Суперкомпьютеры и параллельная обработка данных – 2021

Практическое задание

Отчет о проделанной работе (часть 2)

Выполнил студент 328 группы ВМК МГУ
Столяров Роман Константинович

2021 год

1) Задача

Реализовать параллельную версию **алгоритма Гаусса решения СЛАУ** (приведения матрицы к верхнему треугольному виду) при помощи технологии **MPI.**
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.
Сравнить эффективность **OpenMP** и **MPI-версий** параллельной программы.

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 <mpi.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 *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 < 2000; 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("GAUSS %dx%d\n----------------------------------\n",N,N);
                        /* 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;
                                        }
                        wtime(&time0);
                }
                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);
                        }
                        wtime(&time1);
```

```
                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);
}
```

4) Аналитика

Для запуска программы был изменен код (добавлен цикл по размеру матрицы) и сделан скрипт запускающий ее для разных размеров матриц (с сеткой размеров матриц: 100x100, 300x300, …, 1900x1900) на разном числе потоков/процессов на машине Polus.

Результаты измерений:

1, 2, 4, 8, 16 потоков:

Слева — MPI, справа — OpenMP:

100	300	500	700	900	1100	1300	1500	1700	1900
0.003145	0.083245	0.37736	1.05913	2.17471	4.12186	6.70754	10.2726	14.9158	20.6274
0.00304	0.080456	0.372636	1.05747	2.18697	3.99454	6.5754	10.1265	14.6079	20.5443
0.003074	0.080266	0.372441	1.02069	2.163	3.9812	6.52993	10.106	14.6961	20.8225
0.003039	0.079847	0.369352	1.0119	2.15895	3.95116	6.47067	10.0004	14.9389	20.6318
0.003058	0.080189	0.369363	1.01274	2.14637	3.94281	6.49299	10.031	14.5575	20.3005
100	300	500	700	900	1100	1300	1500	1700	1900
0.001752	0.041533	0.251342	0.576652	1.09031	2.04497	3.34029	5.16816	7.39002	10.3797
0.001757	0.04192	0.189157	0.521978	1.11921	2.08283	3.32441	5.10913	7.32096	10.4378
0.001753	0.041687	0.189128	0.540406	1.13546	2.08826	3.43153	5.01124	7.24218	10.1044
0.001746	0.041248	0.186066	0.506488	1.07733	1.95769	3.23353	5.14564	7.69577	10.8139
0.001746	0.041826	0.20792	0.521948	1.12047	2.06824	3.43021	5.23339	7.8196	10.7563
100	300	500	700	900	1100	1300	1500	1700	1900
0.00174	0.032253	0.145032	0.295464	0.617216	1.18116	1.84108	2.94634	4.19801	5.5377
0.001752	0.032143	0.098398	0.302971	0.606525	1.13293	1.77616	2.94319	4.10318	5.56688
0.00176	0.022506	0.140308	0.301496	0.667321	1.18572	1.80559	2.75989	3.99554	5.79086
0.001726	0.032403	0.145429	0.302548	0.640459	1.19108	1.9188	2.91846	4.01282	5.83927
0.00176	0.033364	0.136008	0.275288	0.569373	1.08429	1.80541	2.83934	5.53964	8.16894
100	300	500	700	900	1100	1300	1500	1700	1900
0.001874	0.025641	0.113286	0.217603	0.444128	0.802008	1.31103	2.02353	2.92314	4.15281
0.001932	0.025846	0.098178	0.238344	0.462898	0.800012	1.32472	1.99698	2.89946	4.05423
0.001879	0.025807	0.102202	0.237856	0.447841	0.808219	1.32068	2.01743	2.93027	3.1738
