МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В.Ломоносова

Факультет вычислительной математики и кибернетики

Практикум по курсу

"Суперкомпьютеры и параллельная обработка данных"

Разработка параллельной версии программы, включающей в себя умножение векторов и сложение матриц

ОТЧЕТ

о выполненном задании

студента 320 учебной группы факультета ВМК МГУ

Грибов Ильи Юрьевича

Москва
2022

Содержание

1 Постановка задачи

1. Реализовать алгоритм параллельного подсчета ядра gemver с использованием OpenMP и MPI.
2. Исследовать зависимость времени выполнения программы от размера входных данных и числа используемых потоков.
3. Построить графики зависимости времени исполнения от числа потоков для различного объёма входных данных.
4. Сравнить эффективность OpenMP и MPI-версий параллельной программы..

2 Описание алгоритма и код программы

Ниже представлена реализация самого наивного алгоритма с использованием вложенных циклов for.

Математически опишем производимые алгоритмом операции.

1. A = A + u_1 * v_1 + u_2 * v_2, где A - это матрица размера n × n, а u_1 и u_2 - это столбы высоты n, v_1 и v_2 - это строки длины n.
2. x = x + β * A * y, где A - это матрица размера n × n , а x и y - это вектор столбцы высоты n, β - вещественный коэффициент.
3. x = x + z, где x и z - это вектор столбцы высоты n.
4. w = w + α * A * x, где A - это матрица размера n × n, а w и x - это вектор столбцы высоты n, α - вещественный коэффициент.

Ниже представлен код на языке C.

```
        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[i] = x[i] + beta * A[j][i] * y[j];
            }
        }

        for (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];
            }
        }
```

2.1 Параллельный алгоритм

Для начала обратим внимание на то, что массив A во втором цикле обходится в неправильном порядке, что в свою очередь не позволяет работать КЭШу. Для того чтобы исправить эту проблему было принято решение поменять местами индексы проходов во 2 цикле.

```
        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 (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];
            }
        }
```

Однако так как цикл, в котором был изменен порядок обхода теперь создает зависимость: на каждом шаге все нити в возможной параллельной области будут одновременно работать с массивом x и на чтение и на запись, было принято решение перейти к блочному выполнению второго цикла.

Экспериментальным путем было выяснено, что для последовательной программы лучше всего подходит размер блока равный размеру данных, это можно объяснить большим количеством накладных расходов и большим размером КЭШа на машине в которой производилась проверка.

Итоговый код для распараллеливания выглядит следующим образом:

```
        int BLOCK_SIZE = n;
        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 j1 = 0; j1 < n; j1 += BLOCK_SIZE) {
            for (int i = 0; i < n; ++i) {
                for (int j2 = 0; j2 < min(BLOCK_SIZE, n - j1); ++j2) {
                    x[j1 + j2] = x[j1 + j2] + beta * A[i][j1 + j2] * 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];
            }
        }
```

Наиболее важным отличием этого кода является то, что в нем совершенно очевидно можно распределить вычисления в каждом цикле, так как мы четко и ясно избавились от зависимости между нитями, по которым будут разделяться итерации внешенго цикла.

Дальнейшая модификация кода сводится к добавлению кляуз **pragma omp for** и **pragma omp parallel** в нужных местах.

2.2 OpenMP-версия

```
    if (n < 400){
        omp_set_num_threads(1);
    }
    #pragma omp parallel
    {
        int BLOCK_SIZE = n / omp_get_num_threads();
```

```
        #pragma omp for
        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];
            }
        }

        #pragma omp for
        for (int j1 = 0; j1 < n; j1 += BLOCK_SIZE) {
            for (int i = 0; i < n; ++i) {
                for (int j2 = 0; j2 < min(BLOCK_SIZE, n - j1); ++j2) {
                    x[j1 + j2] = x[j1 + j2] + beta * A[i][j1 + j2] * y[i];
                }
            }
        }

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

        #pragma omp for
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                w[i] = w[i] + alpha * A[i][j] * x[j];
            }
        }
    }
```

А также при инициализации матрицы и всех векторов

```
    #pragma omp parallel for
    for (int 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 (int j = 0; j < n; j++) {
            A[i][j] = (double) (i*j % n) / n;
        }
    }
```

Стоит так же отметить, что размер блока в данном случае был выбран уже n/Th, где Th - это количество нитей в параллельной области. Данный выбор можно обосновать эмпирически. Кроме того, для малого размера данных нерационально создавать множество нитей, поэтому вычисление маленьких наборов данных происходит последовательно.

