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

Практическое задание №2

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

выполнила: Мацак Алиса Игоревна, 524 группа

дата подачи: 26.01.2019 г.

Москва, 2019 г.

Содержание

1. Краткое описание задания и программной реализации

1.1. Краткое описание задания

- Расширить реализацию, сделанную в Практическом задании № 1, на параллельные вычисления в рамках параллельной модели с распределенной памятью.

  Расчетная область задачи разделяется на части — подобласти, которые распределяются между параллельными процессами

  Каждый процесс работает со своей частью расчетной области и обменивается с соседними процессами только информацией по интерфейсным ячейкам, граничащим с ячейками других подобластей.

  Для обмена сообщениями используется MPI.

- Программа должна использовать MPI для распараллеливания с распределенной памятью, OpenMP или posix threads для многопоточного распараллеливания.
- Для каждой из базовых операций и для всего алгоритма выполнить сравнение MPI и OpenMP–распараллеливания на многоядерном процессоре. Убедиться, что ускорения сопоставимы.
- Исследовать масштабирование и параллельную эффективность на кластере.

1.2. Краткое описание программной реализации

Программа состоит из следующих файлов:

- csr_matrix.hpp — содержит реализацию матрицы в формате CSR.
  - `void MPI_update(std::vector<double> & x, std::vector <std::vector <int> > recieveRows, std::vector <std::vector <int> > sendRows, std::vector<int> globToLoc)` — обновляет гало–значения вектора x.

  Класс MPI_CompressedSparseRowMatrix содержит методы:

  - `int rowsHaloSize()` — возвращает размер вектора Rows + размер вектора Halo (обозначения взяты из задания);
  - `int rowsSize()` — возвращает размер вектора Rows (обозначение взято из задания);
  - `int haloSize()` — возвращает размер вектора Halo (обозначение взято из задания);
  - `int globColsSize()` — возвращает размер вектора JA (обозначение взято из задания);
  - `int getProcessesNumnber()` — возвращает число работающих MPI процессов;
  - `void copyPart(std::vector<int> partCopy)` — копирует вектор partCopy в вектор Part матрицы (обозначение взято из задания);
  - `std::vector<int> getPart()` — возвращает вектор Part матрицы (обозначение взято из задания);

- `void fillPart(int gridX, int gridY, int gridZ, int partsX, int partsY, int partsZ)` — заполняет вектор `Part` матрицы (обозначение взято из задания);
- `void printPart()` — выводит вектор `Part` матрицы (обозначение взято из задания);
- `void printHalo()` — выводит вектор `Halo` матрицы (обозначение взято из задания);
- `std::vector <std::vector <int> > getRecieveRows()` — возвращает вектор, каждый элемент которого содержит вектор глобальных номеров строк, которые нужны текущему процессу от других процессов; при этом номер элемента соответствует номеру процесса, который владеет нужными строками;
- `void printRecieveRows()` — выводит номера всех процессов и глобальные номера строк, которые нужны от этих процессов текущему процессу;
- `std::vector <std::vector <int> > getSendRows()` — возвращает вектор, каждый элемент которого содержит вектор глобальных номеров строк, которые нужны другим процессам от текущего процесса; при этом номер элемента соответствует номеру процесса, который запрашивает нужные строки;
- `void printSendRows()` — выводит номера всех процессов и глобальные номера строк, которые нужны этим процессам от текущего процесса;
- `void printDataStartRow()` — выводит вектор IA (обозначение взято из задания);
- `void printGlobalColumnNumbersRow()` — выводит вектор JA в глобальной нумерации (обозначение взято из задания);
- `void printLocalColumnNumbersRow()` — выводит вектор JA в локальной нумерации (обозначение взято из задания);
- `void printElementsRow()`— выводит вектор A (обозначение взято из задания);
- `void print()` — печатает матрицу целиком в глобальной нумерации;
- `std::vector<int> getGlobToLoc()` — выводит вектор `globToLoc` (обозначение взято из задания);
- `void pushBackRows(int globRow)` — добавляет глобальный номер строки в конец вектора `Rows` (обозначение взято из задания);
- `int getGlobRowNumber(int locRow)` — возвращает глобальный номер строки по ее локальному номеру;
- `void fillCommunicationRows()` — заполняет два вектора:
  - вектор, каждый элемент которого содержит вектор глобальных номеров строк, которые нужны текущему процессу от других процессов; при этом номер элемента соответствует номеру процесса, который владеет нужными строками;
  - вектор, каждый элемент которого содержит вектор глобальных номеров строк, которые нужны другим процессам от текущего процесса; при этом номер элемента соответствует номеру процесса, который запрашивает нужные строки;
- `void fillHaloAndGlobToLoc()` — заполняет вектор `Halo` и вектор `GlobToLoc`; вызывает `fillCommunicationRows()`;

