МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ М. В. ЛОМОНОСОВА
ФИЛИАЛ МОСКОВСКОГО ГОСУДАРСТВЕННОГО УНИВЕРСИТЕТА
ИМЕНИ М. В. ЛОМОНОСОВА В ГОРОДЕ САРОВЕ

Параллельные методы решения задач

Лабораторная работа:

«Многопоточная реализация операций с сеточными данными на неструктурированной смешанной сетке, решение СЛАУ»

Выполнил студент
1-го курса магистратуры
Козлов Н.М.
Группа: ВМ-124
Дата подачи: 09.04.2025
Вариант: А2

Саров 2024

2.1. Описание задачи

Цель данного задания: реализация решения СЛАУ, хранящегося в CSR формате с использованием технологии OpenMP.
Поэтому нужно реализовать:
1) реализация вспомогательных функций
2) решение СЛАУ в CSR формате методом сопряжённых градиентов.
2) параллелизация участков алгоритмов и вспомогательных функций с использованием OpenMP
3) вывод отчёта о времени работы вспомогательных функций и этапов программы

1.2 Описание программной реализации

Для решения задания были добавлены следующие функции:

```
//Скалярное произведение двух векторов
	double dot(const double* a, const double* b, const int N)
//Линейная комбинация векторов
	void axpy(const double* a, const double beta, const double* b, double*& res, const int N)
//Умножение матрицы на вектор с применением OpenMP
	void SpmV(const int* IA, const int* JA, const double*A, const double* v, double*& res, const int N)
//Решение СЛАУ методом сопряжённых гардиентов с применением OpenMP
void solve(const int* IA, const int* JA, const double* A, const double* b, const int N, const double eps, const int maxit, const bool norms, double*& x, int& step, double& norm)
//Проверка корректности: вывод невязки, данные о времени работы всех этапов программы и вспомогательных функций
void report(double norm, double timeG, double timeF, double timeS, double dott, double axpyt, double SpmVt, int N, int nnz)
```

Для параллелизации вспомогательных функций использовались:

dot: `#pragma omp parallel for reduction(+:sum) default(shared) schedule(static)`

Обусловлено тем, что reduction быстрее всего справляется с предоставлением доступа к общей памяти в данном случае. Расписание статическое, поскольку всем процессам предоставляются одинаково простые задачи.

axpy: `#pragma omp parallel for schedule(static) default(none) shared(N, a, res, beta, b)`

Поскольку функция применялась в различных ситуациях, все три вектора являются разными сущностями, раздача нитям также происходит статично из-за простоты операций.

SpMV: `#pragma omp parallel for schedule(dynamic, 100)`

Здесь уже динамичное расписание, поскольку столбцы в строках распределены неравномерным образом, соответственно выгоднее использовать динамичный способ, причём с указанием чанка. Кроме того, на том же принципе реализована параллелизация fill.

На этапе solve помимо распараллеленных вспомогательных функций ускоряется инициализация нулями и копирование массивов. Норма невязки считается, как квадратный корень из скалярного произведения вектора r с самим собой.

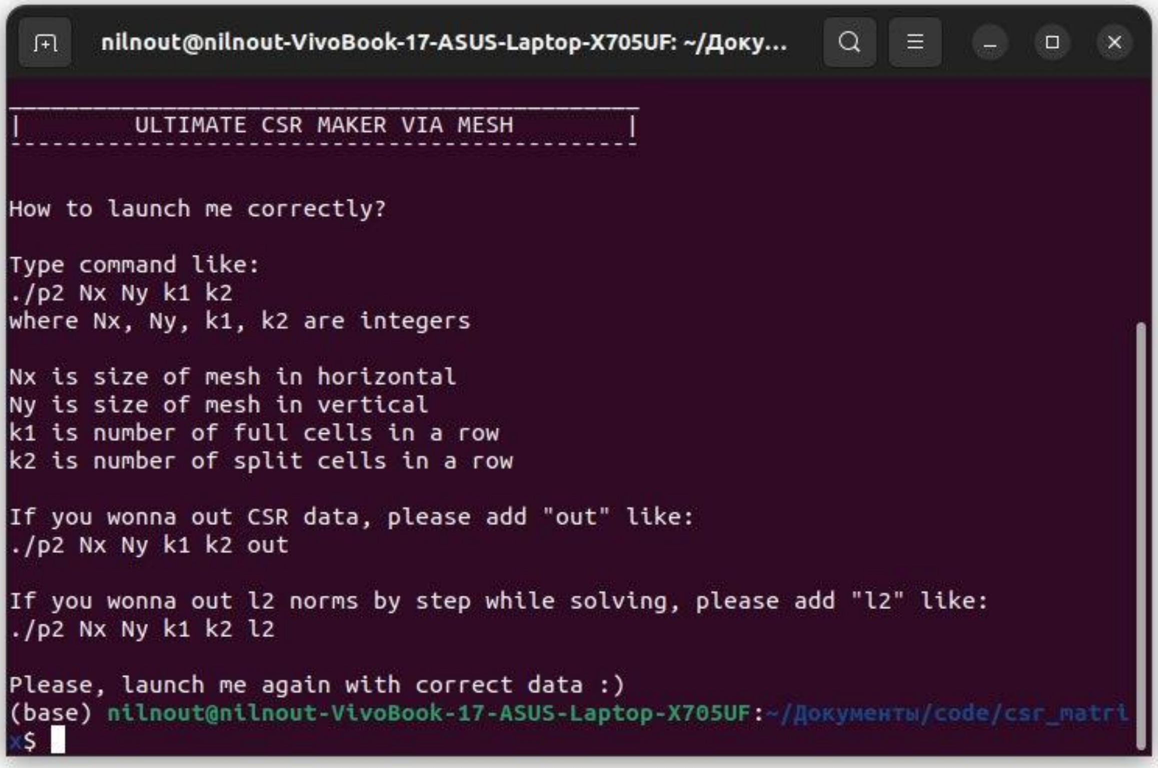
Для компиляции программы необходимо иметь компилятор g++ (устанавливается следующей командой: `sudo apt install build-essential`).

