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

ОТЧЕТ

По курсу «Параллельная обработка данных»

Практическое задание.

Работу выполнил:
Студент 4 курса группы 441/2
Факультета вычислительной математики и кибернетики
Цирунов Леонид Александрович

Москва – 2022

Задача

Реализовать параллельную версию ***алгоритма Гаусса*** решения СЛАУ при помощи технологии **MPI**.

Исследовать масштабируемость полученной параллельной программы: построить графики зависимости времени выполнения программы от числа процессов (ядер) и размерности входной матрицы.

Для каждого набора входных данных найти количество процессов (ядер), при котором время выполнения задачи перестаёт уменьшаться.

Определить основные причины недостаточной масштабируемости программы при максимальном числе используемых процессов (ядер).

Сравнить эффективность распараллеливания программы средствами ***OpenMP*** и ***MPI***.

Описание алгоритма Гаусса

Алгоритм решения СЛАУ методом Гаусса состоит из прямого и обратного хода.

В процессе прямого хода система приводится к эквивалентной путем приведения матрицы системы к верхней треугольной форме. На i-м шаге в строке выбирается первый ненулевой элемент a_{ii} – ведущий, который существует в силу не вырожденности матрицы, после чего i-я строка делится на данный элемент. Затем из остальных $i + 1 \ldots n$ строк вычитается i-я строка, умноженная на $a_{i+1,i} \ldots a_{n,i}$ соответственно. Сложность прямого хода $Q_1 = \frac{n(n+1)}{2}$ делений и $Q_2 = \frac{n(n^2-1)}{3}$ сложений и умножений.

В процессе обратного хода последовательно определяются все неизвестные путем решения уравнений (с одной неизвестной) и подстановки решений из решенного в следующее (получая уравнение с одной неизвестной), процесс начинается с x_n и заканчивается на x_1. Сложность обратного хода: $Q_3 = \frac{n(n-1)}{2}$.

Общая сложность метода Гаусса: $Q = Q_2 + Q_3 = \frac{n^3}{3} + O(n^2)$.

Код программы

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <mpi.h>

void prt1a(char *t1, double *v, int n,char *t2);

int N;
```

```
double *A;
#define A(i,j) A[(i)*(N+1)+(j)]
double *X;
int *map;

int main(int argc,char **argv) {
        double time0, time1;
        int rank, nprocs;
        int i, j, k;
        /* create arrays */
        MPI_Init(&argc, &argv);
        for (N=100; N < 4501; N += 200) {
            MPI_Comm_rank(MPI_COMM_WORLD, &rank);
            MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
            A=(double *)malloc(N*(N+1)*sizeof(double));
            X=(double *)malloc(N*sizeof(double));
            map=(int *)malloc(N*sizeof(int));
            if (rank == 0) {
                printf("\n----------------------------------\nGAUSS %dx%d\n",N,N);
                /* initialize array A*/
                for(i=0; i <= N-1; i++)
                    for(j=0; j <= N; j++)
                        if (i==j || j==N)
                            A(i,j) = 1.f;
                        else
                            A(i,j)=0.f;
                time0=MPI_Wtime();
            }
            MPI_Bcast (A,N*(N+1),MPI_DOUBLE,0,MPI_COMM_WORLD);
            for (i=0; i<N; i++) {
                map[i] = i % nprocs;
            }
            /* elimination */
            for (i=0; i<=N-1; i++) {
                MPI_Bcast (&A(i,i),N-i+1,MPI_DOUBLE,map[i],MPI_COMM_WORLD);
                for (k=i+1; k <= N-1; k++) {
                    if (map[k] == rank) {
                        for (j=i+1; j <= N; j++)
                            A(k,j) -= A(k,i)*A(i,j)/A(i,i);
                    }
                }
            }
            if (rank == 0) {
```

```
                /* reverse substitution */
                        X[N-1] = A(N-1,N)/A(N-1,N-1);
                        for (j=N-2; j>=0; j--) {
                                for (k=0; k <= j; k++)
                                        A(k,N) = A(k,N)-A(k,j+1)*X[j+1];
                                X[j]=A(j,N)/A(j,j);
                        }
                        time1=MPI_Wtime();
                        printf("Time in seconds=%gs\n",time1-time0);
                        prt1a("X=(", X,N>100?100:N,"...)\n");
                }
                free(A);
                free(X);
                free(map);
        }
        if (rank == 0) {
                printf("\n");
        }
        MPI_Finalize();
        return 0;
}