- `void globToLocColumnNumber()` — переводит глобальную нумерацию столбцов матрицы в локальную;
- `void changeValue(int locRow, int globCol, double element)` — изменяет ненулевое значение матрицы по адресу (`locRow, globCol`) на `element`;
- `double pushBack(int locRow, int globCol, double element)` — кладет `element` и сопутствующую информацию в конец векторов IA, JA, A при построчном заполнении матрицы, `element` в результате становится по адресу (`locRow, globCol`);
- `double pushBackSinRowColumn (int locRow, int globRow, int globCol)` — кладет `element = sin(globRow, globCol)` и сопутствующую информацию в конец векторов IA, JA, A при построчном заполнении матрицы, element в результате становится по адресу (`locRow, globCol`);
- `double getElementInGlobalColumns(int locRow, int globCol)` — выдает значение элемента матрицы по адресу (`locRow, globCol`), если оно существует;
- `double getElementInLocalColumns (int locRow, int locCol)` — выдает значение элемента матрицы по адресу (`locRow, locCol`), если оно существует;
- `void addNumberOfElements()` — кладет количество ненулевых элементов в конец вектора IA;
- `double getNumberOfElements()` — возвращает количество ненулевых элементов в матрице;
- `std::vector<double> & MPI_matrixVectorProduct(std::vector<double> & x, std::vector<double> & y)` — умножение матрицы на вектор x, результат записывается в вектор y.

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

  Файл содержит функции:

  - `double MPI_dotProduct(std::vector<double> & x, std::vector<double> & y)` — скалярное произведение вектора x на вектор y;
  - `std::vector<double> & MPI_linearCombination(std::vector<double> & x, std::vector<double> & y, double a, double b)` — линейная комбинация векторов: `x = a*x + b*y`.
- `test_basic_operations.hpp` — содержит единственную функцию — `void MPI_testBasicOperationsTime(int nx, int ny, int nz, int px, int py, int pz, int procN),` — которая считает время выполнения базовых операций.
- `main_mpi.cpp` — содержит `int MPI_solverBiCGSTAB (std::vector<double> & solutionVector, int sizeRows, int sizeHalo, MPI_CompressedSparseRowMatrix * matrixA, std::vector<double> & vectorBB, double criterionTol, int iterationsMax, int procN)` — MPI версию 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 генератору на входе.

1.2.1. Сборка и запуск

→ Для создания и запуска программ написанных с использованием `MPI` стандарта на системе `Polus` необходимо загрузить модуль `SpectrumMPI`:

```
> module load SpectrumMPI
```

→ Далее программа компилируется командой:

```
> mpicxx main_mpi.cpp -o main
```

→ Программа запускается командой:

```
> ./main
```

Возможные параметры командной строки:

- `nx=`*<целое число>* (по умолчанию: `nx=1`) — размерность регулярной решетки для генерации матрицы по оси `x`;
- `ny=`*<целое число>* (по умолчанию: `ny=1`) — размерность регулярной решетки для генерации матрицы по оси `y`;
- `nz=`*<целое число>* (по умолчанию: `nz=1`) — размерность регулярной решетки для генерации матрицы по оси `z`;
- `tol=`*<число с плавающей точкой>* (по умолчанию: `tol=1e-06`) — критерий сходимости, отношение нормы невязки к норме правой части СЛАУ;
- `maxit=`*<целое число>* (по умолчанию: `maxit=100`) — максимальное число итераций солвера `BiCGSTAB`;
- `qt` — включает тестирование времени выполнения базовых операций;
- `px=`*<целое число>* (по умолчанию: `px=1`) — число частей по оси `x` для декомпозиции регулярной решетки;
- `py=`*<целое число>* (по умолчанию: `py=1`) — число частей по оси `y` для декомпозиции регулярной решетки;
- `pz=`*<целое число>* (по умолчанию: `pz=1`) — число частей по оси `z` для декомпозиции регулярной решетки.

→ Пример запуска программы на 8 MPI процессах с выводом результатов в файл `test.out` на системе `Polus`:

```
> mpisubmit.pl -p 8 --stdout test.out main -- nx=100 ny=100 nz=100
px=2 py=2 pz=2
```

2. Исследование производительности

2.1. Характеристики вычислительной системы

IBM Polus — параллельная вычислительная система, состоящая из 5 вычислительных узлов (на первый вычислительный узел возложены функции frontend узла).

Основные характеристики каждого узла:

- 2 десятиядерных процессора IBM POWER8 (каждое ядро имеет 8 потоков), всего 160 потоков.
- Общая оперативная память 256 Гбайт (в узле 5 оперативная память 1024 Гбайт) с ECC контролем.
- 2 x 1 ТБ 2.5” 7K RPM SATA HDD.
- 2 x NVIDIA Tesla P100 GPU, NVLink.
- 1 порт 100 ГБ/сек.
- Производительность кластера (Tflop/s): 55.84 (пиковая), 40.39 (Linpack).

Программное обеспечение:

- операционная система Linux Red Hat 7.5;
- компиляторы C/C++, Fortran;
- поддержка OpenMP;
- программные средства параллельных вычислений стандарта MPI: библиотека IBM Spectrum MPI, Open MPI;
- планировщик IBM Spectrum LSF;
- CUDA 9.1;
- математическая библиотека IBM ESSL/PESSL.