Компиляция происходит с помощью следующей команды (при терминале, открытом в папке с файлом программы):

```
module load gcc/gcc-12.2
gcc main.cpp -o program -fopenmp
sbatch conf.sb
```

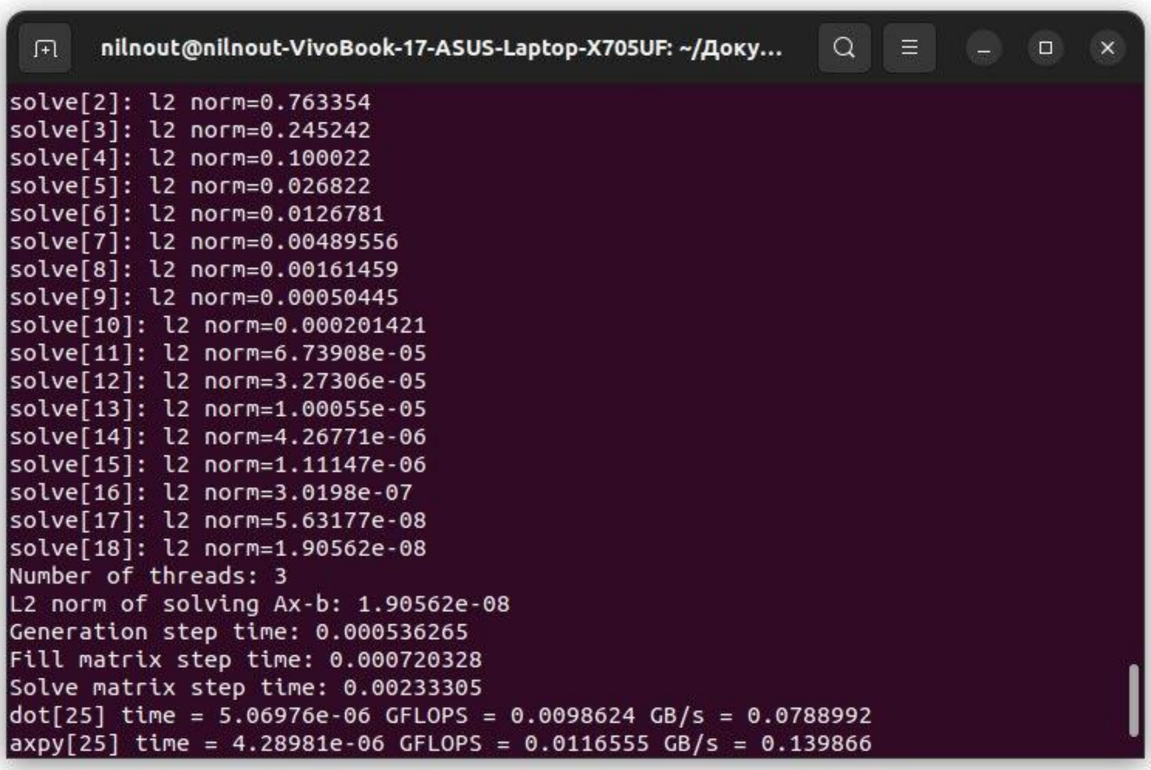
```
#!/bin/bash
#SBATCH --time = 59:00 // время работы программы
#SBATCH --nodes=1 // число узлов
#SBATCH --ntasks=1 // число процессов
#SBATCH --ntasks-per-node=1 // сколько процессов на 1 узел
#SBATCH --cpus-per-task=8 // число ядер на 1 процесс
#SBATCH --output=prog_%j.out // имя файла для stdout
#SBATCH --error=prog_%j.err // имя файла для stderr
export OMP_PROC_BIND="close"
export OMP_PLACES="cores"

export OMP_NUM_THREADS=4
./pr 100 100 4 5
./pr 1000 100 4 5
./pr 1000 1000 4 5
./pr 10000 1000 4 5
./pr 10000 10000 4 5
```

Рисунок 1 – batch-файлик для запуска на кластере



```
nilnout@nilnout-VivoBook-17-ASUS-Laptop-X705UF: ~/Доку...
_____________________________________________
|        ULTIMATE CSR MAKER VIA MESH         |
---------------------------------------------

How to launch me correctly?

Type command like:
./p2 Nx Ny k1 k2
where Nx, Ny, k1, k2 are integers

Nx is size of mesh in horizontal
Ny is size of mesh in vertical
k1 is number of full cells in a row
k2 is number of split cells in a row

If you wonna out CSR data, please add "out" like:
./p2 Nx Ny k1 k2 out

If you wonna out l2 norms by step while solving, please add "l2" like:
./p2 Nx Ny k1 k2 l2

Please, launch me again with correct data :)
(base) nilnout@nilnout-VivoBook-17-ASUS-Laptop-X705UF:~/Документы/code/csr_matri
x$
```