2.3 MPI-версия

MPI-версия выглядит следующим образом:

В функции void kernel gemver(...)

```
    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++) {                          //for use *
    comment it
            MPI_Send(x, n, MPI_DOUBLE, i, 13, MPI_COMM_WORLD);        //for use *
    comment it
        }                                                             //for use *
    comment it
    } 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]); //for use *
    comment it
    }
    MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD); //*
```

В функции main(...)

```
    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();
```

```
51              }
52
53              kernel_gemver (n, size, alpha, beta,
54                             *A,
55                             *u1,
56                             *v1,
57                             *u2,
58                             *v2,
59                             *w,
60                             *x,
61                             *y,
62                             *z);
63
64              if (myrank == 0) {
65                  size = k + (0 < m);
66                  for (int k = 0; k < size; k++) {
67                      printf("%lf\n", (*w)[k]);
68                  }
69                  double (*cur)[size]; cur = (double(*)[size])malloc ((size) * sizeof(
    double));
70                  for (int j = 1; j < use_proc; j++) {
71                      size = k + (j < m);
72                      MPI_Recv(*cur, size, MPI_DOUBLE, j, 13, MPI_COMM_WORLD, &status[0]);
73                      for (int k = 0; k < size; k++) {
74                          printf("%lf\n", (*cur)[k]);
75                      }
76                  }
77                  printf("MPI --- %lf\n", MPI_Wtime() - time_1);
78              } else {
79                  MPI_Send(w, size, MPI_DOUBLE, 0, 13, MPI_COMM_WORLD);
80              }
81
82              free((void*)A);
83              free((void*)u1);
84              free((void*)v1);
85              free((void*)u2);
86              free((void*)v2);
87              free((void*)w);
88              free((void*)x);
89              free((void*)y);
90              free((void*)z);
91              break;
92          }
93      }
94      MPI_Finalize();
```

3 Тестирование программы

3.1 OpenMP

В этом разделе представлены результаты тестирования, описанных в предыдущем разделе, программ и трехмерные графики зависимостей. Тестирование производилось на компьютере Polus.

Конфигурации запущенных программ для OpenMP:

- 1, 4, 8, 16, 32, 64 потоков
- 40, 120, 400, 2000, 4000, 10000, 20000 размер данных

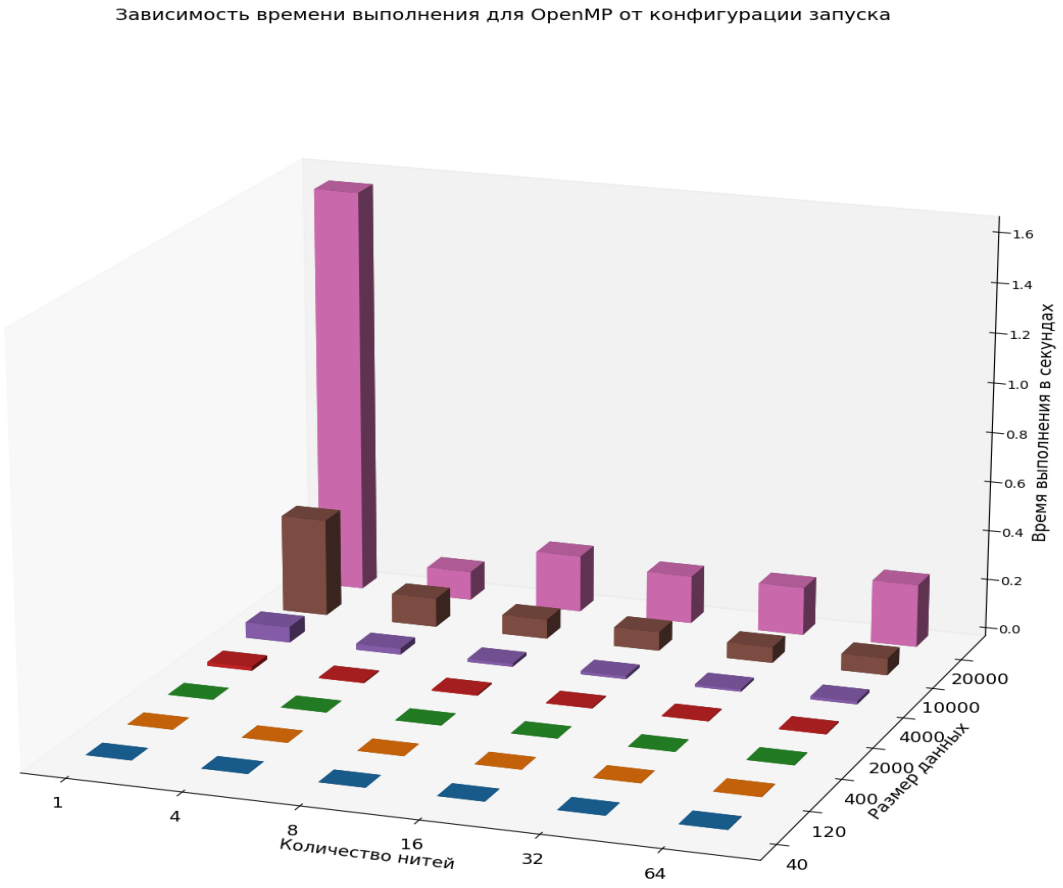
Каждая конфигурация была запущена 5 раз. Ниже приведены усредненные результаты.