2.2. Результаты измерений производительности

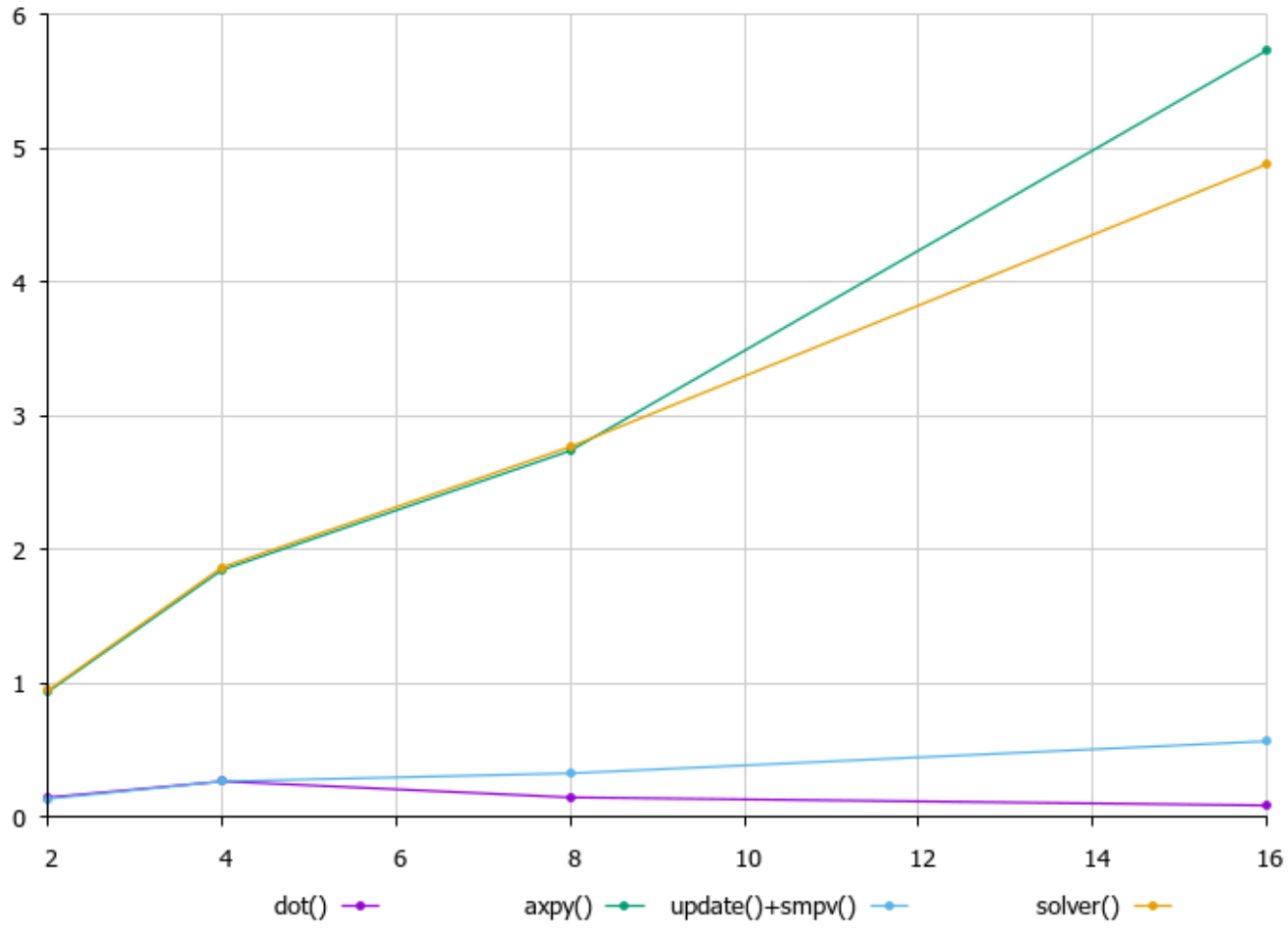
2.2.1. Сравнение MPI и OpenMP на одном многоядерном процессоре

$N = 10^8$					
operation	nt/np	timeOMP	accelerationOMP	timeMPI	accelerationMPI
dot()	2	4.55s	0.007X	56.3s	0.0005X
	4	3.94s	0.008X	27.0s	0.001X
	8	2.24s	0.001X	17.8s	0.002X
	16	3.59s	0.008X	14.3s	0.002X
axpy()	2	4.55s	1.97X	70.5s	0.13X
	4	3.03s	2.95X	33.5s	0.27X
	8	2.51s	3.56	18.4s	0.49X
	16	3.74s	2.39X	8.85s	1.01X
spmv() / update()+ spmv()	2	54.9s	2.11X	942.0s	0.12X
	4	29.0s	4.0X	474.0s	0.25X
	8	18.4s	6.0X	265.0s	0.44X
	16	15.3s	7.58X	140.0s	0.83X
solver()	2	867.71s	1.07X	885.32s	1.05X
	4	849.43s	1.09X	423.57s	2.19X
	8	836.3s	1.11X	224.38s	4.13X
	16	794.92s	1.17X	126.29s	7.34X