Рисунок 2 – Запуск программы без аргументов



```
nilnout@nilnout-VivoBook-17-ASUS-Laptop-X705UF: ~/Доку...
solve[2]: l2 norm=0.763354
solve[3]: l2 norm=0.245242
solve[4]: l2 norm=0.100022
solve[5]: l2 norm=0.026822
solve[6]: l2 norm=0.0126781
solve[7]: l2 norm=0.00489556
solve[8]: l2 norm=0.00161459
solve[9]: l2 norm=0.00050445
solve[10]: l2 norm=0.000201421
solve[11]: l2 norm=6.73908e-05
solve[12]: l2 norm=3.27306e-05
solve[13]: l2 norm=1.00055e-05
solve[14]: l2 norm=4.26771e-06
solve[15]: l2 norm=1.11147e-06
solve[16]: l2 norm=3.0198e-07
solve[17]: l2 norm=5.63177e-08
solve[18]: l2 norm=1.90562e-08
Number of threads: 3
L2 norm of solving Ax-b: 1.90562e-08
Generation step time: 0.000536265
Fill matrix step time: 0.000720328
Solve matrix step time: 0.00233305
dot[25] time = 5.06976e-06 GFLOPS = 0.0098624 GB/s = 0.0788992
axpy[25] time = 4.28981e-06 GFLOPS = 0.0116555 GB/s = 0.139866
```

Рисунок 3 – Запуск программы с флагом ‘l2’

3.1 Описание компьютерной платформы

Intel Xeon Gold 6140 2.3 GHz.
TPP=2,3 GHz *18 cores *32 IPS (16 via AVX512 * 2 via FMA3) =1.3248 TFLOPS
BW=2,666 GHz *8Б*6 *2=256GB/c

3.2 Результат вычислительных экспериментов

Согласно заданию, были произведены замеры производительности работы вспомогательных функции и солвера при разном размере массивов. График зависимости продемонстрирован на рисунке 4. При вычислении было проведено усреднение: для вспомогательных функций по 100 запускам, для солвера – по 10.

Как видно из графика, с увеличением размера массива производительность функций выходит на плато.

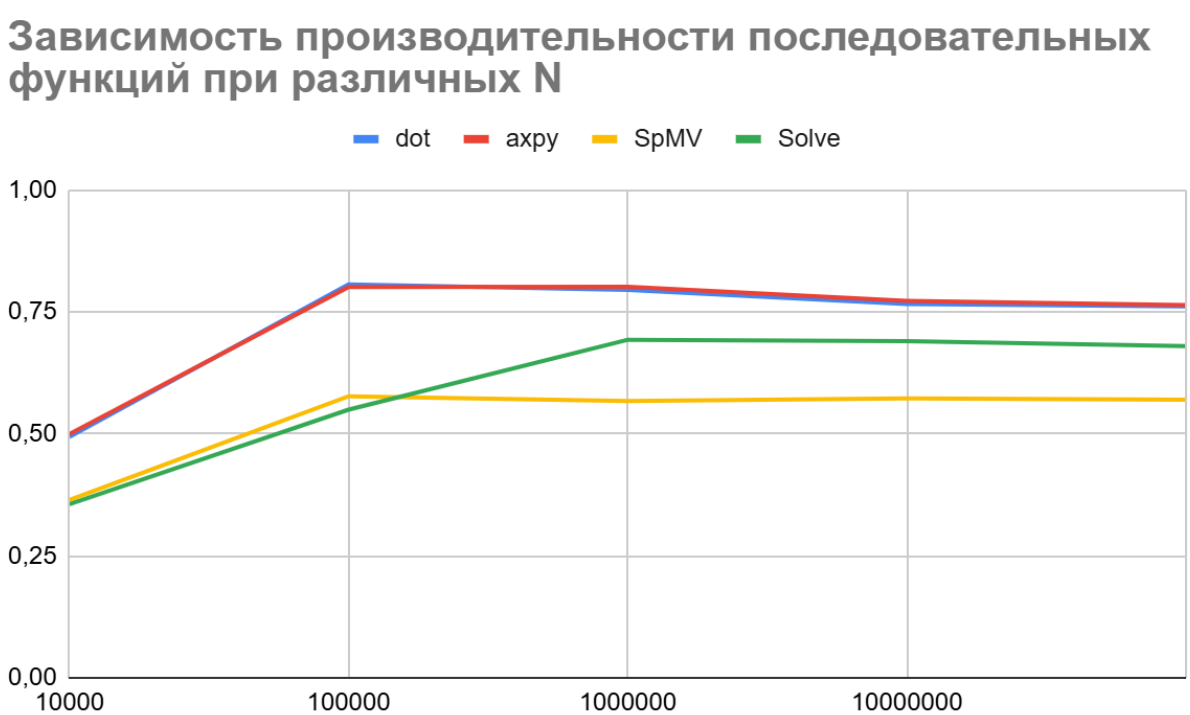


Рисунок 4 – График зависимости производительности функций при последовательном запуске в зависимости от параметра N

Формулы расчёта числа операций с двойной точностью:
dot: `2*N (умножение + сложение)`
axpy: `2*N (умножение + сложение)`
SpMV: `2.0*IA[N] (умножение + сложение)`

(в случае, если диагональный SpMV: `2*N (умножение + предварительное деление)`)

Solve: `14.0*N+2.0*IA[N] (3*dot+3*axpy+spmv-a+spmv-m)`

Теперь посмотрим на ускорение алгоритма с увеличением числа нитей. Это продемонстрировано рисунками 5-9.

