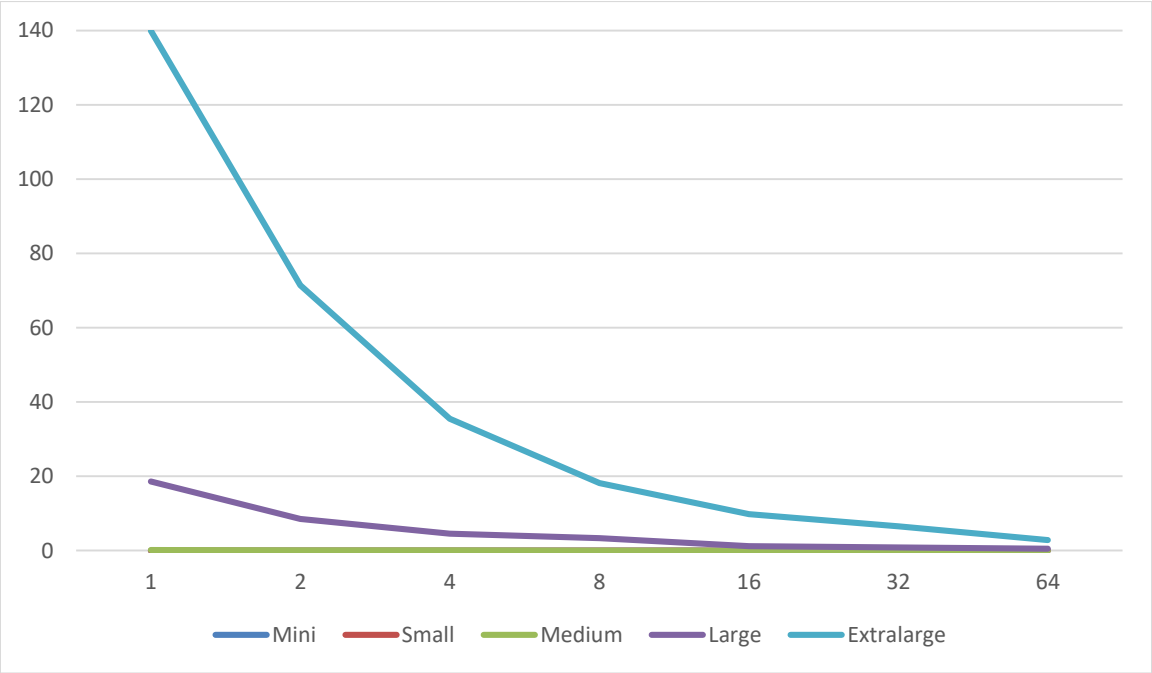
Работа задачи рассмотрена на суперкомпьютере Polus с различным числом нитей (1 - 64) и различными наборами данных, предоставленных вместе с исходным кодом (см ниже). Каждое измерение проводилось 6 раз. В таблице и на графиках записаны усредненные результаты времени выполнения.
Наборы данных:
Mini – 16, 18, 22, 24
Small – 40, 50, 70, 80
Medium – 180, 190, 210, 220
Large – 800, 900, 1100, 1200
Extralarge – 1600, 1800, 2200, 2400