2.2.2. Параллельное ускорение

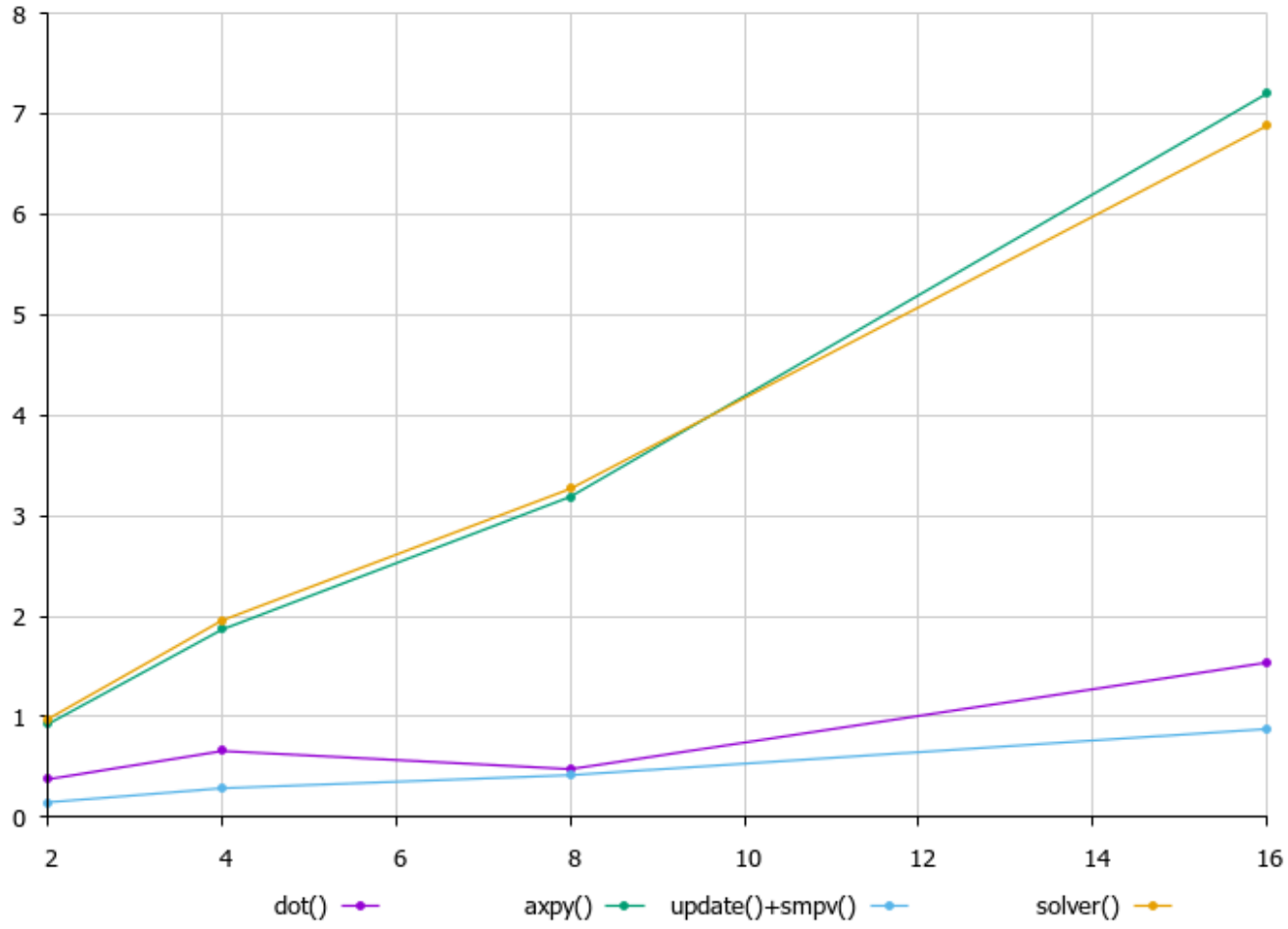
$N = 10^5$			
operation	nt	timeMPI	accelerationMPI
dot()	2	0.053s	0.15X
	4	0.03s	0.27X
	8	0.052s	0.15X
	16	0.09s	0.09X
axpy()	2	0.067s	0.94X
	4	0.034s	1.85X
	8	0.023s	2.74X
	16	0.011s	5.73X
update() +spmv()	2	0.9s	0.14X
	4	0.49s	0.27X
	8	0.39s	0.33X
	16	0.23s	0.57X
solver()	2	0.87s	0.95X
	4	0.44s	1.87X
	8	0.3s	2.77X
	16	0.17s	4.88X