void prt1a(char * t1, double *v, int n,char *t2) {
        int j;
        printf("%s",t1);
        for(j=0;j<n;j++)
                printf("%.4g%s",v[j], j%10==9? "\n": ", ");
        printf("%s",t2);
}
```

Анализ полученных данных

Запуск программы производился на машине Polus с использованием .lsf скрипта. Программа была скомпилирована различными компиляторами: mpicc и mpixlc, поэтому в отчете приведен анализ двух наборов данных.

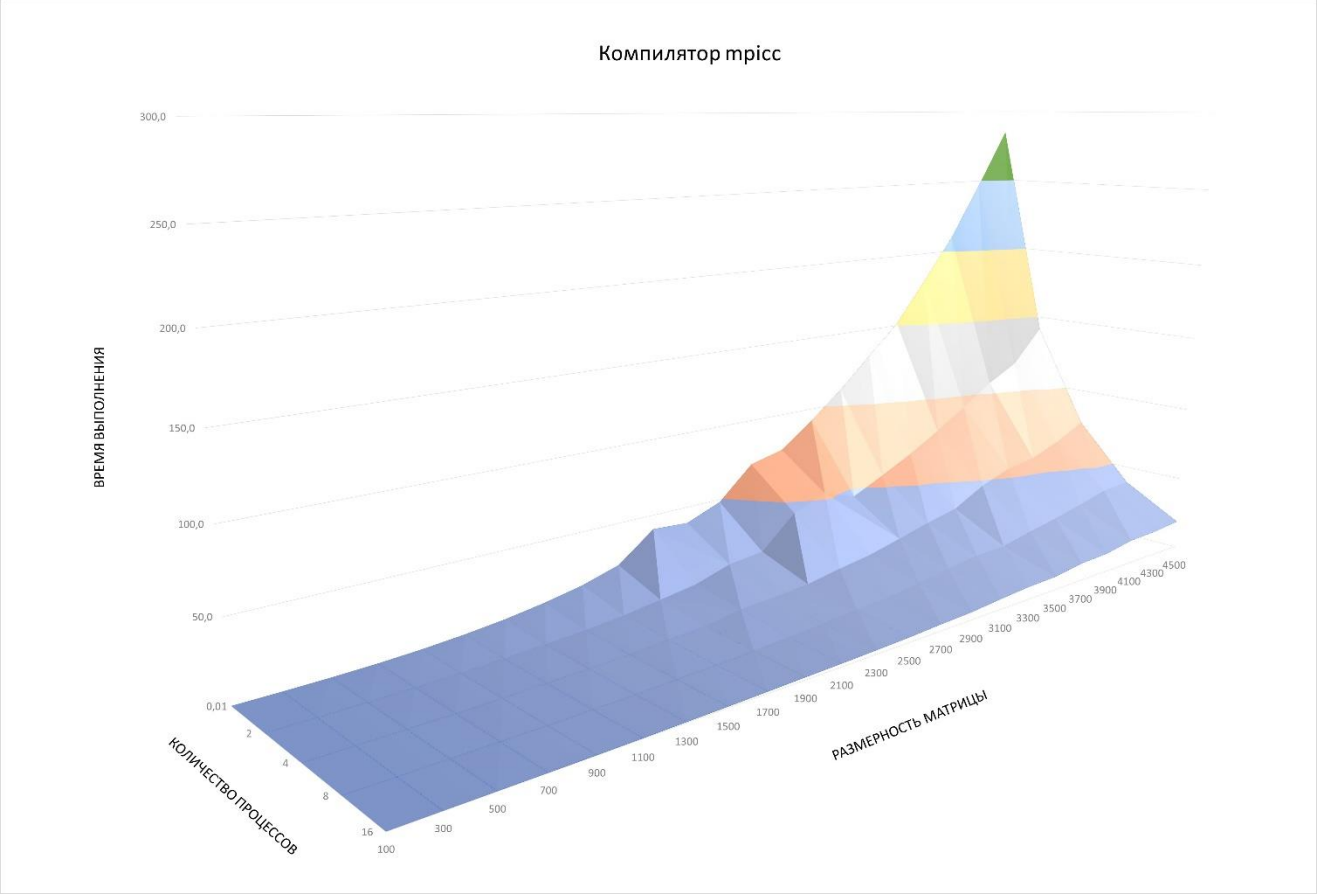
1. mpicc:

Таблица с данными:

	1	2	4	8	16
100	0,00308463	0,0026683	0,00180843	0,0018012	0,00392065
300	0,0813704	0,0646177	0,0527207	0,0263747	0,0267147
500	0,374894	0,222063	0,176682	0,0974792	0,109692
700	1,02673	0,527302	0,35396	0,215036	0,116643
900	2,22478	1,14683	0,828586	0,380589	0,216588
1100	4,1153	2,14906	1,42256	0,769735	0,297484
1300	6,74015	3,38123	2,18148	1,07421	0,564674
1500	10,4425	5,24413	3,07963	1,92397	0,811152
1700	15,2443	7,56328	4,82926	2,64704	1,17929
1900	22,2817	10,9425	5,95887	3,66789	1,63953
2100	40,5395	14,4107	9,95086	4,60739	2,02074
2300	39,0952	21,5508	11,2095	6,05291	2,57073
2500	48,1797	24,0568	13,1494	8,12122	3,21544
2700	69,4007	44,7111	17,1719	9,51667	4,3253
2900	75,0516	53,1340	20,5011	12,1561	5,09369
3100	91,6102	45,8362	25,7589	15,7208	6,92467
3300	110,044	55,9468	31,2487	18,5187	8,47938
3500	130,816	66,7273	35,8869	19,2972	9,04285
3700	151,387	79,0107	47,1604	23,7248	11,9629
3900	180,061	92,5793	54,5361	27,6619	12,6984
4100	209,633	106,735	57,9425	32,1998	16,0668
4300	246,354	117,877	65,5819	37,392	16,6473
4500	284,917	140,962	75,8207	38,6394	18,6021