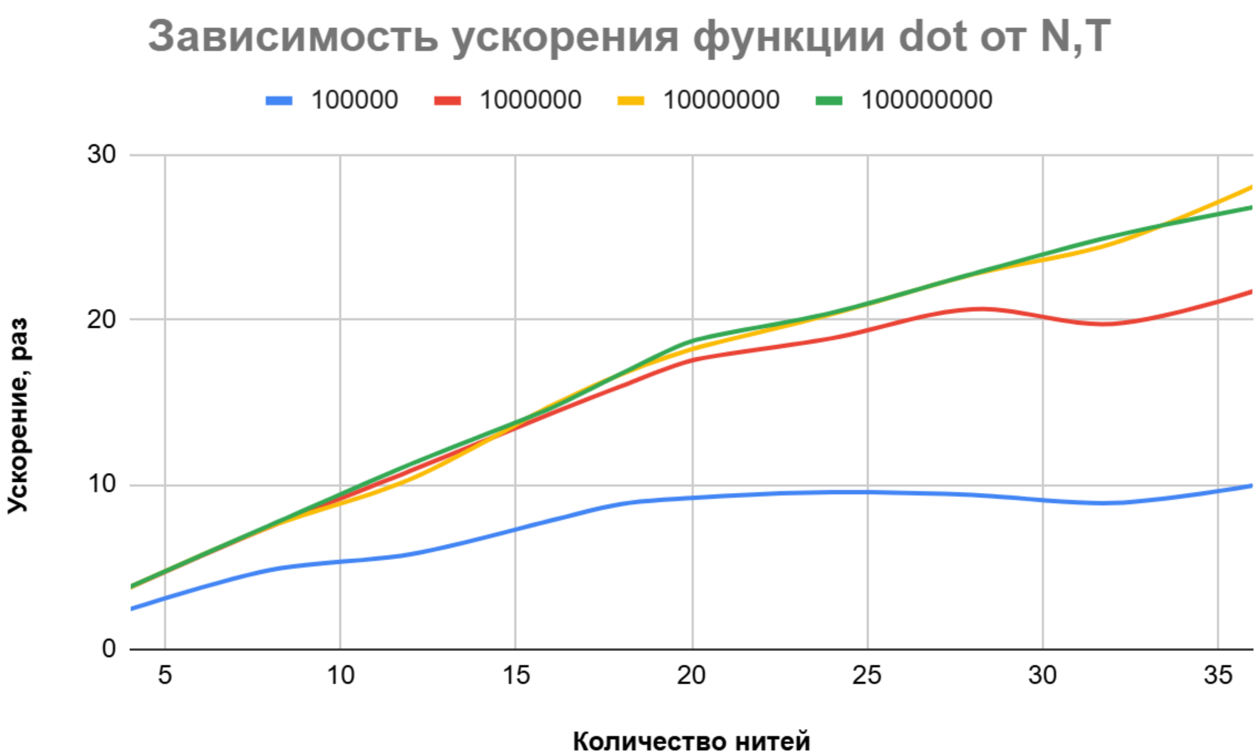


Рисунок 5 – График зависимости ускорения функции dot в зависимости от параметров N, T

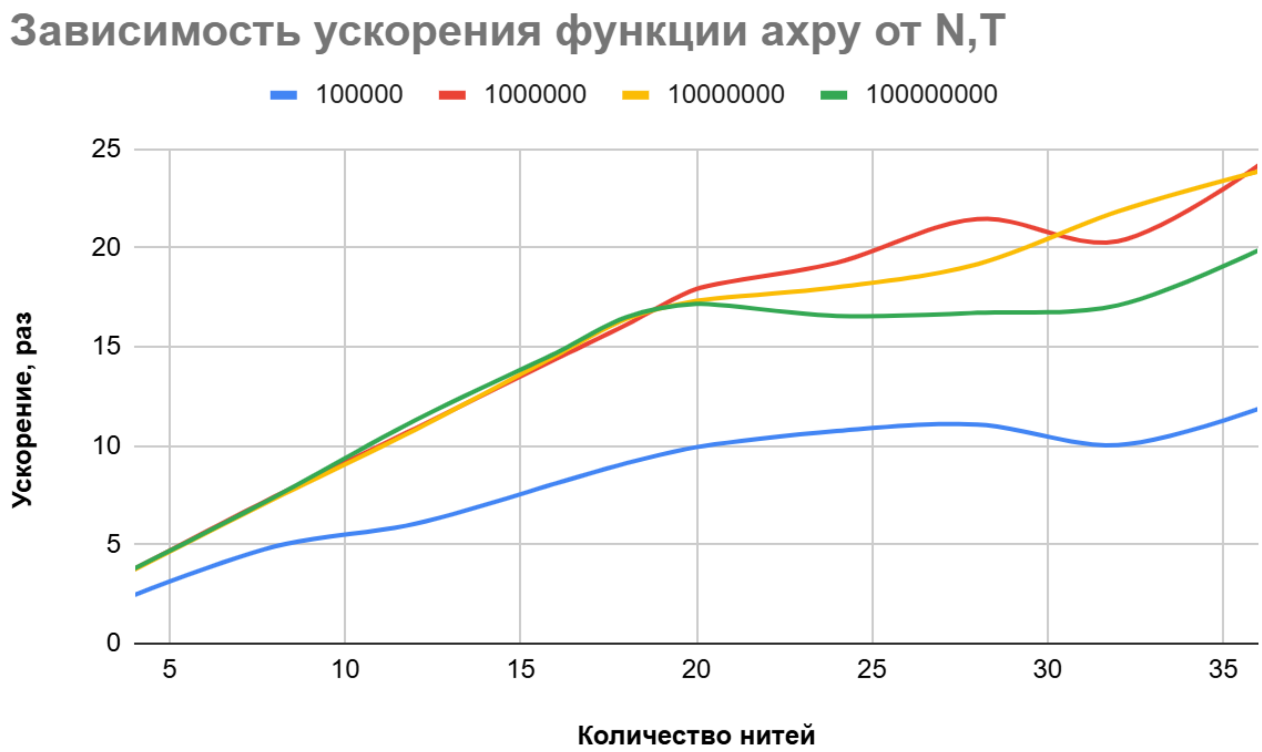


Рисунок 6 – График зависимости ускорения функции axpy в зависимости от параметров N, T

