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

Практическое задание №1

Многопоточная реализация солвера `BiCGSTAB` для СЛАУ с разреженной матрицей, заданной в формате `CSR`

выполнила: Мацак Алиса Игоревна, 524 группа

дата подачи: 15.11.2018 г.

Москва, 2018 г.

Содержание

1. Описание задания и программной реализации

1.1. Краткое описание задания

- Сделать генератор матрицы для расчетной области, представленной трехмерной декартовой решеткой заданного размера `Nx`, `Ny`, `Nz`; проверить корректность заполнения матрицы.
- Сделать исходные последовательные реализации операций скалярного произведения векторов, линейной комбинации векторов, умножения матрицы на вектор. Сделать проверочный тест, выполняющий это операции.
- Сделать солвер `BiCGSTAB` на основе реализованных операций.
- Сделать многопоточные параллельные реализации базовых операций. Получить многопоточную корректно работающую версию солвера.

1.2. Краткое описание программной реализации

Программа состоит из следующих файлов:

- `csr_matrix.hpp` — содержит реализацию матрицы в формате `CSR`.

  Класс `CompressedSparseRowMatrix` содержит методы:

  - `int size()` — возвращает размерность матрицы;
  - `void printDataStartRow()` — выводит вектор IA (обозначение взято из задания);
  - `void printColumnNumbersRow()` — выводит вектор `JA` (обозначение взято из задания);
  - `void printElementsRow()` — выводит вектор `A` (обозначение взято из задания);
  - `void print()` — печатает матрицу целиком;
  - `void changeValue (int row, int column, double element)` — изменяет ненулевое значение матрицы `(row, column)` на `element`;
  - `double pushBack(int row, int column, double element)` — кладет `element` и сопутствующую информацию в конец векторов `IA`, `JA`, `A` при построчном заполнении матрицы, `element` в результате становится на место `(row, column)`;
  - `double pushBackSinRowColumn(int row, int column)` — кладет `element = sin(row, column)` и сопутствующую информацию в конец векторов `IA`, `JA`, `A` при построчном заполнении матрицы, element в результате становится на место `(row, column)`;

- `double getElement(int row, int column)` — выдает значение элемента матрицы по месту `(row, column)`, если оно существует;
- `void addNumberOfElements()` — кладет количество ненулевых элементов в конец вектора `IA`;
- `double getNumberOfElements()` — возвращает количество ненулевых элементов в матрице;
- `std::vector<double> & matrixVectorProduct(std::vector<double> & x, std::vector<double> & y)` — умножение матрицы на вектор `X`, результат записывается в вектор `y`;
- `std::vector<double> & matrixVectorProduct_par(std::vector<double> & x, std::vector<double> & y)` — умножение матрицы на вектор `X`, результат записывается в вектор `y`, *параллельная версия*.

■ `lin_algebra.hpp` — содержит операции скалярного произведения векторов и линейной комбинации векторов.

Файл содержит функции:

- `double dotProduct(std::vector<double> & x, std::vector<double> & y)` — скалярное произведение вектора `x` на вектор `y`;
- `double dotProduct_par(std::vector<double> & x, std::vector<double> & y)` — скалярное произведение вектора `x` на вектор, *параллельная версия*;
- `std::vector<double> & linearCombination(std::vector<double> & x, std::vector<double> & y, double a, double b)` — линейная комбинация векторов: `x = a*x + b*y`;
- `std::vector<double> & linearCombination_par(std::vector<double> & x, std::vector<double> & y, double a, double b)` — линейная комбинация векторов: `x = a*x + b*y`, *параллельная версия*.

■ `test_basic_operations.hpp` — содержит единственную функцию `void testBasicOperationsSpeedup(int nx, int ny, int nz)`, которая тестирует ускорение базовых операций.

■ `main_with_par.cpp` — содержит параллельную и обычную версии солвера BiCGSTAB.

■ `parse_args.hpp` — содержит функцию парсинга командной строки.

- `generator.hpp` — содержит классы—генераторы матриц в формате CSR: генератор матрицы, формат значений которой описан в задании и диагональной, элементы которой обратны диагональным элементам матрицы, которая подается генератора на входе.
- `vector_operations_tests.cpp` — содержит гугл тесты корректности базовых операций.