График зависимости ускорения от числа процессов при $N = 10^5$



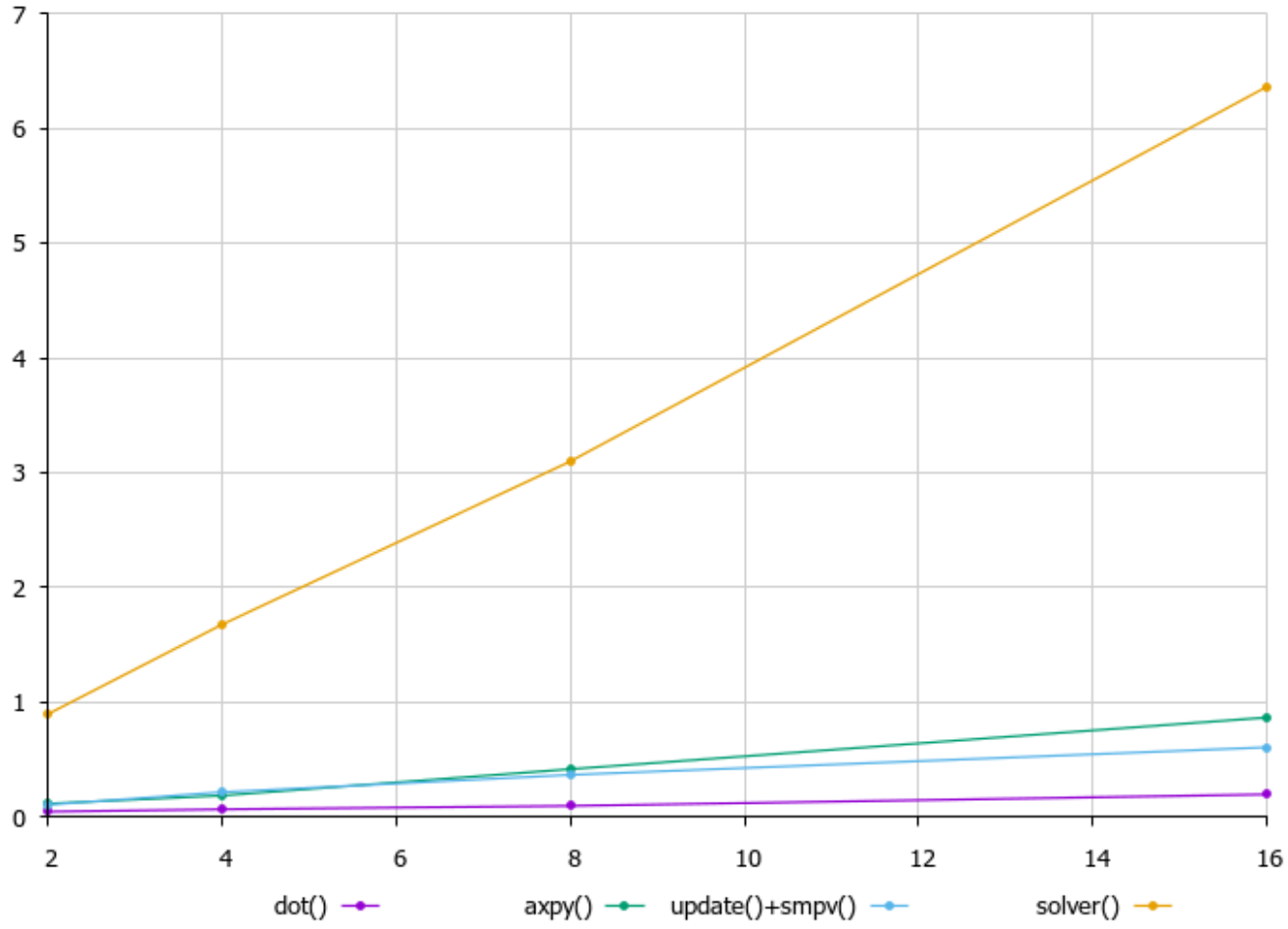
$N = 10^6$			
operation	nt	timeMPI	accelerationMPI
dot()	2	0.532s	0.38X
	4	0.302s	0.66X
	8	0.418s	0.48X
	16	0.13s	1.54X
axpy()	2	0.675s	0.93X
	4	0.334s	1.87X
	8	0.196s	3.19X
	16	0.087s	7.2X
update() +spmv()	2	9.19s	0.15X
	4	4.69s	0.29X
	8	3.21s	0.42X
	16	1.54s	0.88X
solver()	2	8.46s	0.98X
	4	4.2s	1.96X
	8	2.52s	3.27X
	16	1.2s	6.88X

График зависимости ускорения от числа процессов при $N = 10^6$



$N = 10^7$			
operation	nt	timeMPI	accelerationMPI
dot()	2	5.55s	0.05X
	4	4.3s	0.07X
	8	3.16s	0.1X
	16	1.42s	0.2X
axpy()	2	7.0s	0.12X
	4	4.38s	0.19X
	8	1.96s	0.42X
	16	0.952s	0.87X
update() +spmv()	2	101.0s	0.11X
	4	49.9s	0.22X
	8	29.8s	0.37X
	16	17.9s	0.61X
solver()	2	93.34s	0.9X
	4	49.87s	1.68X
	8	26.96s	3.1X
	16	13.15s	6.36X