(количество процессов (ядер) - по горизонтали, размерность матрицы – по вертикали)

График:



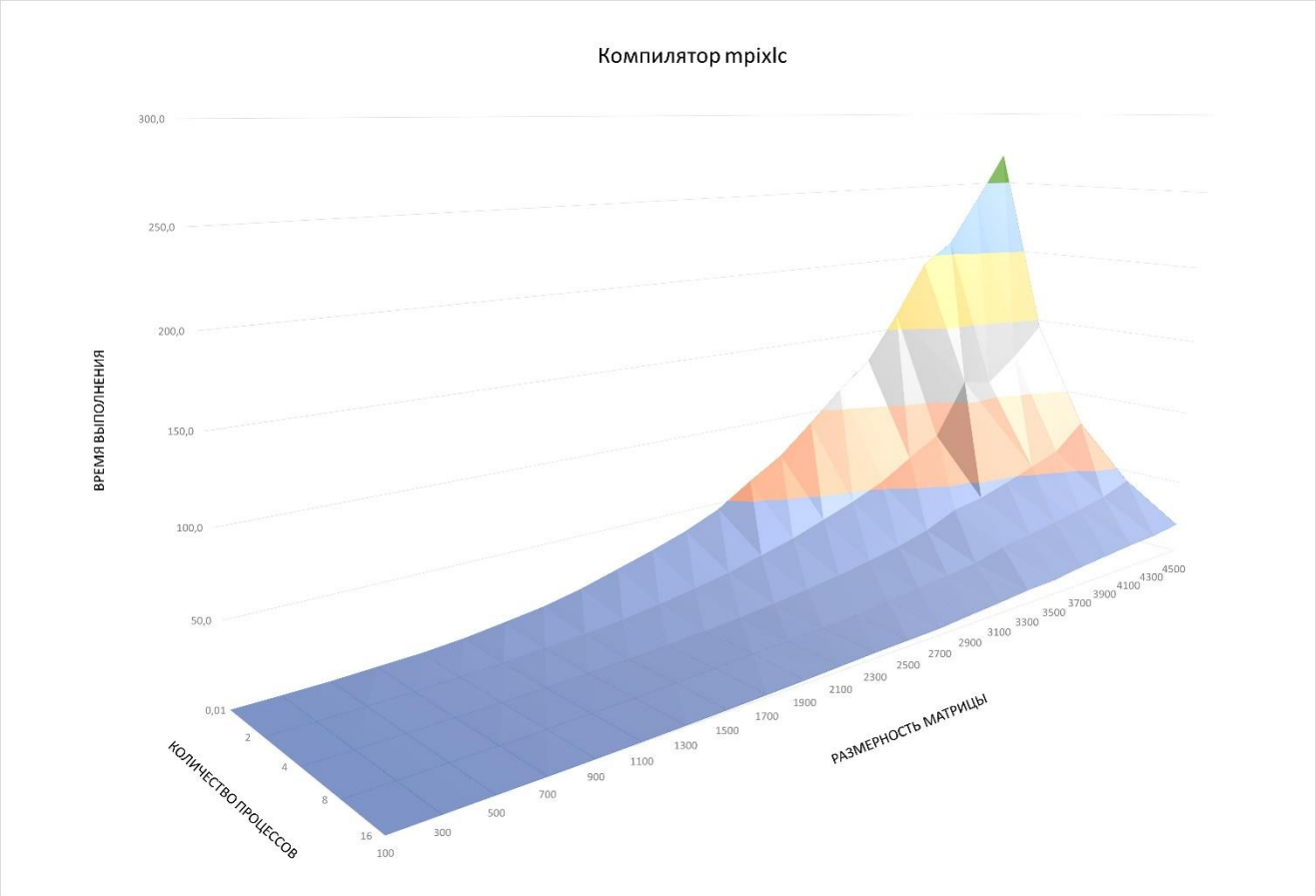
2. mpixlc:

Таблица с данными:

	1	2	4	8	16
100	0,00415576	0,00178297	0,00133558	0,00139559	0,00201979
300	0,117947	0,0503844	0,022081	0,0184587	0,0153005
500	0,397879	0,266639	0,0978995	0,079676	0,0625358
700	1,31551	0,522701	0,264035	0,211094	0,165732
900	2,18001	1,10663	0,55691	0,457161	0,266911
1100	4,07545	2,04373	1,03577	0,80684	0,42542
1300	7,14063	3,50360	1,72352	1,38021	0,686491
1500	10,501	5,10973	2,62341	2,07849	1,06108
1700	15,5153	7,49961	3,81722	2,96093	1,54389
1900	22,1313	10,4767	5,36879	4,35488	2,10114
2100	28,8028	14,1939	7,02731	4,71445	2,83458
2300	36,253	18,425	9,40685	5,79289	3,55094
2500	45,8873	24,081	12,0957	7,16041	3,92428
2700	60,3849	29,8475	15,0895	9,34946	4,30306
2900	73,581	37,2869	18,9082	11,8288	5,91905
3100	92,436	44,7983	23,0157	13,6552	7,2493
3300	112,147	53,6237	28,7217	16,1771	8,92284
3500	129,733	66,162	37,6256	21,0768	9,58313
3700	159,779	78,0327	41,0972	24,1604	12,0728
3900	194,125	113,436	47,3822	27,6061	13,7996
4100	207,918	109,913	53,9894	31,0384	15,5433
4300	238,199	125,812	61,0646	35,4716	16,6282
4500	269,97	145,252	78,055	42,5427	19,3087

(количество процессов (ядер) - по горизонтали, размерность матрицы – по вертикали)

График:



Промежуточные результаты:

1. При увеличении числа процессов уменьшается время работы алгоритма. Это

изменение нелинейно из-за того, что параллельно работает только прямой ход алгоритма Гаусса.

2. Использование компилятора mpixlc вместо mpicc не ускоряет программу, как это было с OpenMP версией. Временные результаты, можно сказать, равноценны.