1.2.1. Сборка и запуск

➔ Программа собирается командой:
```
> make main
```

➔ Запустить программу:
`> ./main` (+ работают параметры командной строки, описанные в задании)

➔ Собрать тесты корректности базовых операций:
```
> make test
```

➔ Запустить тесты корректности базовых операций:
```
> ./test
```

➔ Удалить `main` и `test`:
```
> make clean
```

Чтобы файл `test` скомпилировался, надо скачать `googletest`:

```
> git clone https://github.com/google/googletest.git
> cd googletest
> cmake .
> make
```

2. Исследование производительности

2.1. Характеристики вычислительной системы

Intel® Core™ i3-2330M CPU
# of Cores 2
# of Threads 4
Processor Base Frequency 2.20 GHz
Cache 3 MB L3
Memory Size 3.8 GB
Max Memory Bandwidth 21.3 GB/s

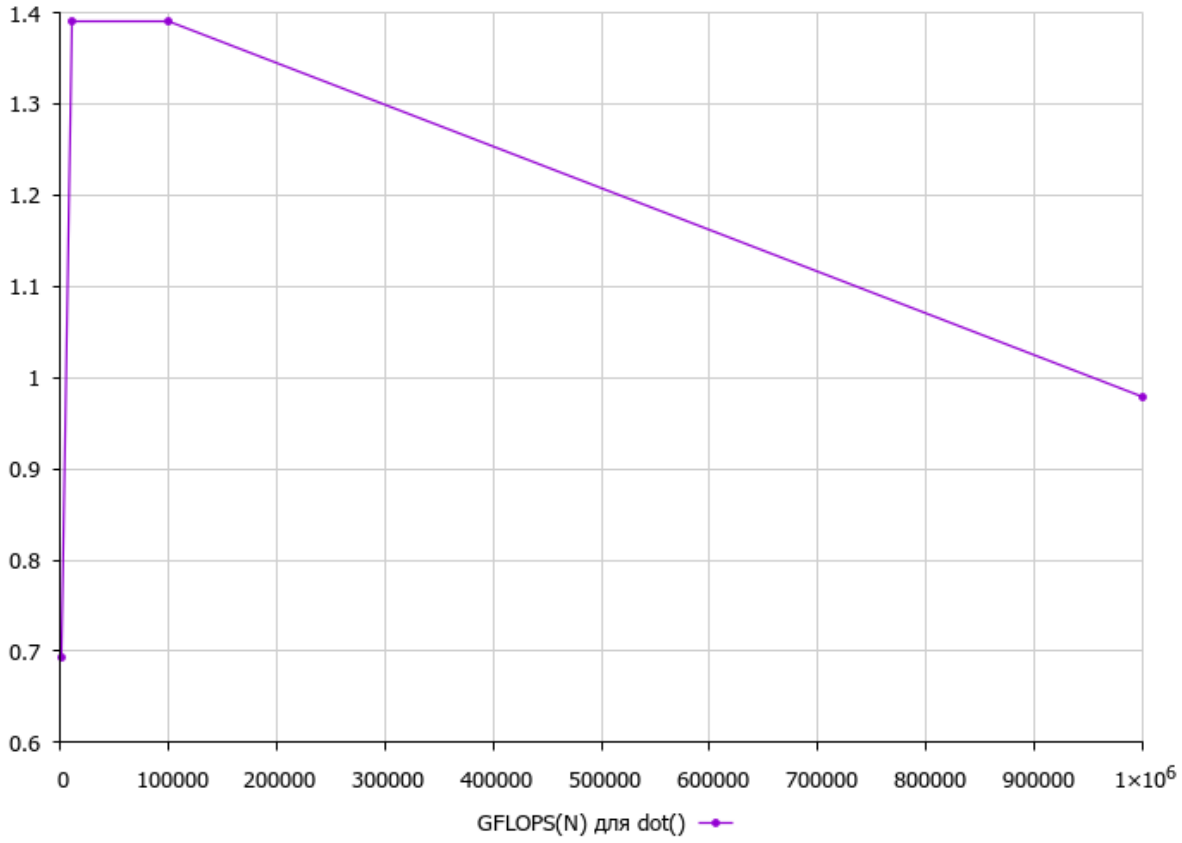
2.2. Результаты измерений производительности