N \ threads	40	120	400	2000	4000	10000	20000
1	0.000009	0.000058	0.000623	0.015659	0.063031	0.392101	1.627662
4	0.000011	0.000040	0.000237	0.004425	0.024110	0.116007	0.412000
8	0.000022	0.000069	0.000297	0.005191	0.012988	0.078194	0.228573
16	0.000079	0.000132	0.000284	0.003303	0.012298	0.076301	0.192419
32	0.000291	0.000307	0.000372	0.002774	0.009800	0.063658	0.193400
64	0.002062	0.001960	0.001642	0.003848	0.012367	0.067495	0.253308

Таблица, отражающая во сколько раз ускорилась программа.

	Во сколько раз ускорилась программа						
N \ threads	40	120	400	2000	4000	10000	20000
1	1	1	1	1	1	1	1
4	0,81	1,45	2,628691983	3,538757062	2,614309415	3,37997707	3,950635922
8	0,41	0,8405797101	2,097643098	3,016567135	4,853018171	5,014464025	7,120972293
16	0,11	0,4393939394	2,193661972	4,740841659	5,125304928	5,13887105	8,458946362
32	0,03	0,1889250814	1,674731183	5,644917087	6,431734694	6,159492915	8,416039297
64	0,004	0,02959183673	0,3794153471	4,069386694	5,096708984	5,809334025	6,425624141

График, отражающий зависимость времени выполнения программы от различных входных данных и числа процессов.



3.2 MPI

Конфигурации запущенных программ для MPI:

- 1, 4, 8, 16, 32, 48 потоков
- 40, 120, 400, 2000, 4000, 10000, 20000 размер данных