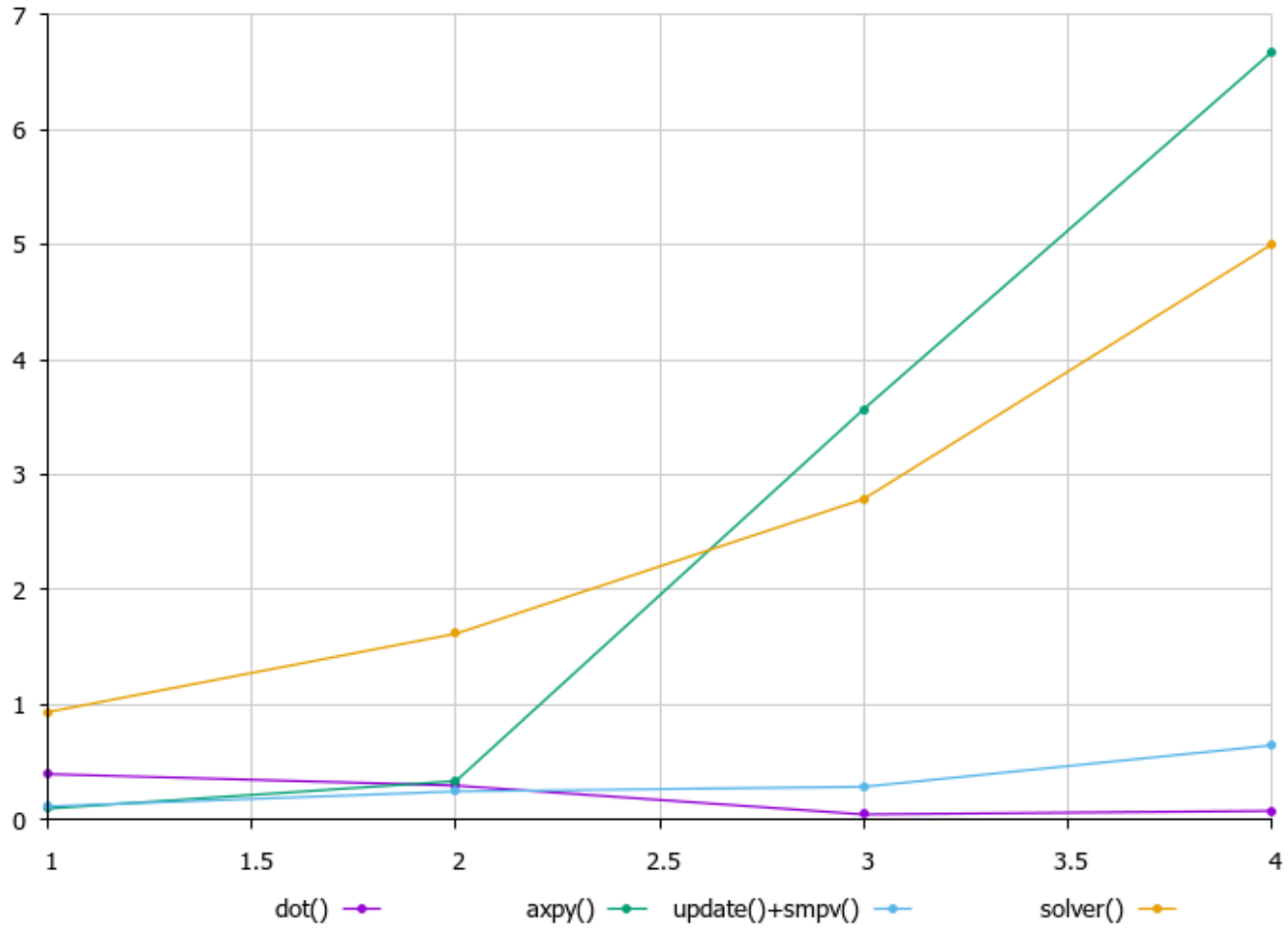
График зависимости ускорения от числа процессов при $N = 10^7$



2.2.3. Масштабирование

$\frac{N}{P} = 10^4$				
operation	N	Nt	timeMPI	accelerationMPI
dot()	$N = 2 * 10^4$	2	0.011s	0.4X
	$N = 4 * 10^4$	4	0.018s	0.3X
	$N = 8 * 10^4$	8	0.096s	0.05X
	$N = 16 * 10^4$	16	0.077s	0.08X
axpy()	$N = 2 * 10^4$	2	0.014s	0.1X
	$N = 4 * 10^4$	4	0.014s	0.34X
	$N = 8 * 10^4$	8	0.014s	3.57X
	$N = 16 * 10^4$	16	0.015s	6.67X
update() +spmv()	$N = 2 * 10^4$	2	0.18s	0.12X
	$N = 4 * 10^4$	4	0.2s	0.25X
	$N = 8 * 10^4$	8	0.37s	0.29X
	$N = 16 * 10^4$	16	0.33s	0.65X
solver()	$N = 2 * 10^4$	2	0.18s	0.94X
	$N = 4 * 10^4$	4	0.21s	1.62X
	$N = 8 * 10^4$	8	0.24s	2.79X
	$N = 16 * 10^4$	16	0.27s	5.0X