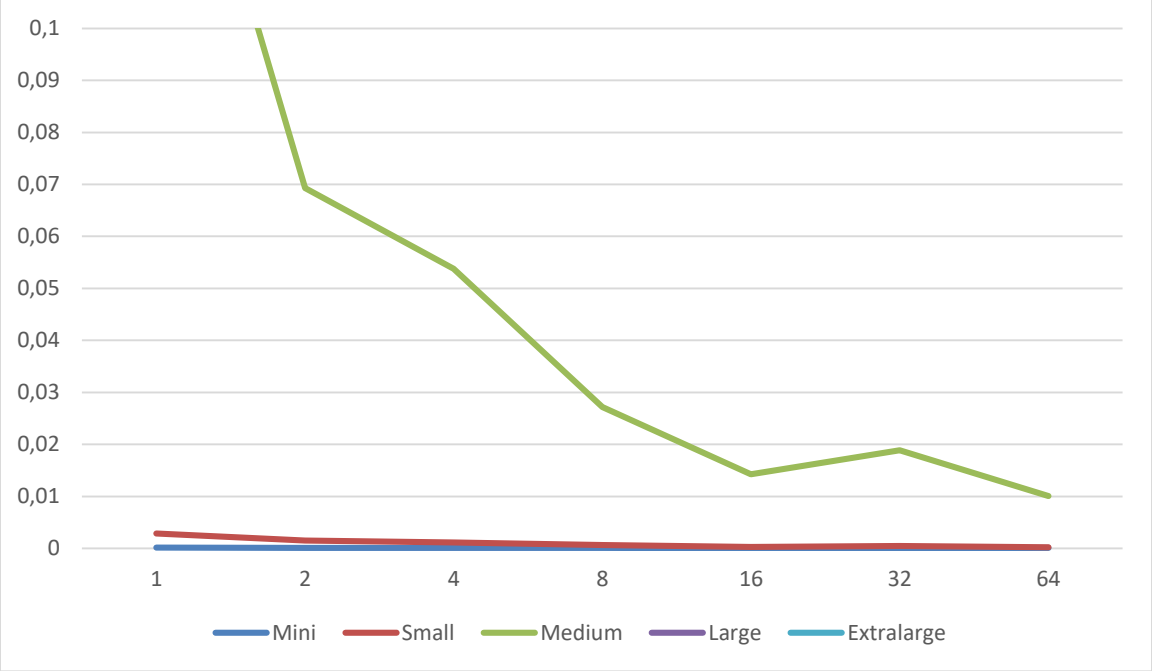
Таблица с результатами

Нити/размеры матриц	Mini	Small	Medium	Large	Extralarge
1	0.000151	0.002874	0.166226	18.561344	139.938766
2	0.000095	0.001511	0.069315	8.515160	71.453593
4	0.000085	0.001124	0.053768	4.528845	35.470738
8	0.000068	0.000660	0.027161	3.303941	18.185954
16	0.000049	0.000285	0.014227	1.193156	9.835589
32	0.000059	0.000482	0.018878	0.822930	6.493348
64	0.000084	0.000203	0.010056	0.525736	2.835041

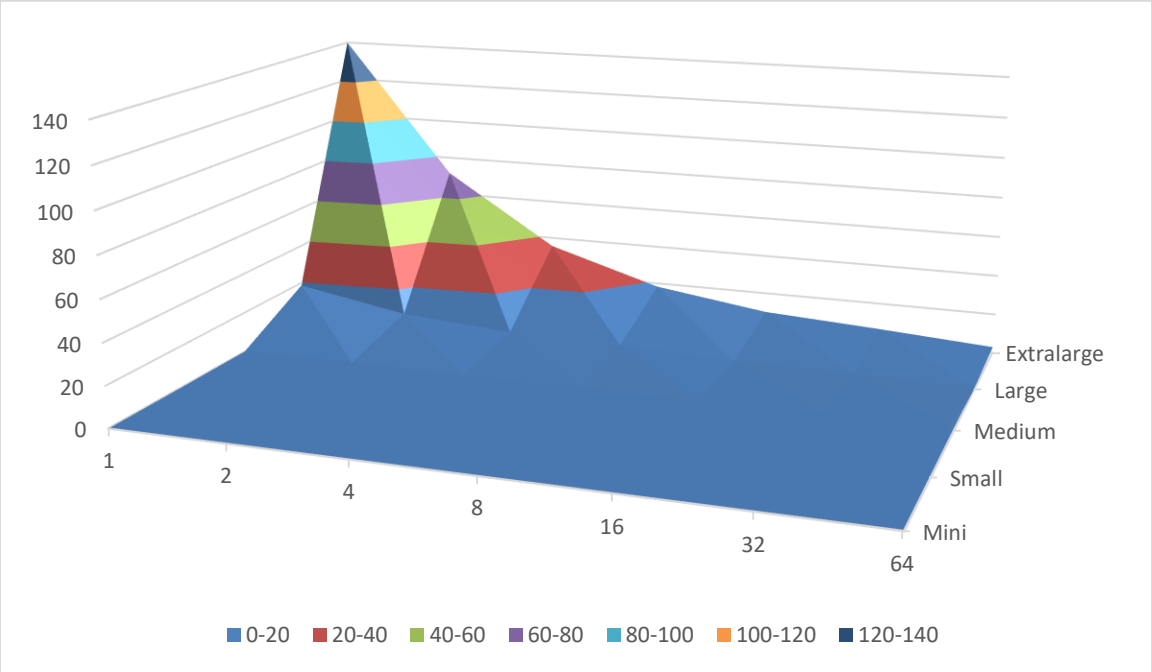
Графики: время выполнения программы в зависимости от размеров матриц и количества потоков

