Суперкомпьютеры и параллельная обработка данных – 2021

Практическое задание

Отчет о проделанной работе (часть 1)

Выполнил студент 328 группы ВМК МГУ
Столяров Роман Константинович

2021 год

1) Задача

Реализовать параллельную версию а**лгоритма Гаусса решения СЛАУ** (приведения матрицы к верхнему треугольному виду) при помощи технологии **OpenMP.**
Исследовать масштабируемость полученной параллельной программы: построить графики зависимости времени исполнения от числа ядер/процессоров для различного объёма входных данных.
Для каждого набора входных данных найти количество ядер/процессоров, при котором время выполнения задачи перестаёт уменьшаться.
Определить основные причины недостаточной масштабируемости программы при максимальном числе используемых ядер/процессоров.

2) Описание метода Гаусса

Алгоритм решения СЛАУ методом Гаусса состоит из 2-х этапов – 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 невырожденности матрицы, после чего i-я строка делится на данный элемент. Затем из остальных $i+1 \dots n$ строк вычитается i-я строка, умноженная на $a_{i+1,i} \dots a_{n,i}$ 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: $Q = Q_2 + Q_3 = \frac{n^3}{3} + O(n^2)$.

3) Текст 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);
void wtime(double *t) {
   static int sec = -1;
   struct timeval tv;
   gettimeofday(&tv, (void *)0);
   if (sec < 0) sec = tv.tv_sec;
   *t = (tv.tv_sec - sec) + 1.0e-6*tv.tv_usec;
}