2.2.1. Последовательная производительность

Базовая функция `dot()`

	time	GFLOPS
N = 1000	0.00115s	0.693
N = 10000	0.00574s	1.39
N = 100000	0.0573s	1.39
N = 1000000	0.817s	0.979

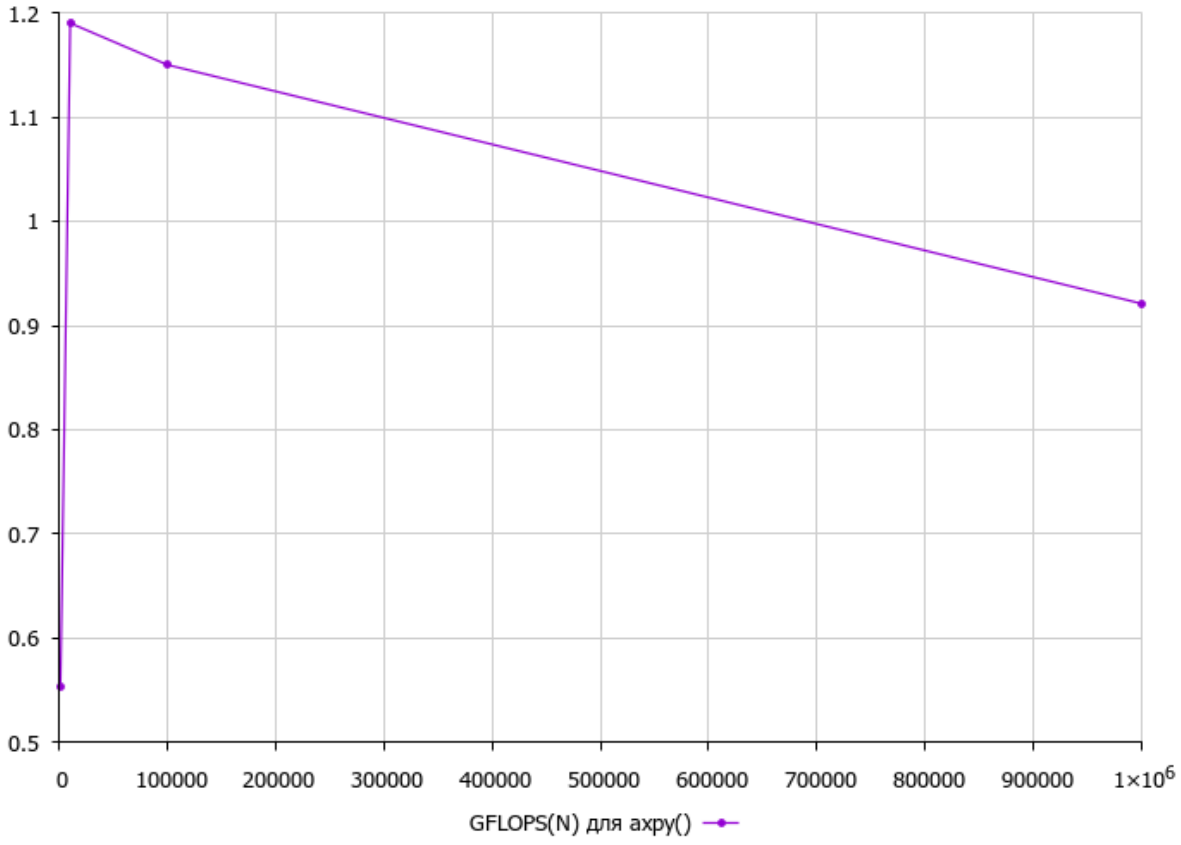
График `GFLOPS(N)` для `dot()`



Базовая функция `axpy()`

	time	GFLOPS
N = 1000	0.00217s	0.553
N = 10000	0.0101s	1.19
N = 100000	0.104s	1.15
N = 1000000	1.3s	0.921