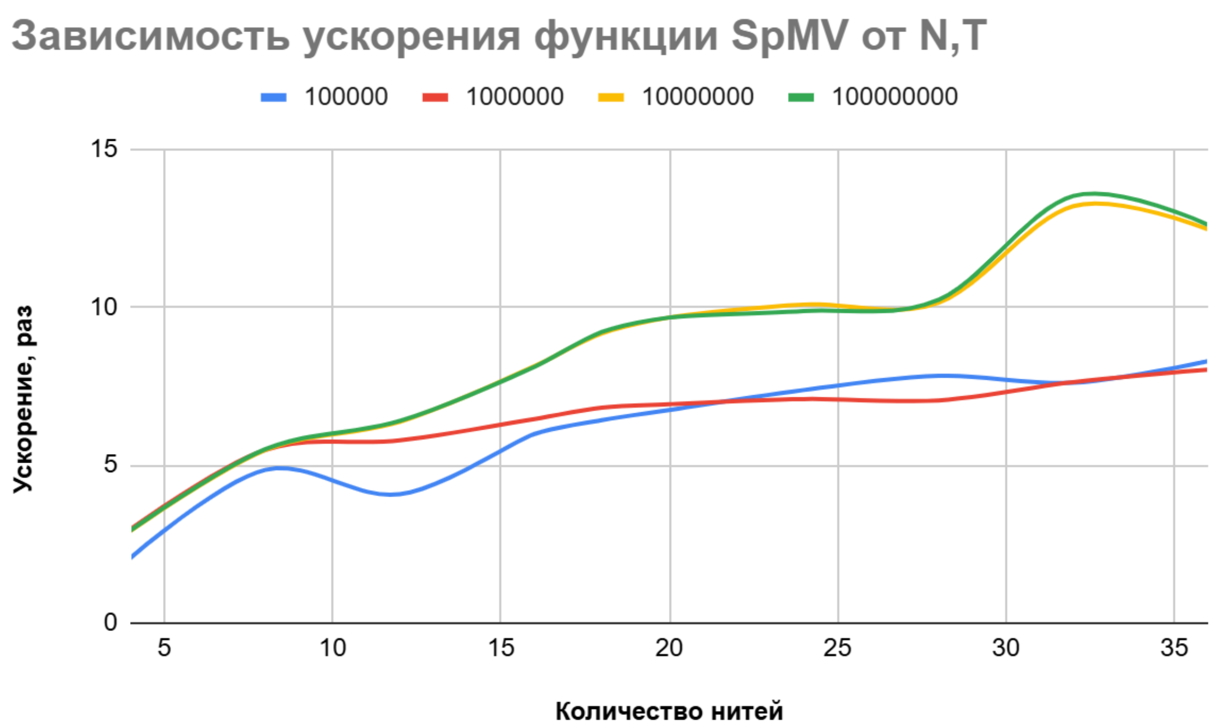


Рисунок 7 – График зависимости ускорения функции SpMV в зависимости от параметров N, T

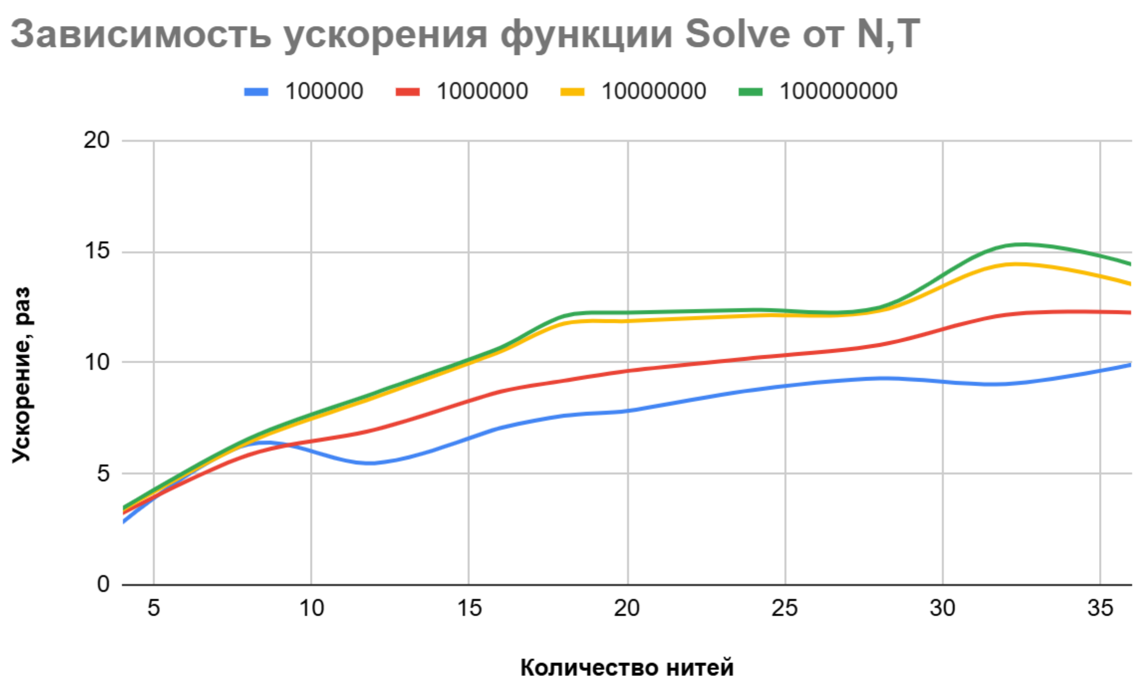


Рисунок 8 – График зависимости ускорения функции solve в зависимости от параметров N, T

Итак, начнём с хороших новостей – ускорение получено! Во всех случаях! Причём в общем случае ускорение растёт с увеличением числа процессов. Для 4, 8, 32 и 36 нитей была использована опция «close». В остальных же – «spread».