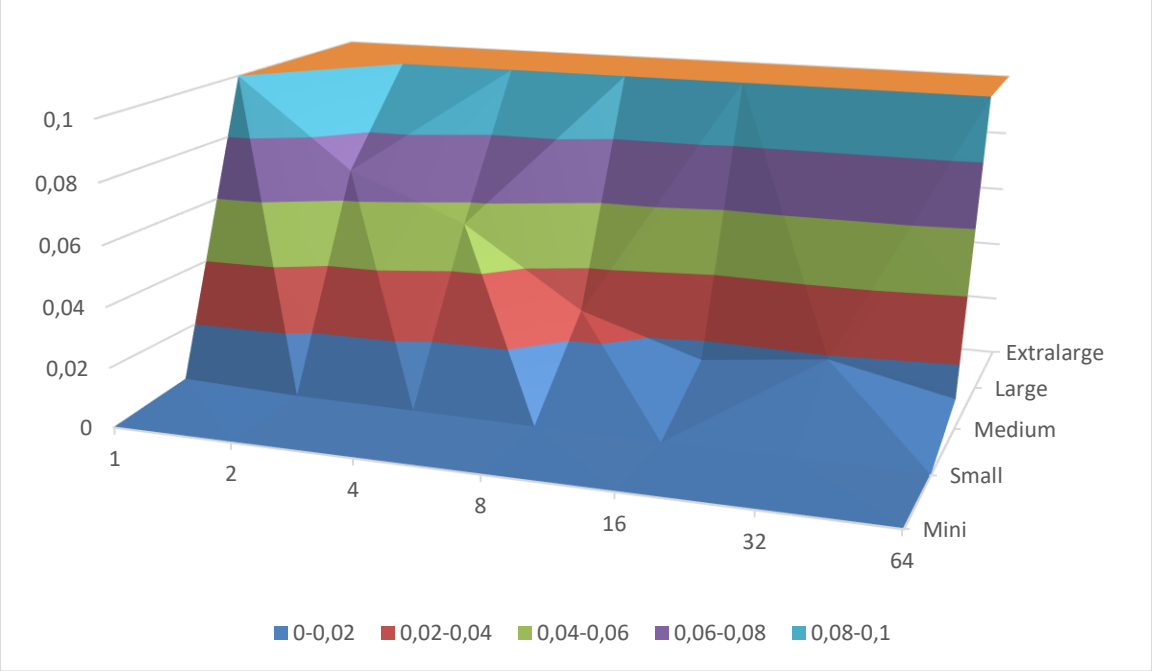
В виде линий:



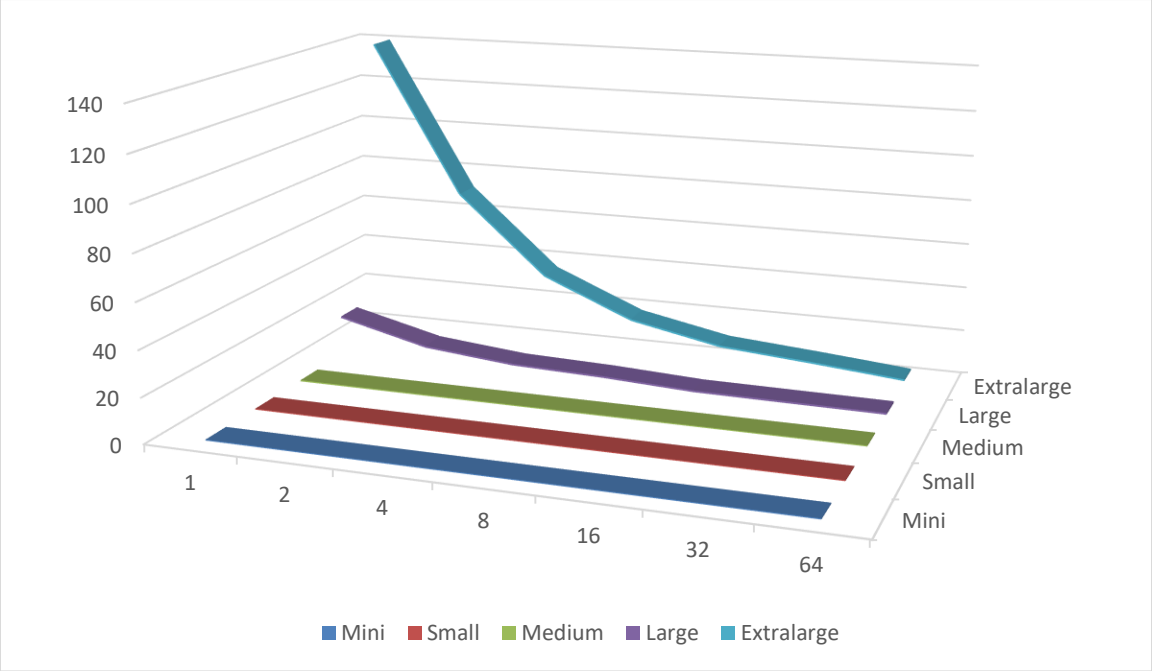


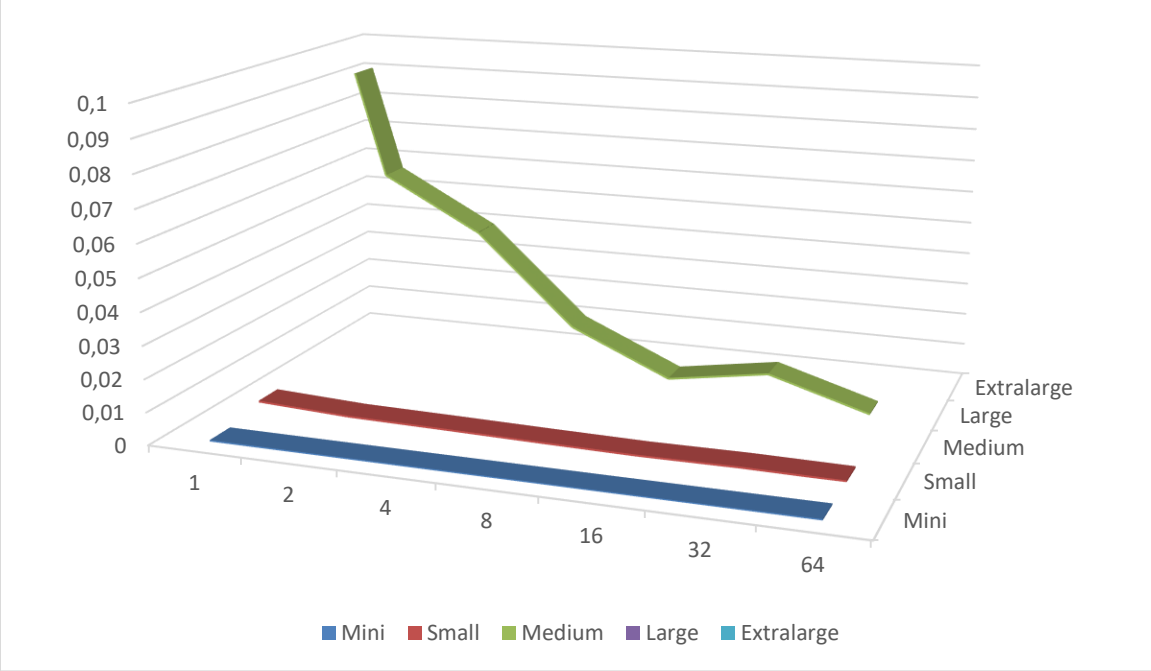
В виде поверхности:





В виде линий в трехмерном пространстве:





Вывод:

Распараллеливание программы дало выигрыш около 10 по времени раз на массивах небольшого размера (Mini, Small, Medium) и до 50 раз на массивах большого размера (Large, Extralarge).

Как показали эксперименты, в отличие от технологии OpenMP, MPI не создает больших накладных расходов при увеличении числа нитей, что позволяет получать улучшенный результат даже при малых наборах данных. Вместе с этим при увеличении числа нитей существенно возрастает производительность, позволяя быстро выполнять программу даже при больших наборах данных.

Отсюда следует вывод, что MPI дает прирост производительности даже при небольших наборах данных и большом числе нитей. Исходя из этого можно сказать, что, в отличие от OpenMP, использование MPI оправдано при любых наборах данных, даже при большом числе нитей.