График зависимости ускорения от различных N и P при фиксированном $\frac{N}{P} = 10^4$

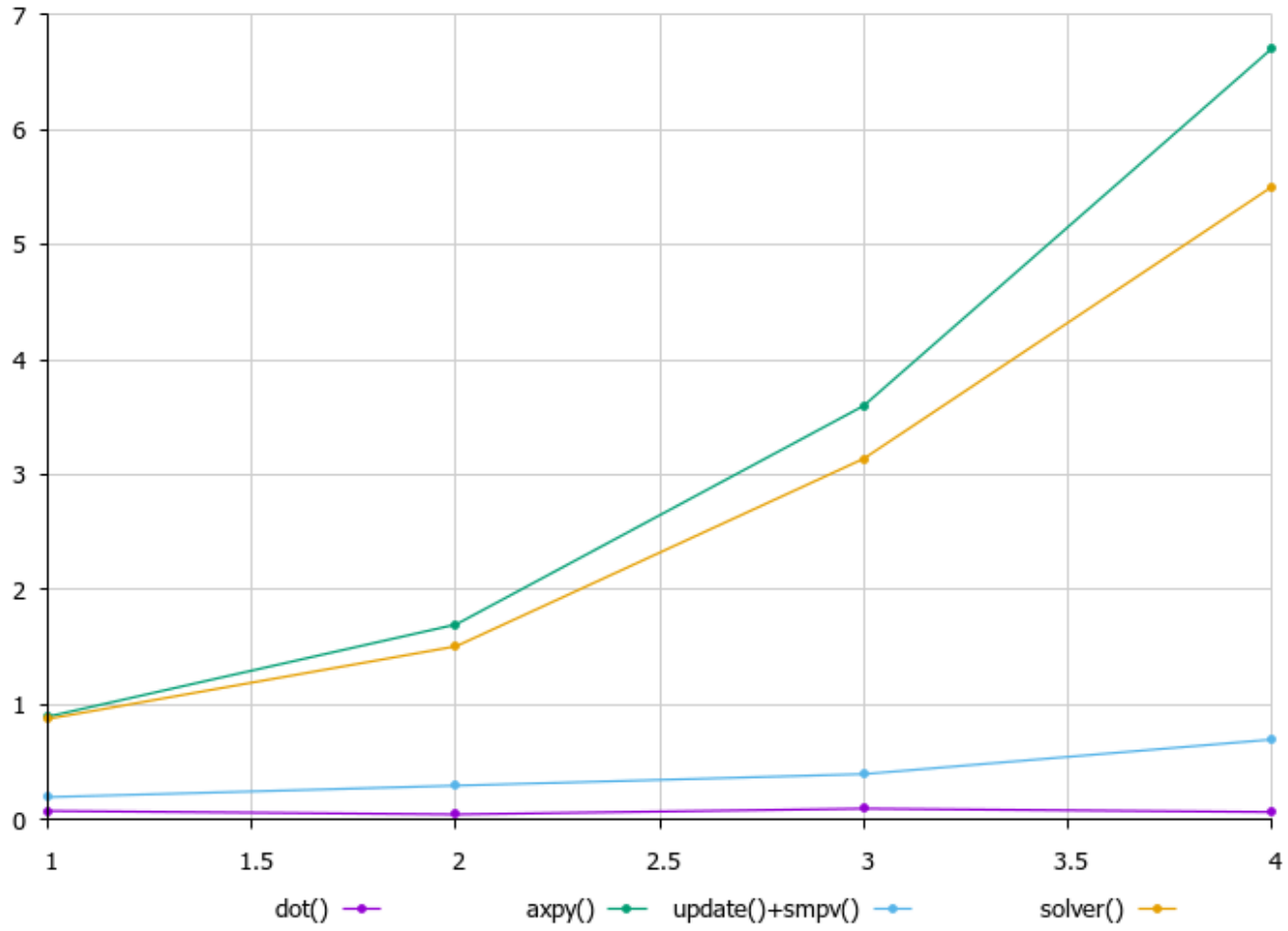


На графике:

- ★ 1: $nt = 2, N = 2 * 10^4$;
- ★ 2: $nt = 4, N = 4 * 10^4$;
- ★ 3: $nt = 8, N = 8 * 10^4$;
- ★ 4: $nt = 16, N = 16 * 10^4$.