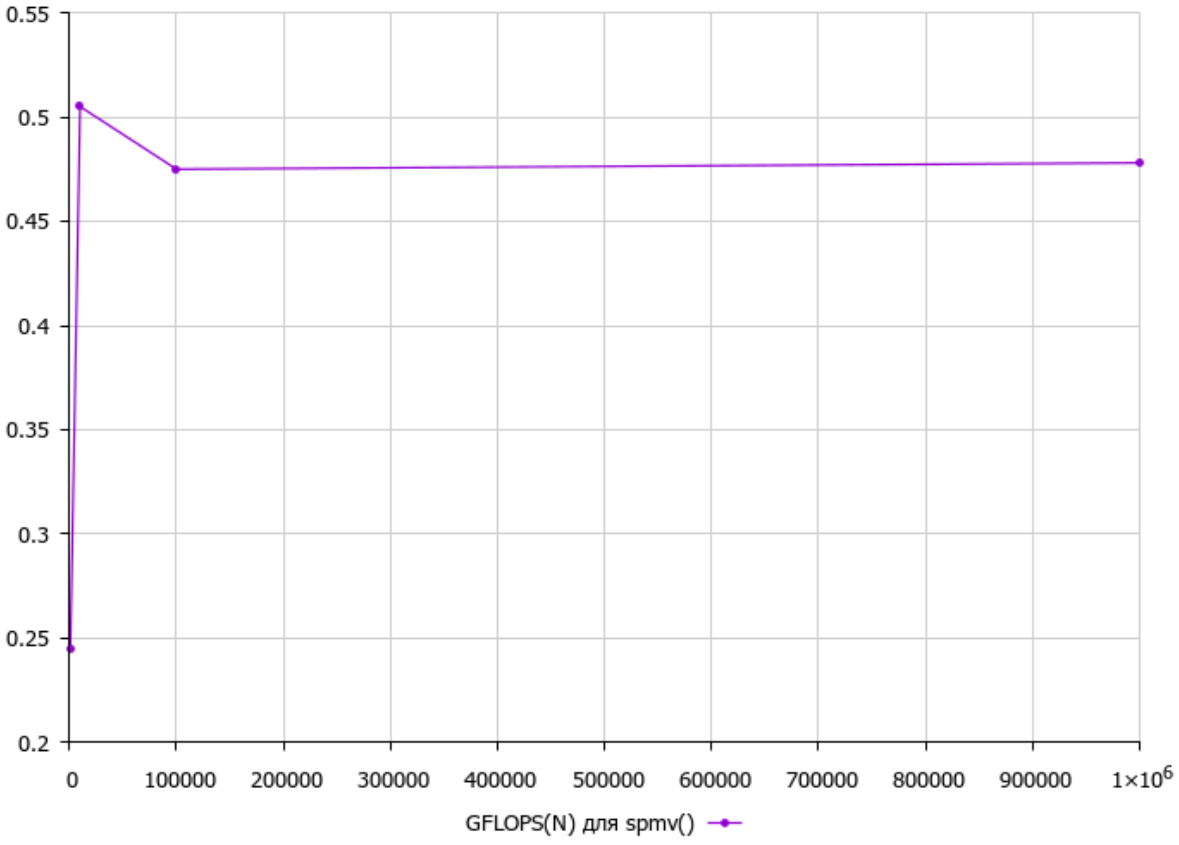
График `GFLOPS(N)` для `axpy()`



Базовая функция `spmv()`

	time	GFLOPS
N = 1000	0.0209s	0.245
N = 10000	0.104s	0.505
N = 100000	1.14s	0.475
N = 1000000	11.6s	0.478