0.001945	0.026189	0.115244	0.207491	0.360552	0.67493	1.09425	1.71166	2.3504	3.18205
0.001898	0.025555	0.113204	0.19679	0.370647	0.605896	1.04905	1.66303	2.39985	3.31719
100	300	500	700	900	1100	1300	1500	1700	1900
0.001833	0.014971	0.061836	0.164473	0.230163	0.352917	0.657336	0.912527	1.3827	1.97719
0.002539	0.022203	0.096094	0.134852	0.241834	0.37137	0.637869	0.961884	1.48113	1.95417
0.002011	0.022496	0.096139	0.167088	0.237519	0.42536	0.685313	0.974815	1.37412	1.90456
0.002435	0.02215	0.094899	0.16021	0.224085	0.357554	0.624197	0.950783	1.32484	1.84539
0.00162	0.015019	0.060006	0.162563	0.260252	0.444453	0.726169	1.07071	1.52794	1.88956

100	300	500	700	900	1100	1300	1500	1700	1900
0.02943	0.130735	0.492963	1.35161	2.82707	5.19384	8.54769	13.1331	19.0989	25.9922
0.010576	0.069182	0.32688	0.874723	1.85918	3.48525	5.79669	8.72034	12.5755	18.0125
0.010792	0.072062	0.331389	0.918943	1.93774	3.52152	5.85508	8.94691	13.0275	18.2568
0.010845	0.071629	0.331686	0.93704	1.96804	3.50084	5.8735	8.96066	13.0386	18.2292
0.010811	0.071603	0.328594	0.895178	1.91117	3.46838	5.79672	8.95168	13.0114	14.2893
100	300	500	700	900	1100	1300	1500	1700	1900
0.047869	0.036387	0.167474	0.460188	0.722359	1.0276	1.68329	2.58821	3.82431	5.34352
0.039399	0.035512	0.163087	0.445808	0.94305	1.71767	2.81964	4.36554	6.3827	8.90107
0.009584	0.035306	0.161814	0.445826	0.946594	1.71889	2.83422	4.34195	6.31795	8.82224
0.009627	0.035371	0.161808	0.438808	0.930831	1.70553	2.80561	4.28023	6.26888	8.76596
0.00957	0.035077	0.160495	0.439084	0.931797	1.68824	2.80544	2.77854	3.83229	5.35839
100	300	500	700	900	1100	1300	1500	1700	1900
0.042369	0.018208	0.082604	0.22389	0.476986	0.861293	1.42308	2.18945	2.96971	2.76607
0.008332	0.01063	0.047274	0.127998	0.270234	0.500112	0.9511	1.40758	1.94317	2.54903
0.008554	0.012869	0.047452	0.137306	0.374231	0.516186	0.817	1.34553	2.03792	2.75108
0.008304	0.010632	0.047229	0.127903	0.270126	0.491073	0.808801	1.2819	2.00078	2.86313
0.008265	0.010624	0.04717	0.127917	0.270992	0.500462	0.808967	1.24898	2.06973	2.65589
100	300	500	700	900	1100	1300	1500	1700	1900
0.034269	0.009589	0.0412	0.111189	0.234372	0.513504	0.878021	1.36039	1.98436	2.77848
0.009206	0.011917	0.052891	0.141558	0.298928	0.541987	0.892531	1.36632	1.99541	2.73475
0.008729	0.010565	0.045899	0.123482	0.243044	0.430495	0.710688	1.08584	1.5819	2.20233
0.058512	0.009793	0.041665	0.111509	0.234146	0.430305	0.705815	1.08605	1.57906	2.2044
0.041056	0.010606	0.041984	0.111947	0.237385	0.430874	0.708775	1.08626	1.57442	2.17694
100	300	500	700	900	1100	1300	1500	1700	1900
0.046131	0.00622	0.023992	0.062599	0.129724	0.234948	0.384304	0.587699	0.872778	1.1888
0.008967	0.010191	0.043665	0.116675	0.245774	0.287769	0.408106	0.590396	0.915227	1.2499
0.009057	0.010465	0.044497	0.118578	0.162198	0.237141	0.386732	0.590868	0.89866	1.23342
0.00893	0.010211	0.043605	0.117056	0.240079	0.236988	0.416757	0.640656	0.905444	1.22069
0.008842	0.006079	0.024353	0.063245	0.130999	0.235872	0.385753	0.588504	0.872621	1.22098

Результаты OpenMP чуть хуже чем в прошлом отчете из за загруженности машины Polus в выходные перед зачетами. А для справедливой оценки необходимо было запускать программы на MPI и OpenMP в одно время и на одной машине.