Но и это ещё не всё! На рисунке 9 показано ускорение функции fill от числа нитей и размера массива.

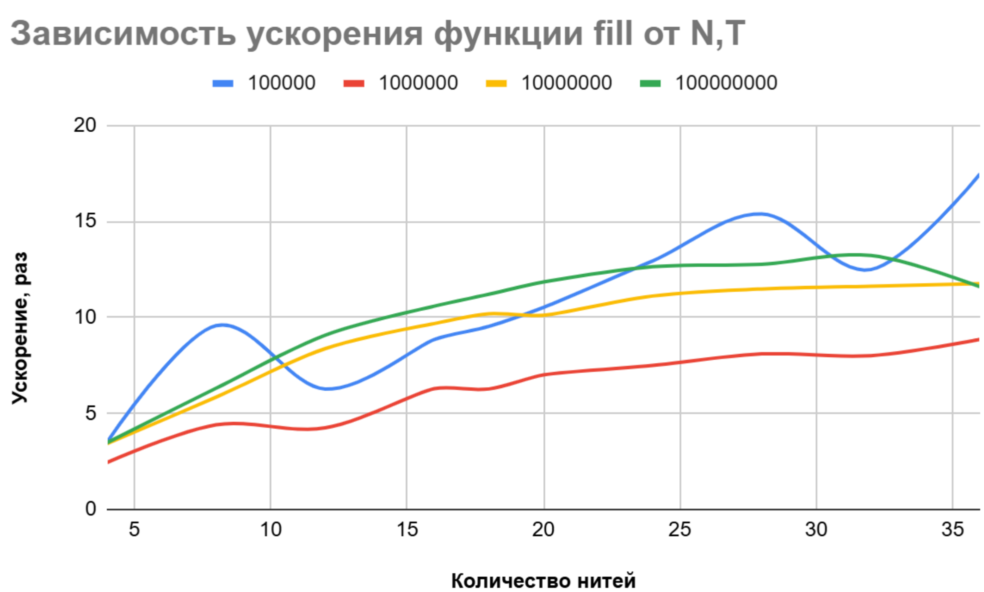


Рисунок 9 – График зависимости ускорения функции fill в зависимости от параметров N, T

На рисунках 10-13 продемонстрирована зависимость производительности функций от размера массивов и числа нитей.

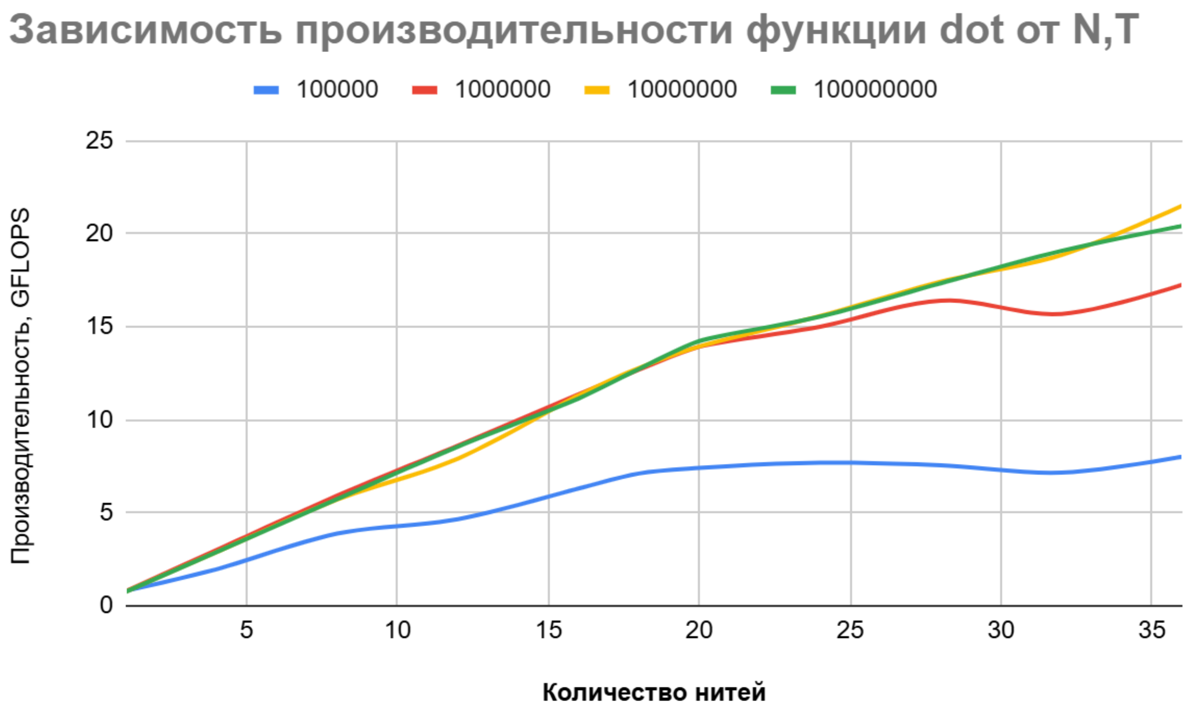


Рисунок 10 – График зависимости производительности функции dot в зависимости от параметров N, T