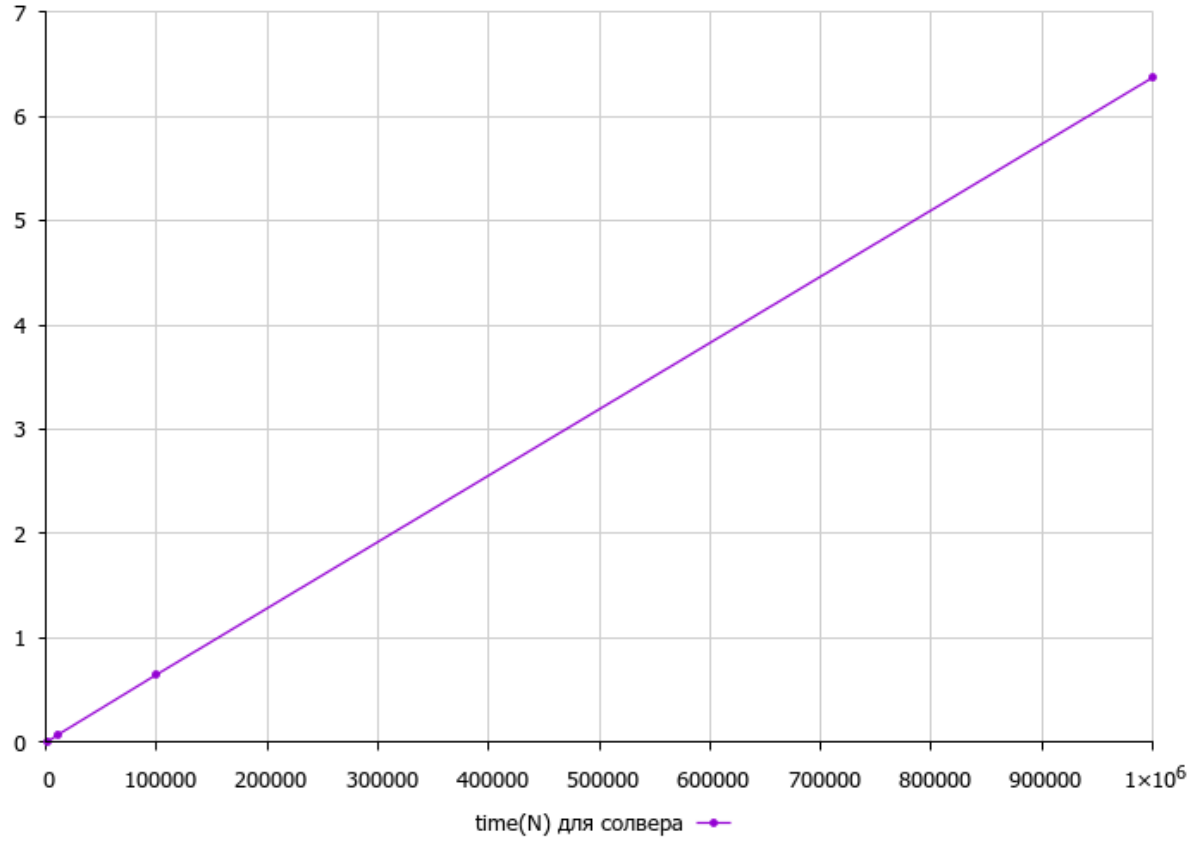
График `GFLOPS(N)` для `spmv()`



Весь алгоритм солвера

	time
N = 1000	0.00685
N = 10000	0.0719
N = 100000	0.653
N = 1000000	6.37

График time(N) для всего алгоритма солвера



2.2.2. Параллельное ускорение

N = 1000

	dot()	axpy()	smpv()	solver
2 threads	0.756X	0.929X	1.69X	1.137X
4 threads	1.3X	0.472X	3.82X	0.391X

N = 10000

	dot()	axpy()	smpv()	solver
2 threads	1.65X	1.33X	1.65X	1.041X
4 threads	2.15X	0.943X	1.65X	1.149X

N = 100000

	dot()	axpy()	smpv()	solver
2 threads	1.8X	1.33X	1.62X	1.05X
4 threads	2.72X	1.46X	1.77X	1.05X

N = 1000000

	dot()	axpy()	smpv()	solver
2 threads	1.12X	1.1X	1.38X	1.04X
4 threads	1.18X	1.18X	1.83X	1.03X

3. Анализ полученных результатов

3.1. Процент от пиковой производительности

Пиковая производительность была посчитана теоретически по следующей формуле (CPU IPC взято отсюда, из строки с Sandy Bridge):

```
(CPU speed in GHz) x (number of CPU cores) x (CPU instruction
per cycle) x (number of CPUs per node) =
2.20 * 2 * 8 * 1 = 35 GFLOPS
```

