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 **OpenMP**.

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

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

Определить основные причины недостаточной масштабируемости программы при максимальном числе используемых потоков (нитей).

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 <omp.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 main(int argc,char **argv) {
   double time0, time1;
   int i, j, k;
   /* create arrays */
   for (N=100; N < 4501; N += 200) {
        A=(double *)malloc(N*(N+1)*sizeof(double));
        X=(double *)malloc(N*sizeof(double));
        printf("\n----------------------------------\nGAUSS %dx%d\n",N,N);
        time0 = omp_get_wtime();
        #pragma omp parallel shared(A,N,X) private(i,j,k)
        {
                #pragma omp for schedule(static) collapse(2)
                /* 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;
                        }
                }
                /* elimination */
                for (i=0; i<N-1; i++) {
                #pragma omp for schedule(static)
                        for (k=i+1; k <= N-1; k++) {
                                for (j=i+1; j <= N; j++)
                                        A(k,j) -= A(k,i)*A(i,j)/A(i,i);
                        }
                }
                /* reverse substitution */
                X[N-1] = A(N-1,N)/A(N-1,N-1);
                for (j=N-2; j>=0; j--) {
                #pragma omp for schedule(static)
                        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 = omp_get_wtime();
                printf("Time in seconds=%gs\n",time1-time0);
                prt1a("X=(", X,N>9?9:N,"...)\n");
                free(A);
                free(X);
        }
        printf("\n");
        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: gcc и xlc, поэтому в отчете приведен анализ двух наборов данных.

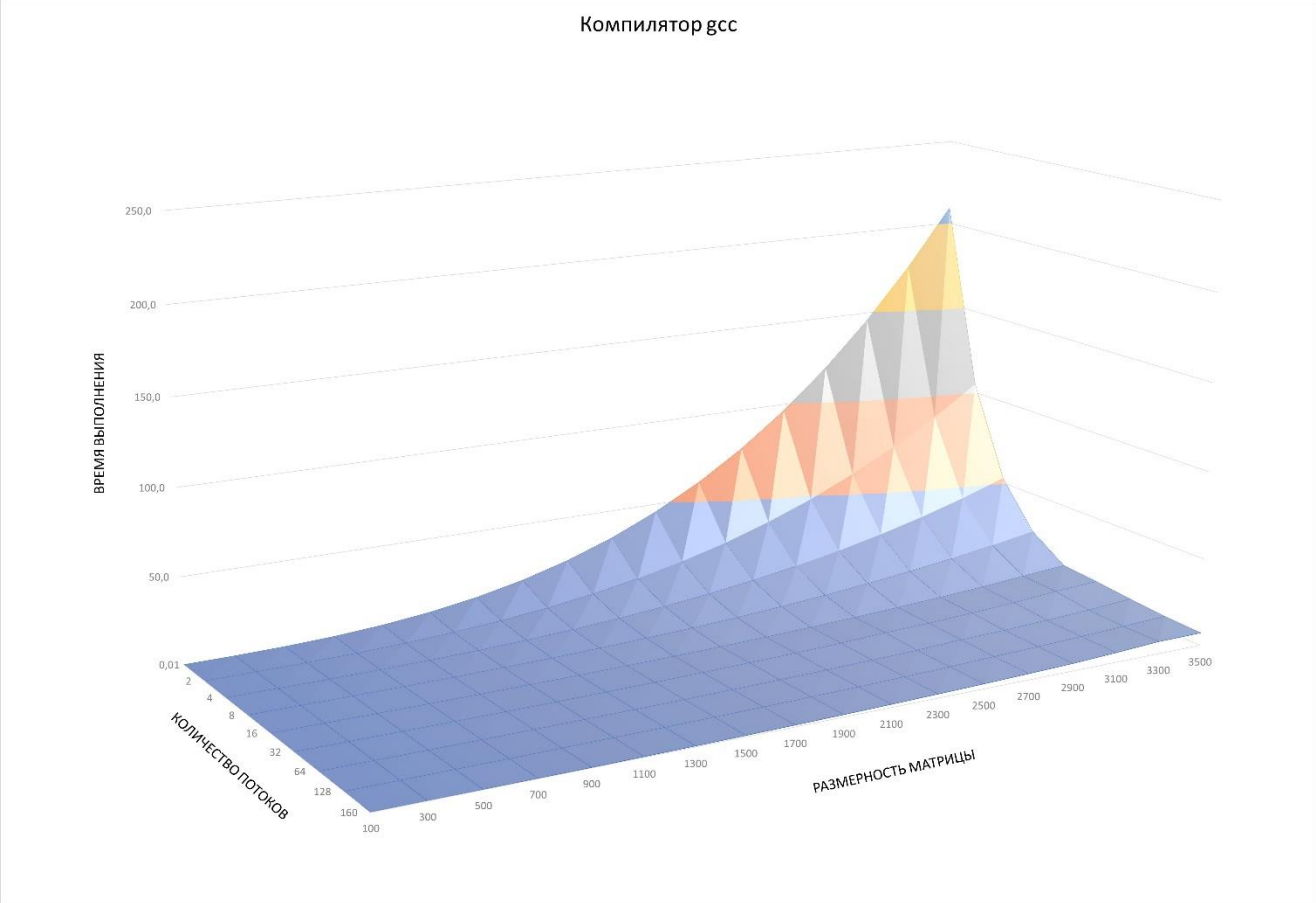
1. gcc:

Таблица с данными:

	1	2	4	8	16	32	64	128	160
100	0,00518253	0,00294275	0,00257661	0,00148253	0,00236604	0,00424843	0,00686342	0,013939	0,038053
300	0,133587	0,0674127	0,0347485	0,0199675	0,0148566	0,0171023	0,0221056	0,0352423	0,04198
500	0,614123	0,30804	0,156865	0,083078	0,0493011	0,0510743	0,0517869	0,0721121	0,0819862
700	1,68102	0,842187	0,427423	0,221773	0,123339	0,119413	0,1086	0,130948	0,14243
900	3,56614	1,80428	0,903884	0,467462	0,25156	0,234136	0,201572	0,220511	0,227855
1100	6,5007	3,29392	1,64678	0,920704	0,448949	0,410453	0,341698	0,352215	0,348923
1300	10,7406	5,39096	2,71479	1,39105	0,738891	0,661768	0,541587	0,533384	0,508695
1500	16,4949	8,27738	4,16528	2,132	1,11163	1,00170	0,807099	0,772381	0,724369
1700	24,0158	12,0763	6,05768	3,12893	1,61324	1,43918	1,15216	1,07678	0,99121
1900	33,5101	16,8082	8,45303	4,31736	2,23725	1,99727	1,58515	1,45737	1,32606
2100	45,2306	22,749	11,4098	5,82247	3,01560	2,68168	2,11653	1,92037	1,73524
2300	59,39800	29,8273	14,9869	7,64455	3,94582	3,50838	2,75396	2,47703	2,22801
2500	76,2648	38,3633	19,2409	9,81262	5,05491	4,77252	3,51607	3,1325	2,78845
2700	96,0578	48,2486	24,2349	12,3504	6,35983	5,65172	4,7628	3,97298	3,46305
2900	119,008	59,8548	30,0285	15,2972	7,87335	6,95411	5,43701	4,80939	4,23746
3100	145,388	72,9694	36,6722	18,712	9,60946	8,45248	6,62606	5,79830	5,49665
3300	175,345	87,995	44,2324	22,532	11,6793	10,6328	7,98087	6,94331	7,00352
3500	209,202	105,057	52,7715	26,9187	13,8038	12,1568	9,99664	8,20848	7,21274

(количество потоков - по горизонтали, размерность матрицы – по вертикали)