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 < 2000; N += 200) {
                A=(double *)malloc(N*(N+1)*sizeof(double));
                X=(double *)malloc(N*sizeof(double));
                printf("GAUSS %dx%d\n----------\n",N,N);
                wtime(&time0);
#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++) {
                                        /* this matrix i use for debug */
                                        if (j==N) {
                                                A(i,j) = N;
                                        } else if (i >= j) {
                                                A(i,j) = i + 1 + j;
                                        } else {
                                                A(i,j) = 0;
                                        }
                                }
                        }
                        /* 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);
                        }
}
                wtime(&time1);
                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);
}
```

4) Аналитика

Для запуска программы был сделан скрипт shell запускающий ее для разных размеров матриц (с сеткой 100x100, 300x300, …, 1900x1900) на разном числе потоков на машине Polus.

Результаты измерений:

1, 2, 4, 8, 16 потоков:

100	300	500	700	900	1100	1300	1500	1700	1900
0.008995	0.039514	0.18168	0.49701	1.05521	1.92656	3.18024	4.88842	7.11205	9.97632
0.008953	0.039531	0.181634	0.497006	1.05484	1.93064	3.18511	4.88889	7.11159	9.92782
0.009004	0.039507	0.181649	0.497007	1.05508	1.92599	3.17994	4.88961	7.113	9.97651
0.008998	0.0395	0.181638	0.496999	1.05479	1.9264	3.18133	4.88894	7.11204	9.92829
0.008935	0.039501	0.181622	0.496932	1.05525	1.92553	3.18112	4.88795	7.11196	9.92794
100	300	500	700	900	1100	1300	1500	1700	1900
0.008405	0.020168	0.091629	0.249745	0.577252	0.964794	1.59078	2.44336	3.55802	4.96414
0.008409	0.020138	0.091615	0.24975	0.529258	0.965567	1.59078	2.4435	3.55533	4.96465
0.008362	0.02012	0.091668	0.249804	0.529335	0.964753	1.5909	2.44304	3.56933	4.96419