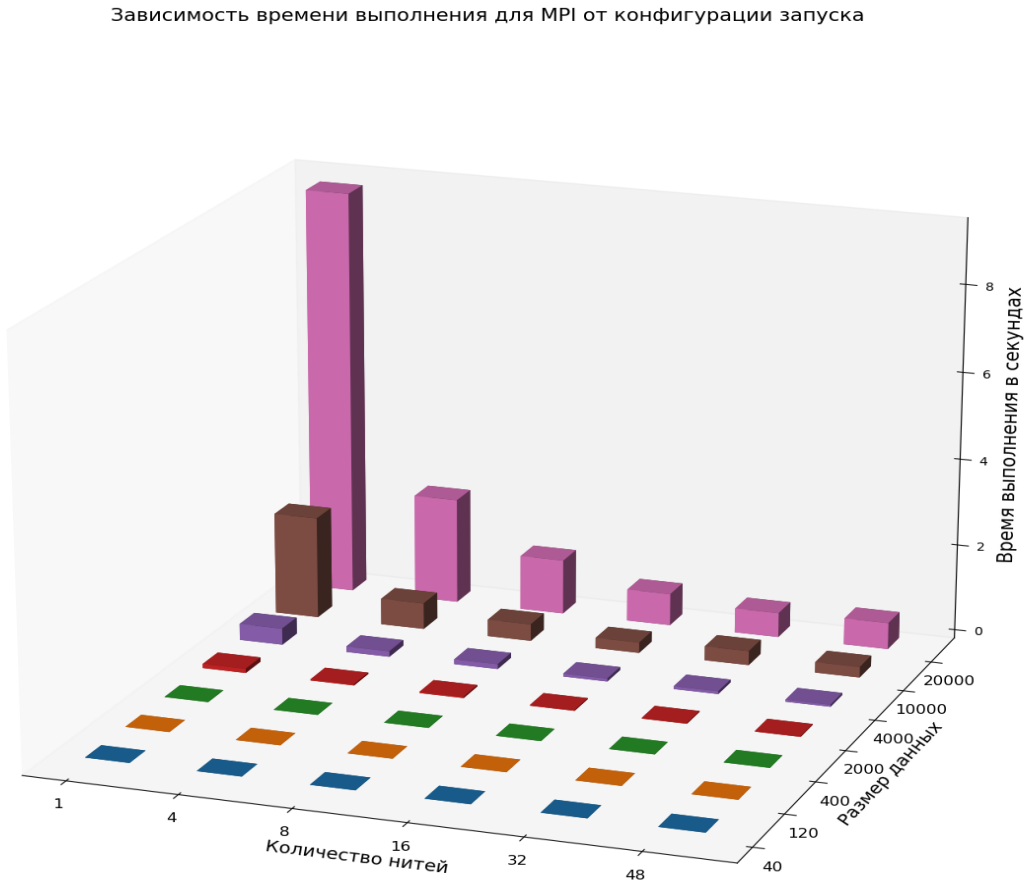
Каждая конфигурация была запущена 5 раз. Ниже приведены усредненные результаты.

	threads						
N		40	120	400	2000	4000	10000	20000
	1	0.000139	0.001381	0.004596	0.096812	0.376207	2.339288	9.310144
	4	0.000393	0.001058	0.001914	0.028520	0.128214	0.601240	2.421311
	8	0.000518	0.001158	0.002335	0.030531	0.111806	0.386254	1.255149
	16	0.001515	0.001346	0.002642	0.020081	0.067433	0.240124	0.733342
	32	0.001140	0.001935	0.003565	0.018153	0.064578	0.325905	0.561925
	48	0.001590	0.002612	0.003441	0.017048	0.046740	0.242333	0.607289

Таблица, отражающая во сколько раз ускорилась программа.

		Во сколько раз ускорилась программа						
	threads							
N		40	120	400	2000	4000	10000	20000
	1	1	1	1	1	1	1	1
	4	0,3536895674	1,305293006	2,401253918	3,394530154	2,934211553	3,890772404	3,845083924
	8	0,2683397683	1,192573402	1,968308351	3,170941011	3,364819419	6,056346342	7,417560784
	16	0,09174917492	1,026002972	1,739591219	4,821074648	5,578974686	9,741999967	12,69550087
	32	0,1219298246	0,7136950904	1,289200561	5,333112984	5,825621729	7,177821758	16,5683036
	64	0,08742138365	0,5287136294	1,335658239	5,678789301	8,048930252	9,653196222	15,33066464

График, отражающий зависимость времени выполнения программы от различных входных данных и числа процессов.



4 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 потоков на которых выполняется программа по причине того, что накладные расходы на создание порции нитей 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 нитей - основная причина недостаточной масштабируемости программы.

Блочный алгоритм позволяет очень хорошо ускорить программу и избежать зависимости по данным при параллельном выполнении 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. Например, если рассматривать Polus и размер данных равный 20000 для MPI, то можно заметить, что минимум времени в проекции трехмерного графика на плоскость соответствующую размеру данных равному 20000 достигается при 32 потоках.

5 Выводы

1. Работа по улучшению и разработке параллельной версии программы при помощи средств OpenMP позволила значительно ускорить полученную версию программы и провести ряд экспериментов с таким компьютером как Polus.

2. В целом OpenMP позволил эффективнее распараллелить исходную программу чем MPI с точки зрения конечного результата. Однако с точки зрения эффективности распараллеливания MPI оказался лучше чем OpenMP. Это можно утверждать, опираясь на графики и таблицы.

3. OpenMP и MPI - это крайне удобные в использовании технологии, которые позволяют быстро получить качественный результат и получить значительный прирост производительности.