Графики:

Слева — MPI, справа — OpenMP

По оси Y — время выполнения в секундах

По оси X — размер стороны матрицы

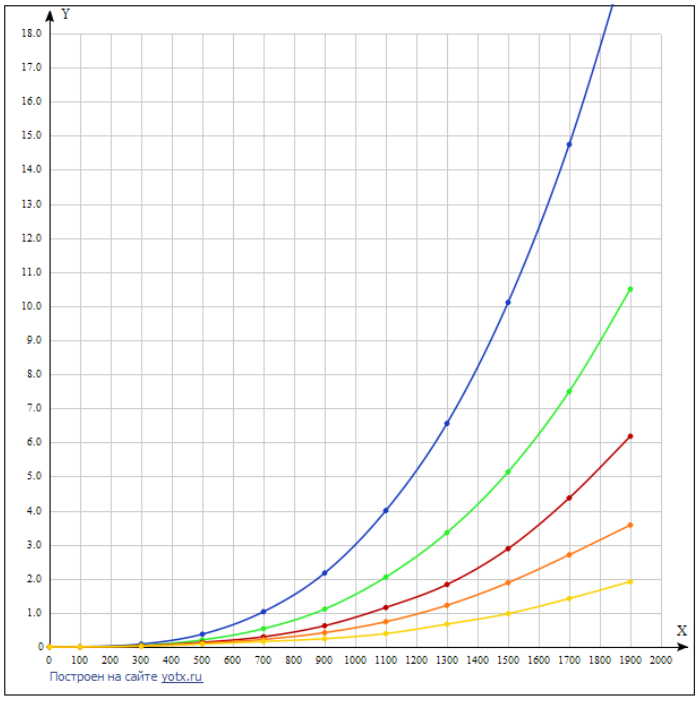
Синий — 1 поток

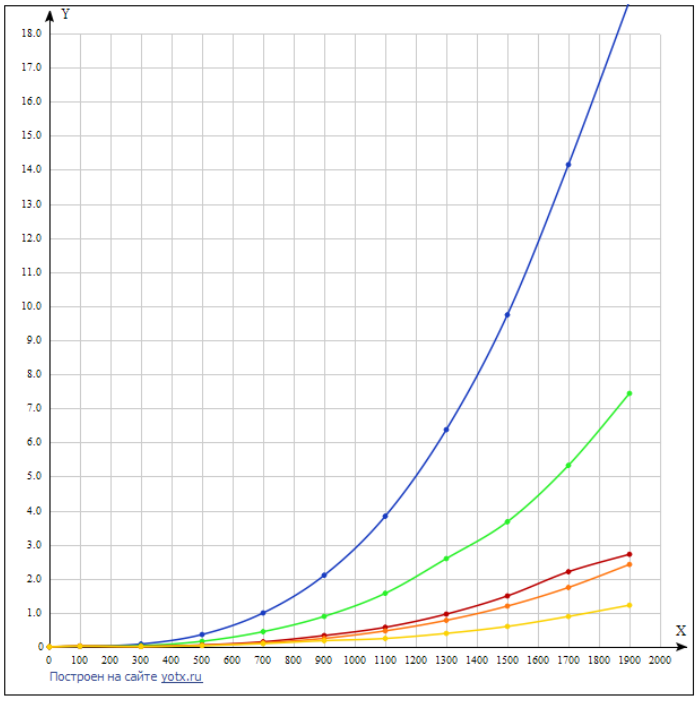
Зеленый — 2 потока

Красный — 4 потока

Оранжевый — 8 потоков

Желтый — 16 потоков





Результат сравнения OpenMP и MPI:

Заметно, что программа на **OpenMP** работает в среднем быстрее аналога на **MPI.**

Причинами подобного поведения могут быть:

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

Для запуска программы на большом количестве процессов был изменен код (добавлен цикл по размеру матрицы) и проделаны несколько запусков данной программы с помощью mpisubmit.bg (с сеткой размеров матриц: 100x100, 300x300, …, 1900x1900) на разном числе процессов на машине Blue Gene/P.