Для базовых операций брались максимальные значения GFLOPS из всех тестов, отличавшихся по количеству нитей (тредов) и размеру сетки:

```
axpy = 1.66 GFLOPS
dot = 3.89 GFLOPS
spmv = 0.914 GFLOPS
```

Итого процент от пика:

```
axpy = 0.05%
dot = 0.1%
spmv = 0.03%
```

3.2. Процент от достижимой производительности

Достижимая производительность была посчитана с помощью бенчмарка LINPACK:

```
CPU frequency:      2.194 GHz
Number of CPUs: 1
Number of cores: 2
Number of threads: 2

Parameters are set to:

Number of tests: 9

Number of equations to solve (problem size) : 15000 14000 13000 12000 11000 10000
8000  6000  1000

Leading dimension of array                    : 15000 14008 13000 12008 11000
10008 8008  6008  1000

Number of trials to run                       : 1    2      2      2      2      2
       2      3      4

Data alignment value (in Kbytes)              : 4    4      4      4      4      4
       4      4      4

Maximum memory requested that can be used=64426309744096, at the size=4151455638

=================== Timing linear equation system solver ===================

Size   LDA  Align. Time(s) GFlops   Residual      Residual(norm) Check
```

```
15000  15000  4     91.085       24.7071  2.111083e-10 3.324990e-02   pass
14000  14008  4     75.791       24.1417  1.937182e-10 3.498344e-02   pass
14000  14008  4     75.676       24.1783  1.937182e-10 3.498344e-02   pass
13000  13000  4     60.867       24.0689  1.570904e-10 3.286999e-02   pass
13000  13000  4     60.870       24.0677  1.570904e-10 3.286999e-02   pass
12000  12008  4     48.013       23.9995  1.246899e-10 3.060496e-02   pass
12000  12008  4     48.137       23.9374  1.246899e-10 3.060496e-02   pass
11000  11000  4     36.894       24.0576  1.217657e-10 3.552412e-02   pass
11000  11000  4     37.246       23.8302  1.217657e-10 3.552412e-02   pass
10000  10008  4     28.084       23.7458  8.915707e-11 3.143769e-02   pass
10000  10008  4     28.158       23.6829  8.915707e-11 3.143769e-02   pass
8000   8008   4     14.604       23.3807  6.860456e-11 3.773846e-02   pass
8000   8008   4     14.342       23.8083  6.860456e-11 3.773846e-02   pass
6000   6008   4     6.279        22.9436  4.341994e-11 4.210803e-02   pass
6000   6008   4     6.358        22.6605  4.341994e-11 4.210803e-02   pass
6000   6008   4     6.277        22.9536  4.341994e-11 4.210803e-02   pass
1000   1000   4     0.047        14.1952  9.776248e-13 3.333952e-02   pass
1000   1000   4     0.047        14.2838  9.776248e-13 3.333952e-02   pass
1000   1000   4     0.047        14.2561  9.776248e-13 3.333952e-02   pass
1000   1000   4     0.047        14.2871  9.776248e-13 3.333952e-02   pass

Performance Summary (GFlops)

Size   LDA    Align.  Average  Maximal
15000  15000  4     24.7071  24.7071
14000  14008  4     24.1600  24.1783
13000  13000  4     24.0683  24.0689
12000  12008  4     23.9685  23.9995
11000  11000  4     23.9439  24.0576
10000  10008  4     23.7144  23.7458
8000   8008   4     23.5945  23.8083
6000   6008   4     22.8526  22.9536
1000   1000   4     14.2556  14.2871

Residual checks PASSED

End of tests
```

Отсюда видно, что она составляет `24 GFLOPS.`

Для базовых операций брались максимальные значения GFLOPS из всех тестов, отличавшихся по количеству нитей (тредов) и размеру сетки:

```
axpy = 1.66 GFLOPS
dot = 3.89 GFLOPS
spmv = 0.914 GFLOPS
```

Итого процент от достижимой производительности:

```
axpy = 0.07%
dot = 0.2%
spmv = 0.04%
```