3. В силу ограниченности ресурсов машины Polus, не удалость достигнуть количества процессов, для которых результат остановился на плато, либо же стал ухудшаться. Поэтому оценить этот момент сложно, так как на полученных данных видно, что временной результат улучшается при повышении количества с 1 процесса до 16.

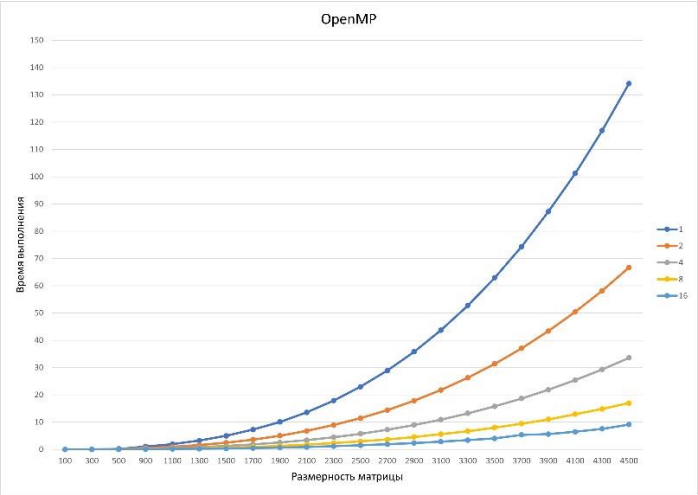
3. OpenMP и MPI:

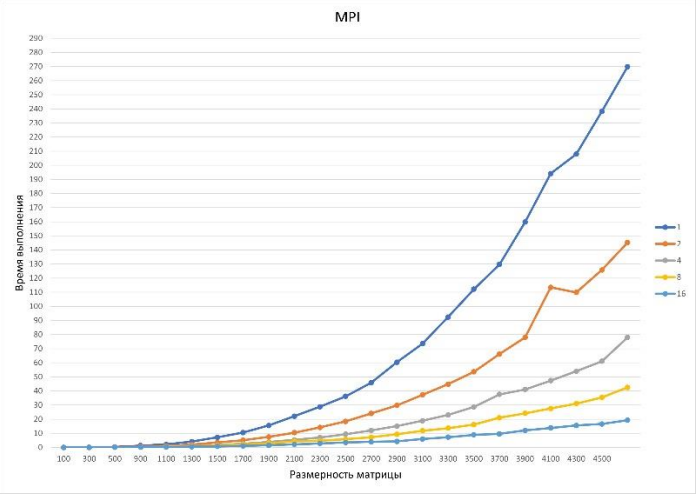
Таблицы с данными:

omp (xlc)					
	1	2	4	8	16
100	0,00893740	0,00849736	0,00809393	0,00818252	0,00853866
300	0,044012	0,0203409	0,010666	0,0060844	0,0043338
500	0,183615	0,092656	0,0472420	0,025000	0,0150213
700	0,50266	0,252413	0,175284	0,066234	0,036784
900	1,06676	0,53493	0,27059	0,138876	0,074397
1100	1,94717	0,97581	0,49154	0,25069	0,13314
1300	3,21374	1,60788	0,80994	0,41147	0,215371
1500	4,954	2,46973	1,24076	0,62994	0,32709
1700	7,2582	3,59434	1,80523	0,91868	0,47734
1900	10,0763	5,0203	2,52044	1,27335	0,65602
2100	13,5988	6,7760	3,40183	1,72268	0,89200
2300	17,863	8,932	4,46440	2,33987	1,16171
2500	22,9498	11,430	5,7326	2,90623	1,48413
2700	28,8864	14,3983	7,2262	3,64794	1,87683
2900	35,807	17,8437	8,9506	4,5279	2,32522
3100	43,741	21,7873	10,9428	5,5798	2,8243
3300	52,771	26,2832	13,2116	6,6724	3,39189
3500	62,949	31,366	15,7718	8,0102	4,04604
3700	74,351	37,0520	18,6522	9,4353	5,3187
3900	87,220	43,393	21,8526	11,0064	5,5841
4100	101,210	50,412	25,4125	12,8919	6,4864
4300	116,916	58,177	29,3290	14,8439	7,5405
4500	134,17	66,697	33,578	16,9880	9,1035

mpi (mpixlc)					
	1	2	4	8	16
100	0,00415576	0,00178297	0,00133558	0,00139559	0,00201979