Результаты измерений:

32, 64, 128, 256, 512 потоков:

100	300	500	700	900	1100	1300	1500	1700	1900
0.001688	0.014727	0.047628	0.113987	0.21679	0.367212	0.571711	0.840849	1.18021	1.59598
0.001704	0.014745	0.047579	0.11395	0.216741	0.367202	0.57163	0.840791	1.18044	1.59591
0.001705	0.01475	0.047609	0.113983	0.21682	0.367163	0.571644	0.840825	1.18024	1.59606
0.001678	0.014743	0.047618	0.113999	0.216832	0.367036	0.571625	0.840797	1.18034	1.59593
0.001707	0.014721	0.047597	0.11396	0.21679	0.367105	0.571566	0.840866	1.18022	1.59597
100	300	500	700	900	1100	1300	1500	1700	1900
0.00169	0.012514	0.03708	0.085273	0.156215	0.254808	0.385411	0.553387	0.763089	1.01153
0.001692	0.012479	0.036991	0.085135	0.156068	0.25461	0.385232	0.553258	0.762743	1.01127
0.001703	0.012522	0.037068	0.085285	0.156225	0.254794	0.385472	0.553513	0.762854	1.01158
0.001694	0.012509	0.037067	0.085282	0.156107	0.254817	0.385462	0.553526	0.762845	1.01155
0.001698	0.012504	0.037088	0.085301	0.156196	0.254955	0.385454	0.55358	0.762905	1.01156
100	300	500	700	900	1100	1300	1500	1700	1900
0.001695	0.01183	0.031112	0.071263	0.125035	0.197095	0.294725	0.408623	0.553487	0.719085
0.001692	0.011812	0.031098	0.071403	0.124928	0.19724	0.294791	0.408608	0.553808	0.718834
0.001691	0.011823	0.031093	0.0713	0.124947	0.197125	0.294698	0.408607	0.55381	0.718835
0.001691	0.011793	0.031122	0.071344	0.124944	0.197205	0.294737	0.408619	0.553472	0.718656
0.001688	0.011823	0.031118	0.071302	0.124947	0.197128	0.294746	0.408481	0.553569	0.719007
100	300	500	700	900	1100	1300	1500	1700	1900