0.008379	0.020126	0.091618	0.24972	0.529254	0.964917	1.59067	2.44366	3.55574	4.96323
0.008411	0.020121	0.091647	0.249758	0.529433	0.964595	1.63856	2.44321	3.55693	4.9651
100	300	500	700	900	1100	1300	1500	1700	1900
0.008178	0.010531	0.046799	0.126909	0.267966	0.492029	0.824257	1.23471	1.78975	2.5089
0.008165	0.01056	0.046847	0.127222	0.272828	0.488658	0.806755	1.2383	1.79312	2.50409
0.008091	0.01058	0.046784	0.126748	0.267631	0.486549	0.804634	1.2355	1.7951	2.51163
0.008132	0.010548	0.046795	0.126909	0.271929	0.491241	0.863553	1.23969	1.83087	2.4997
0.008128	0.010546	0.046902	0.126746	0.267744	0.490988	0.801296	1.22873	1.78673	2.50817
100	300	500	700	900	1100	1300	1500	1700	1900
0.008225	0.005926	0.024838	0.06576	0.13778	0.248918	0.412849	0.625594	0.916698	1.26584
0.008129	0.005926	0.024825	0.065761	0.137748	0.249166	0.408885	0.625756	0.912357	1.2735
0.008147	0.005913	0.024822	0.066622	0.137556	0.248664	0.408287	0.629094	0.96322	1.26347
0.008169	0.005933	0.02483	0.06997	0.141637	0.248888	0.412858	0.625532	0.915154	1.26809
0.008158	0.005934	0.024807	0.065607	0.137517	0.250733	0.414508	0.62696	0.908759	1.28709
100	300	500	700	900	1100	1300	1500	1700	1900
0.008559	0.005808	0.023791	0.06162	0.128036	0.231066	0.378272	0.578605	0.838741	1.16841
0.008512	0.005845	0.023673	0.061671	0.128115	0.231503	0.378648	0.579041	0.839095	1.16951
0.008454	0.005937	0.023602	0.061621	0.128037	0.230963	0.378658	0.578387	0.838797	1.16816
0.008764	0.005865	0.023562	0.061684	0.128045	0.231081	0.378395	0.578572	0.838698	1.16819
0.008507	0.005797	0.023598	0.061625	0.128133	0.231281	0.37846	0.578576	0.838724	1.16801

32, 64, 128 потоков:

100	300	500	700	900	1100	1300	1500	1700	1900
0.00949	0.006507	0.024865	0.063836	0.131469	0.236197	0.386338	0.58916	0.852952	1.18641
0.009273	0.006426	0.025006	0.063761	0.131488	0.236074	0.385832	0.588577	0.852711	1.18625
0.009254	0.006477	0.024868	0.063738	0.131488	0.236087	0.385868	0.59024	0.854899	1.18655
0.023364	0.006693	0.024897	0.063995	0.131555	0.236167	0.385834	0.588673	0.852439	1.18584
0.009439	0.006433	0.024833	0.063774	0.1314	0.236027	0.385795	0.588533	0.852383	1.18607