300	0,117947	0,0503844	0,022081	0,0184587	0,0153005
500	0,397879	0,266639	0,0978995	0,079676	0,0625358
700	1,31551	0,522701	0,264035	0,211094	0,165732
900	2,18001	1,10663	0,55691	0,457161	0,266911
1100	4,07545	2,04373	1,03577	0,80684	0,42542
1300	7,14063	3,50360	1,72352	1,38021	0,686491
1500	10,501	5,10973	2,62341	2,07849	1,06108
1700	15,5153	7,49961	3,81722	2,96093	1,54389
1900	22,1313	10,4767	5,36879	4,35488	2,10114
2100	28,8028	14,1939	7,02731	4,71445	2,83458
2300	36,253	18,425	9,40685	5,79289	3,55094
2500	45,8873	24,081	12,0957	7,16041	3,92428
2700	60,3849	29,8475	15,0895	9,34946	4,30306
2900	73,581	37,2869	18,9082	11,8288	5,91905
3100	92,436	44,7983	23,0157	13,6552	7,2493
3300	112,147	53,6237	28,7217	16,1771	8,92284
3500	129,733	66,162	37,6256	21,0768	9,58313
3700	159,779	78,0327	41,0972	24,1604	12,0728
3900	194,125	113,436	47,3822	27,6061	13,7996
4100	207,918	109,913	53,9894	31,0384	15,5433
4300	238,199	125,812	61,0646	35,4716	16,6282
4500	269,97	145,252	78,055	42,5427	19,3087

Графики:





Вывод

Заметно, что программа на ***OpenMP*** работает быстрее аналога на ***MPI*** в большинстве случаев, однако для малых размерностей алгоритм, распараллеленный средствами MPI, показывает результат немного лучше по времени выполнения.

Причинами подобного поведения могут быть:

1. Использование более тяжеловесных процессов вместо OMP-нитей.
2. Большое число широковещательных-обменов в программе на MPI.
3. Распараллеливание только прямого хода алгоритма Гаусса, основная часть которого выполняется за $O(n^3)$, не затрагивая обратный ход и заполнение матрицы, сложность которых $O(n^2)$.