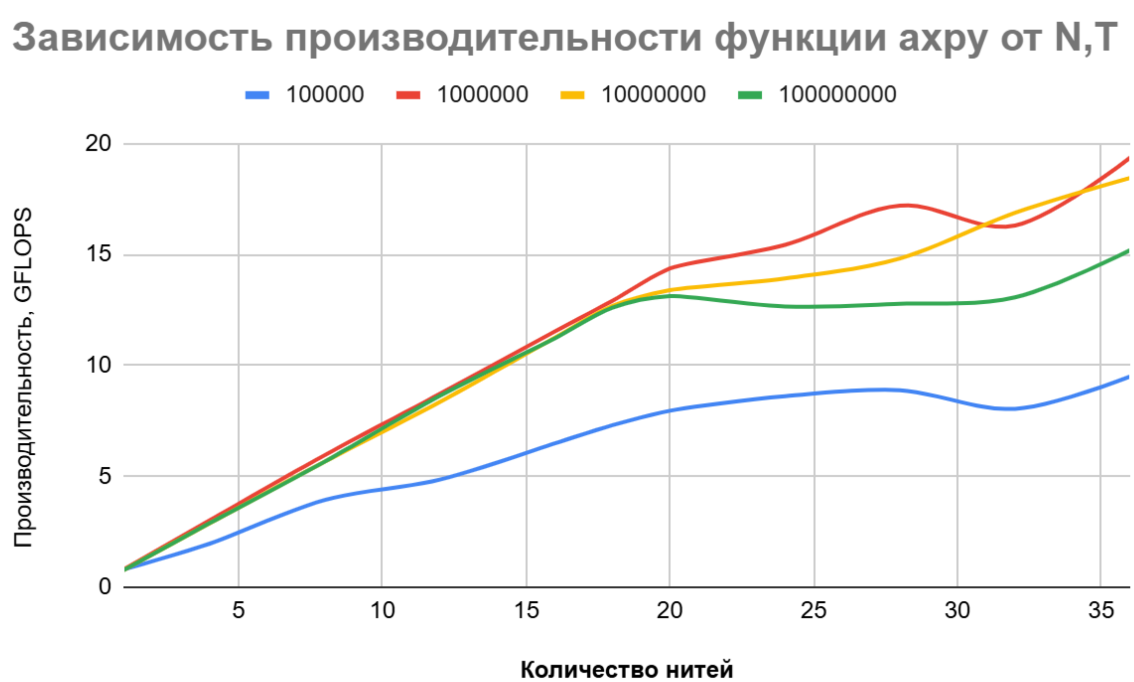


Рисунок 11 – График зависимости производительности функции axpy в зависимости от параметров N, T

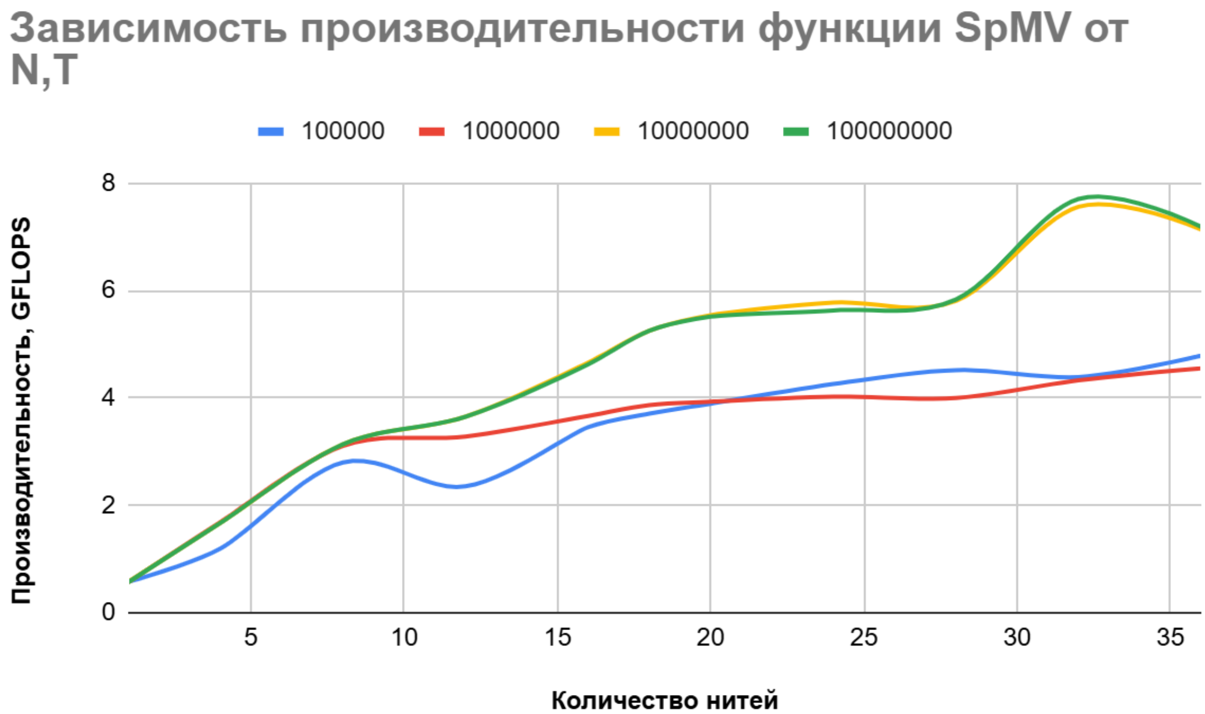


Рисунок 12 – График зависимости производительности функции SpMV в зависимости от параметров N, T