100	300	500	700	900	1100	1300	1500	1700	1900
0.012268	0.008781	0.026209	0.064629	0.131004	0.232606	0.377823	0.574386	0.829608	1.15652
0.011543	0.007637	0.026121	0.064685	0.130896	0.232664	0.377805	0.574155	0.829563	1.151
0.011421	0.007699	0.026237	0.064689	0.130978	0.232796	0.37789	0.574427	0.829462	1.15095
0.011317	0.007665	0.026078	0.06463	0.13086	0.232765	0.377854	0.574668	0.82974	1.15148
0.011381	0.007698	0.026119	0.064679	0.130835	0.232566	0.377922	0.57423	0.829496	1.15098
100	300	500	700	900	1100	1300	1500	1700	1900
0.065147	0.023163	0.053649	0.105116	0.184538	0.302044	0.463045	1.4759	0.970224	1.29394
0.02563	0.019142	0.044848	0.098395	0.184798	0.3033	0.463603	0.678338	0.953061	1.29584
0.023524	0.018951	0.044556	0.097565	0.184259	0.303297	0.464768	0.678152	0.952708	1.30077
0.021737	0.019228	0.044852	0.095131	0.184641	0.302508	0.464139	0.67836	0.952377	1.29487
0.02182	0.019109	0.044617	0.094854	0.184282	0.30249	0.464928	0.677796	0.952662	1.29491

Графики:

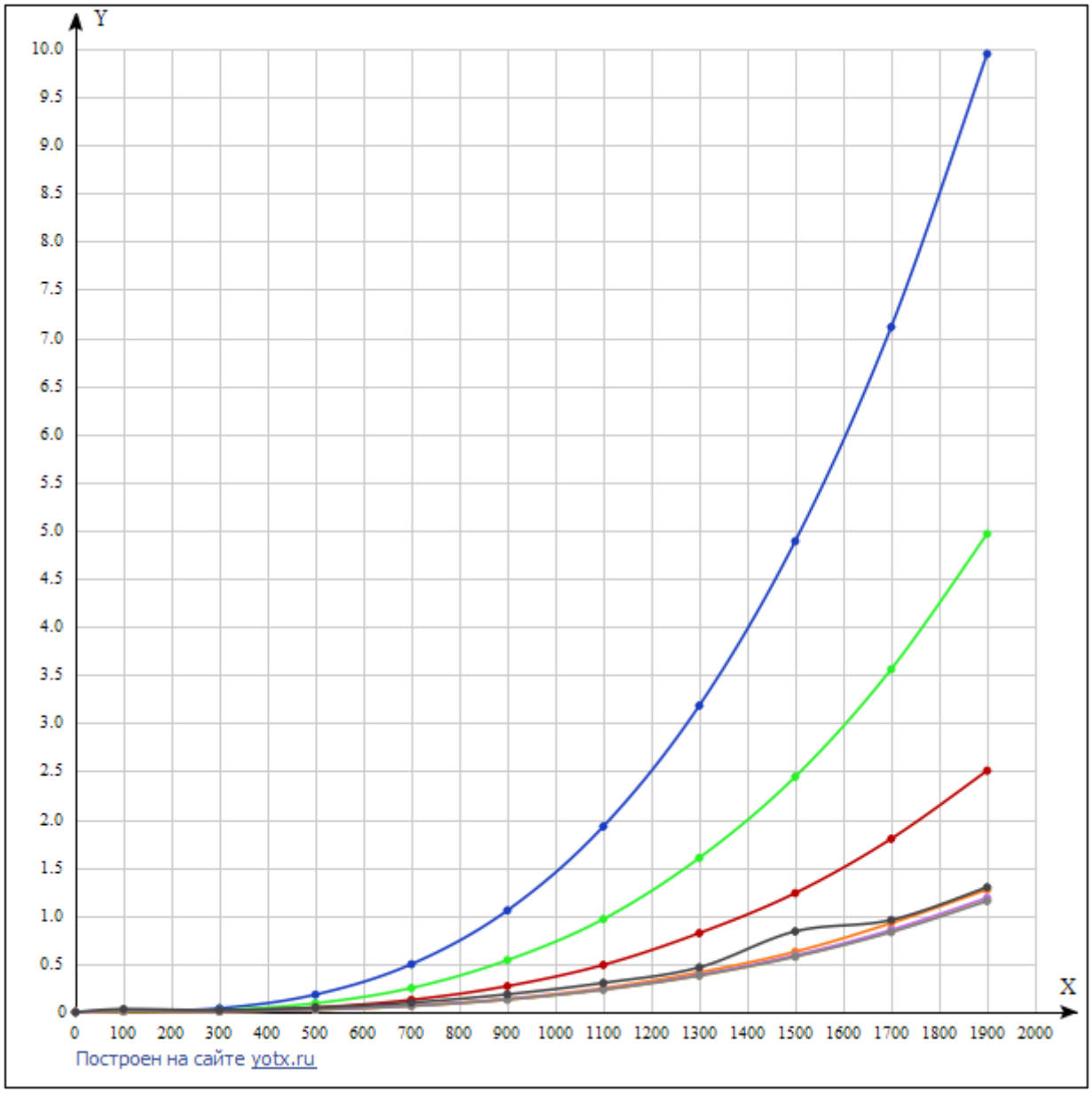
По оси Y — время выполнения в секундах

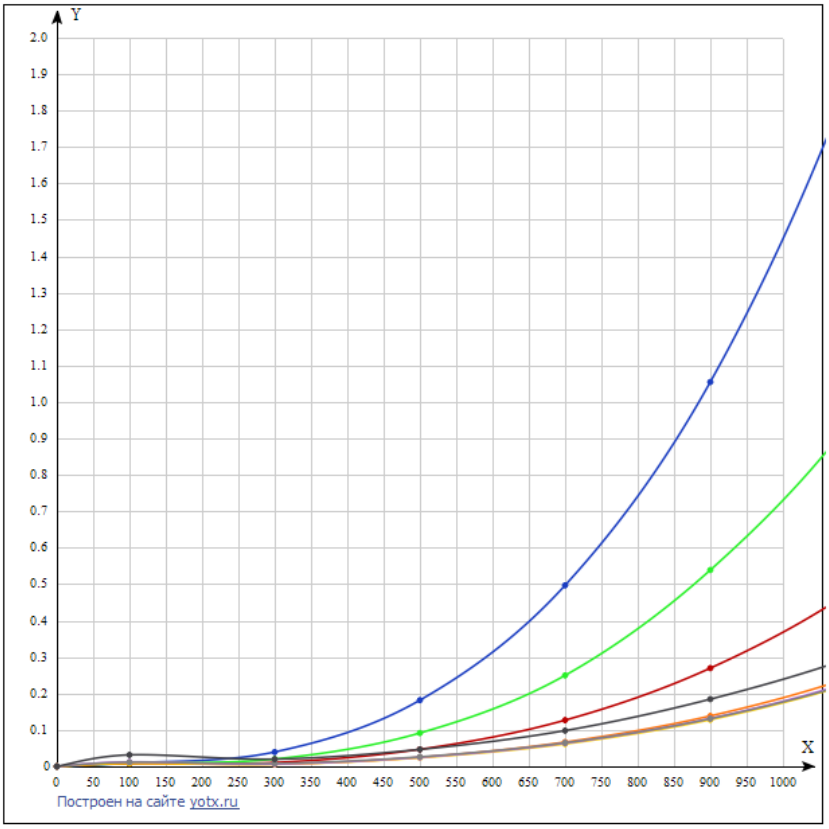
По оси X — размер стороны матрицы

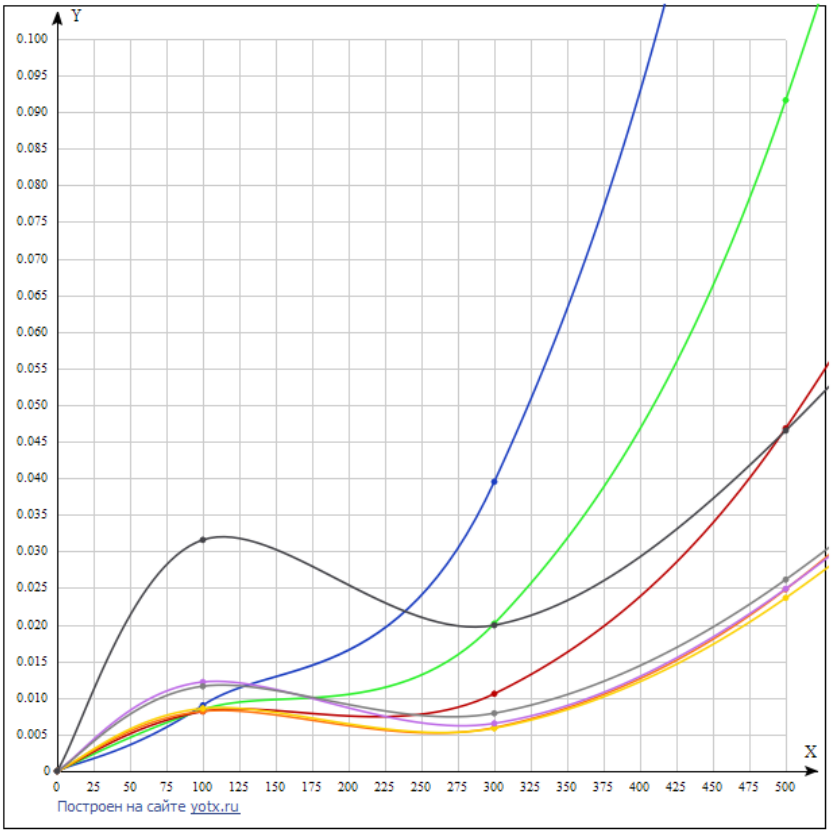
Синий — 1 поток

Зеленый — 2 потока

Красный — 4 потока
Оранжевый — 8 потоков
Желтый — 16 потоков
Фиолетовый — 32 потока
Серый — 64 потока
Черный — 128 потоков




Y
X
Построен на сайте yotx.ru


Y
X
Построен на сайте yotx.ru

Результат:
1) Заметно что **при увеличении числа потоков почти линейно уменьшается время работы алгоритма** (до определенного порога, что вероятно вызвано ограничениями по числу ядер, доступных на суперкомпьютере polus).
2) Заметно что большее число потоков помогает лишь в случае применения алгоритма к большим матрицам (лучший результат достигается при 16 и 64 потоках). В случае же применения алгоритма к маленьким матрицам большое количество 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 поток, после 300 уже 64).
3) Доп. Вывод: статически определенный размер матрицы и использование компилятора xlc вместо gcc позволяют **существенно ускорить работу программы**.