0.001768	0.011925	0.03128	0.06443	0.107765	0.167204	0.24884	0.343602	0.450983	0.569305
0.001786	0.011903	0.031308	0.064432	0.107799	0.167142	0.24888	0.343483	0.450753	0.569126
0.001804	0.011997	0.031396	0.06478	0.108324	0.167704	0.249503	0.344102	0.451942	0.570543
0.001784	0.011935	0.031254	0.06443	0.1078	0.167158	0.248836	0.343579	0.451131	0.569218
0.001787	0.01194	0.031293	0.064491	0.107742	0.16717	0.248833	0.343608	0.450706	0.569706
100	300	500	700	900	1100	1300	1500	1700	1900
0.001776	0.01171	0.030986	0.064318	0.107609	0.162298	0.22837	0.305926	0.392496	0.487097
0.001775	0.011721	0.031029	0.064214	0.107527	0.162245	0.228446	0.305597	0.391476	0.484751
0.001784	0.011707	0.031053	0.064156	0.107368	0.16247	0.228469	0.305183	0.391518	0.486279
0.001857	0.011701	0.030992	0.064176	0.107613	0.16182	0.228514	0.305321	0.392362	0.485246
0.00178	0.011725	0.030996	0.064295	0.107701	0.162411	0.228262	0.304611	0.391211	0.485696

Графики:

По оси Y — время выполнения в секундах

По оси X — размер стороны матрицы

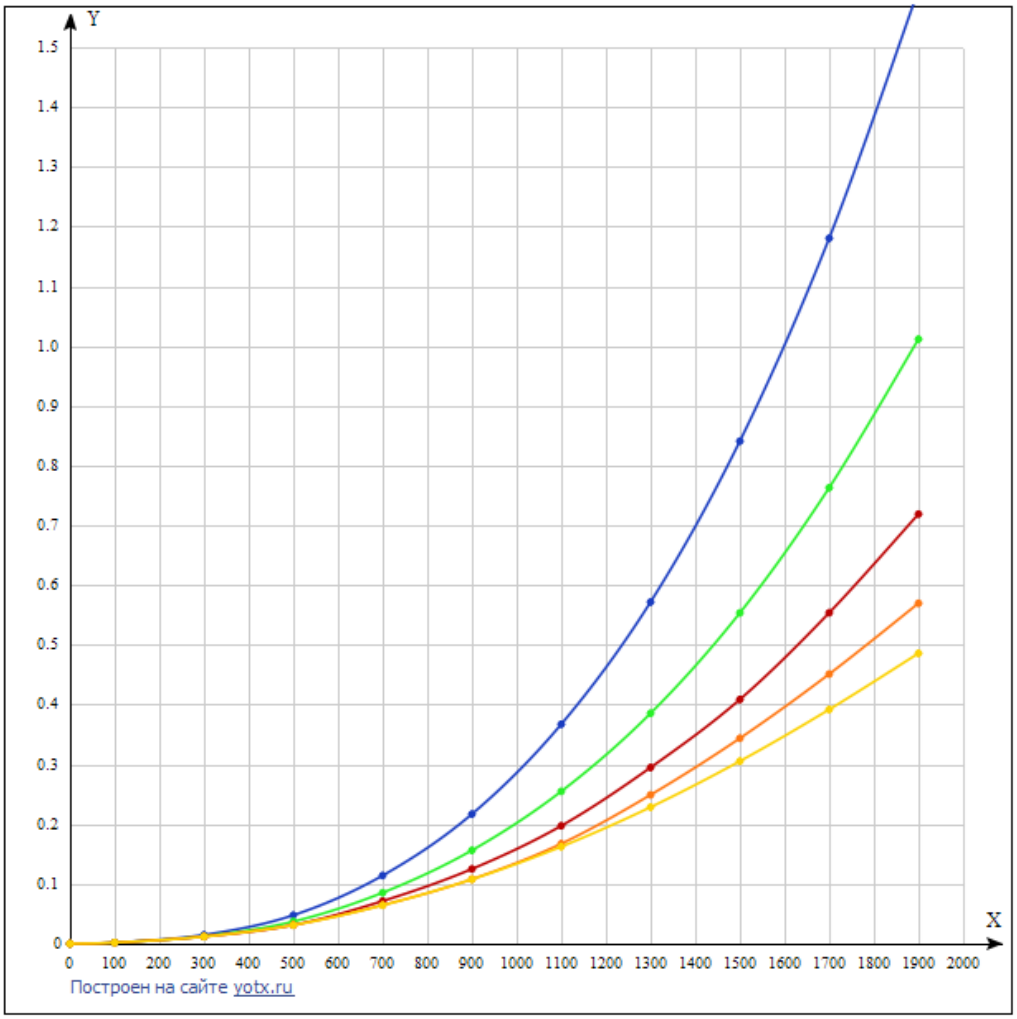
Синий — 32 потока

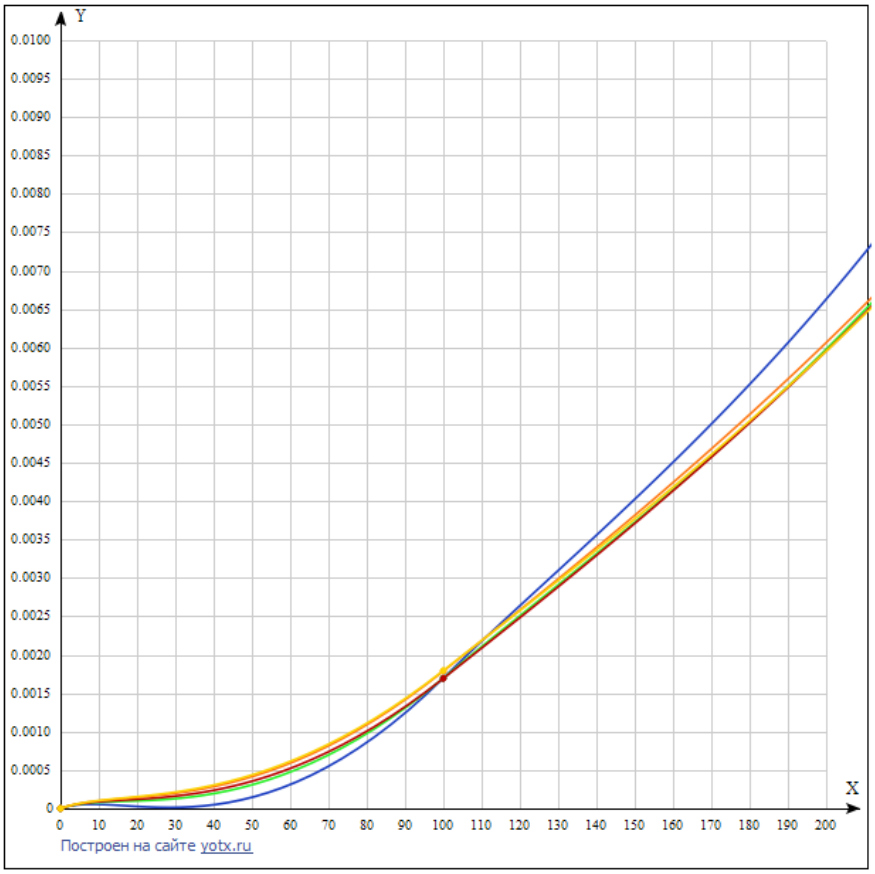
Зеленый — 64 потока

Красный — 128 потоков

Оранжевый — 256 потоков

Желтый — 512 потоков





Результат:

1) Заметно что **при увеличении числа процессов уменьшается время работы алгоритма.** Это изменение **нелинейно** из-за того что параллельно работает только прямой ход алгоритма Гаусса.

2) На малых объемах данных все еще выигрывают версии программы с **меньшим числом процессов**, однако разница не настолько существенна, как при использовании **OpenMP**.