График:



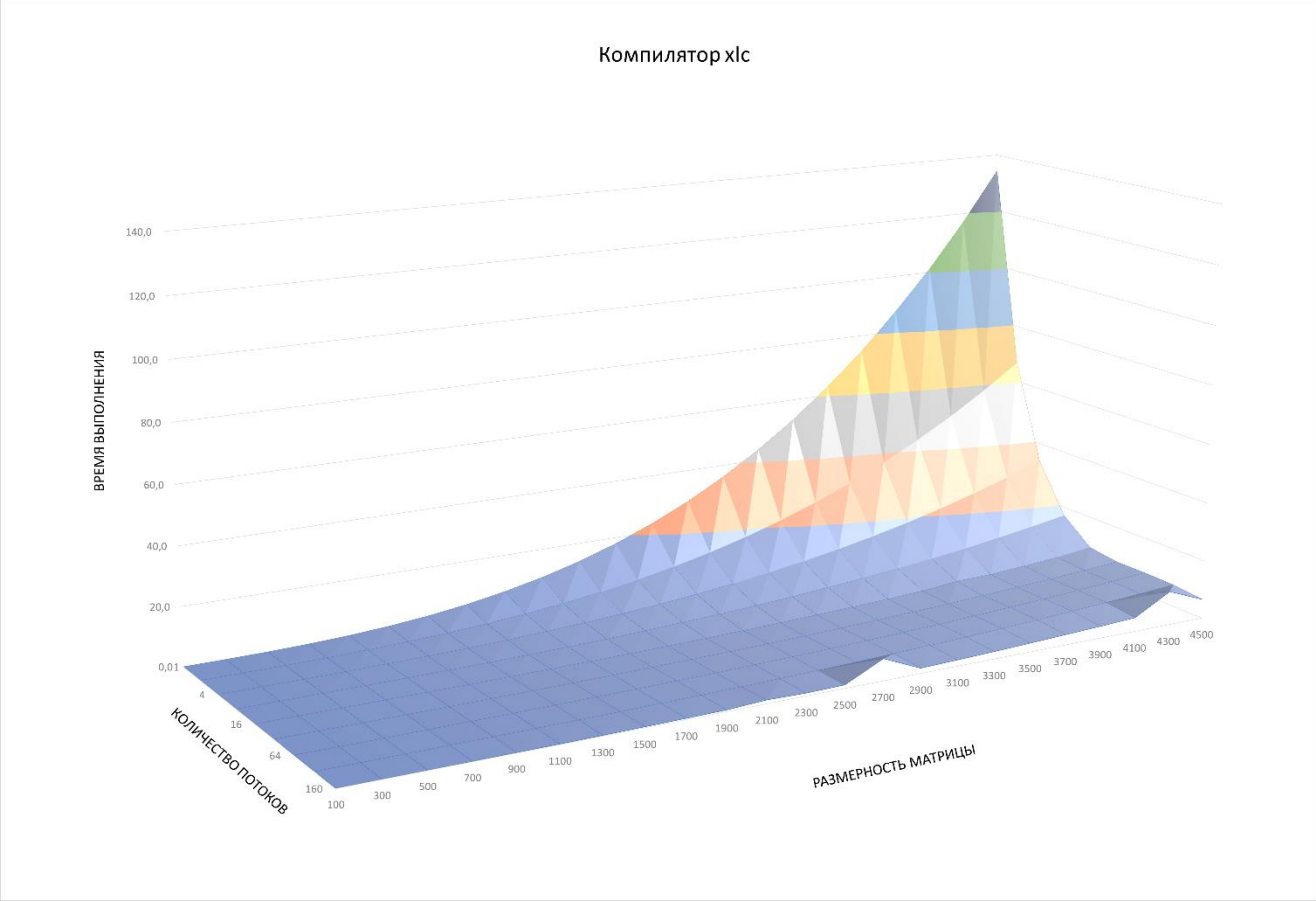
2. xlc:

Таблица с данными:

	1	2	4	8	16	32	64	128	160
100	0,0089374	0,00849736	0,00809393	0,00818252	0,00853866	0,0117433	0,0347027	0,0182231	0,0225381
300	0,0440121	0,0203409	0,0106664	0,00608443	0,00433378	0,00505518	0,00600567	0,00930472	0,0115684
500	0,183615	0,0926559	0,047242	0,0249995	0,0150213	0,0146982	0,0167326	0,0218503	0,0234164
700	0,502663	0,252413	0,175284	0,066234	0,0367836	0,0347739	0,0378633	0,0446827	0,0462361
900	1,06676	0,534932	0,270588	0,138876	0,0743969	0,0690599	0,0733718	0,0808954	0,076937
1100	1,94717	0,975811	0,491537	0,250686	0,133139	0,121649	0,127438	0,13509	0,123746
1300	3,21374	1,60788	0,809943	0,411469	0,215371	0,196296	0,202886	0,211824	0,187412
1500	4,95375	2,46973	1,24076	0,62994	0,327086	0,297333	0,307142	0,312128	0,278155
1700	7,25816	3,59434	1,80523	0,918679	0,477343	0,429117	0,445656	0,44417	0,383134
1900	10,0763	5,02025	2,52044	1,27335	0,656017	0,59423	0,614511	0,608127	0,518896
2100	13,5988	6,77599	3,40183	1,72268	0,892004	0,79816	0,811947	0,809844	1,20046
2300	17,8631	8,93246	4,4644	2,33987	1,16171	1,04446	1,06119	1,05188	0,909703
2500	22,9498	11,4299	5,73258	2,90623	1,48413	1,33673	1,35618	1,34091	1,11467
2700	28,8864	14,3983	7,22624	3,64794	1,87683	1,67991	1,70249	1,67643	7,28992
2900	35,8069	17,8437	8,95056	4,52793	2,32522	2,07700	2,10314	2,06643	1,73338
3100	43,7413	21,7873	10,9428	5,57978	2,82429	2,53280	2,56320	2,52357	2,06896
3300	52,7713	26,2832	13,2116	6,67239	3,39189	3,05145	3,08965	3,01754	2,47539
3500	62,9494	31,3663	15,7718	8,01024	4,04604	3,63784	3,67329	3,59103	2,93842
3700	74,351	37,052	18,6522	9,43533	5,31872	4,30006	4,33406	4,2293	3,49069
3900	87,2202	43,3929	21,8526	11,0064	5,58405	5,02979	5,06920	4,9405	4,0279
4100	101,21	50,4118	25,4125	12,8919	6,48638	5,83629	5,88763	5,75264	4,69092
4300	116,916	58,1772	29,329	14,8439	7,54046	6,73133	6,78324	6,59261	11,4572
4500	134,173	66,6968	33,5776	16,988	9,10348	7,72308	7,76739	7,5493	6,5056

(количество потоков - по горизонтали, размерность матрицы – по вертикали)

График:



Разница размерности в 1000 по сравнению с gcc, возникает в силу скорости работы и возможности уложиться в тайм-аут на машине Polus.

Вывод

1. При увеличении числа потоков почти линейно уменьшается время работы алгоритма.

2. Большое число потоков помогает в случае применения алгоритма к матрицам большой размерности. Лучший результат достигается при 16 и 64 потоках. В случае применения алгоритма к матрицам малой размерности увеличение количества потоков ухудшает время работы алгоритма из-за накладных расходов на порождение этих потоков. Из этого можно сделать вывод, что оптимальное количество потоков пропорционально растет вместе с объемом данных, однако только до некоторого порогового значения (до 100 лучше всего 1–2 потока, от 500 уже 32 потока и больше).

3. Использование компилятора xlc вместо gcc позволяют существенно ускорить работу программы в несколько раз.