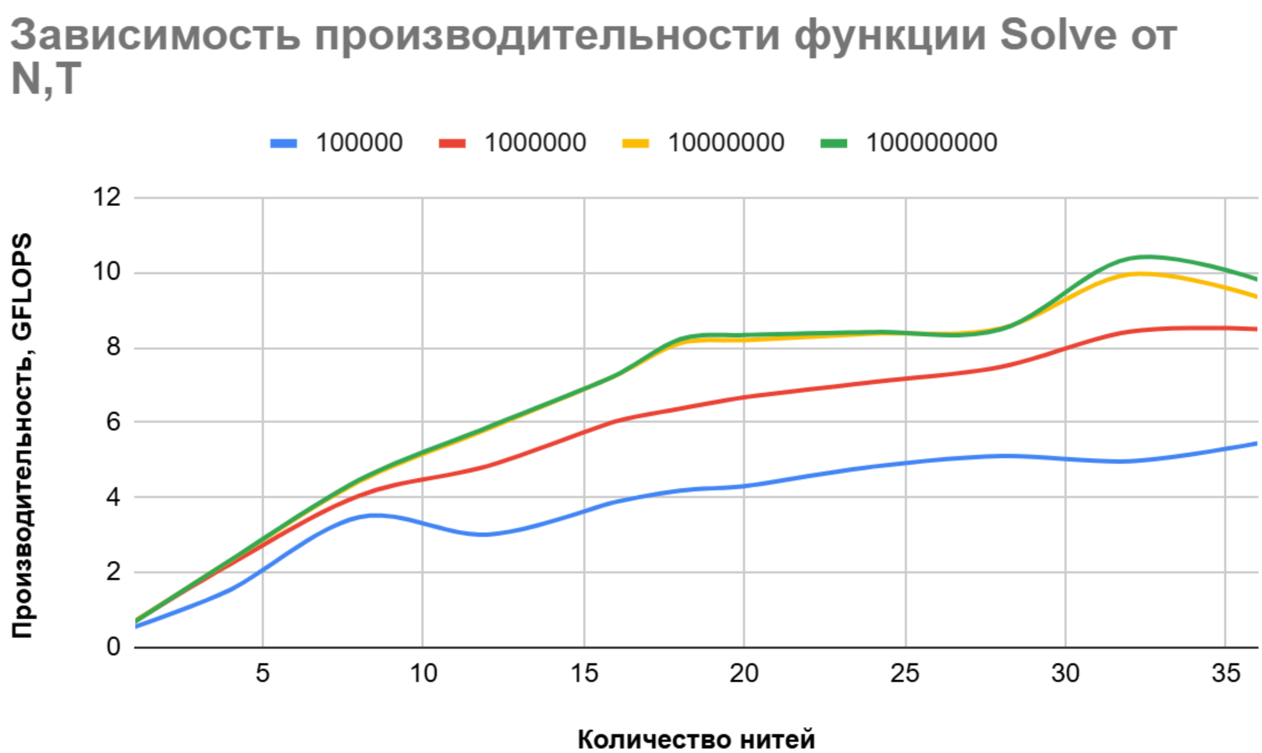


Рисунок 13 – График зависимости производительности функции Solve в зависимости от параметров N, T

Как видно на рисунках 10-13, производительность функций растёт с увеличением числа процессов, причём лучше всего себя показала функция dot.

Анализ результатов

Вспомним формулу расчёта TBP и характеристики системы:

TBP: min(TPP, BW*AI)

TPP=1324.8 GFLOPS

BW=256 GB/c

TBP_{dot}: AI $= \frac{2\times N \text{ операции}}{2\times 8\times N \text{ (чтение)}} = \frac{1}{8}$.

TBP_{dot} = min(TPP, $\frac{256}{8}$)=32 GFLOPS **(2.4%)**

$Rmax_{dot}$ = 21,5146 GFLOPS (***67%*** от TBP_{dot} и **1.62%** от TPP).

TBP_{axpy}: AI $= \frac{2\times N \text{ операции}}{2\times 8\times N \text{ (чтение)} + 8\times N \text{ (запись)}} = \frac{1}{12}$.

TBP_{axpy} = min(TPP, $\frac{256}{12}$)=21,3 GFLOPS **(1.61%)**

Rmax$_{axpy}$ = 19,3679 GFLOPS (***90%*** от TBP_{axpy} и **1.46%** от TPP).

TBP_{SpMV}: AI $= \frac{2\times NNZ \text{ операции}}{8\times NNZ \text{ (чтение A)} + 4\times NNZ \text{ (чтение JA)} + 8\times NNZ \text{ (чтение v[JA[k]])} + 4\times N \text{ (чтение IA)} + 8\times N \text{ (запись res)}}$,

Где $NNZ = O(N)$. Проведём оценку. Пусть $NNZ = k \times N$. Тогда:

AI $= \frac{2\times NNZ}{20\times NNZ\ + 12\times N} = \frac{k\times N}{10\times k\times N + 6\times N} = \frac{1}{10+\frac{6}{k}}$.

$$NNZ = (Nx\ +\ 1)\ *\ (Ny\ +\ 1) + 2\ *\ (Nx\ *\ (Ny\ +\ 1)\ + (Nx\ *\ Ny)\,/\,(k1\ +\ k2)\ *\ k2\ + \text{копейки})$$

$$N = (Nx\ +\ 1)\ *\ (Ny\ +\ 1),$$

$$k = \frac{NNZ}{N} \approx 5.$$

Тогда AI $= \frac{1}{11.2}$.

TBP_{SpMV} = min(TPP, $\frac{256}{11.2}$) = 22,857 GFLOPS **(1,725%)**

Rmax$_{SpMV}$ = 7,70629 GFLOPS (**33,7%** от TBP_{SpMV} и 0.58% от TPP).

Итак, о Sustained performance… Лучше всего себя показала функция dot, что и неудивительно, глядя на TBP. Практические оценки dot и axpy оказались близко к теоретической, что говорит о хорошей реализации параллелизации этих алгоритмов. Что касается SpMV, подозреваю, что правильный подбор размера чанка в динамическом расписании мог бы улучшить итак неплохие результаты.

Закончу отчёт изначальной фразой, ведь как ни крути, победа программиста сегодня только в паре процентов производительности.

Каков итог? Чертовски мало, сэр! Но такова жизнь, таков код, таков кластер.