$\frac{N}{P} = 10^5$				
operation	N	nt	timeMPI	accelerationMPI
dot()	$N = 2 * 10^5$	2	0.11s	0.08X
	$N = 4 * 10^5$	4	0.16s	0.05X
	$N = 8 * 10^5$	8	0.19s	0.1X
	$N = 16 * 10^5$	16	0.3s	0.07X
update() +spmv()	$N = 2 * 10^5$	2	0.14s	0.9X
	$N = 4 * 10^5$	4	0.15s	1.7X
	$N = 8 * 10^5$	8	0.14s	3.6X
	$N = 16 * 10^5$	16	0.15s	6.7X
spmv()	$N = 2 * 10^5$	2	1.77s	0.2X
	$N = 4 * 10^5$	4	2.02s	0.3X
	$N = 8 * 10^5$	8	2.56s	0.4X
	$N = 16 * 10^5$	16	2.9s	0.7X
solver()	$N = 2 * 10^5$	2	1.92s	0.88X
	$N = 4 * 10^5$	4	2.28s	1.51X
	$N = 8 * 10^5$	8	2.14s	3.14X
	$N = 16 * 10^5$	16	2.43s	5.5X

График зависимости ускорения от различных N и P при фиксированном $\frac{N}{P} = 10^5$

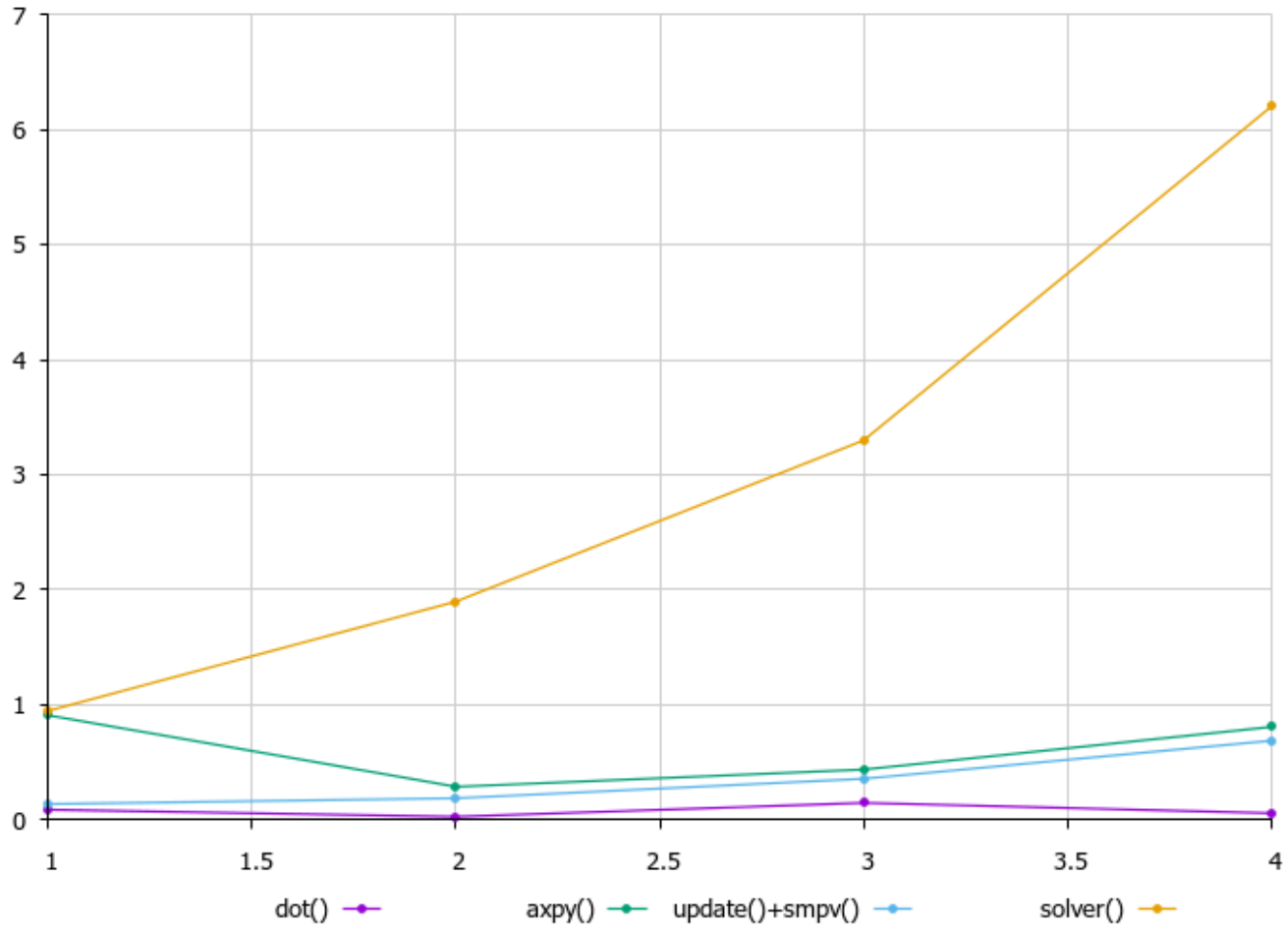


На графике:

- ★ 1: $nt = 2, N = 2 * 10^5$;
- ★ 2: $nt = 4, N = 4 * 10^5$;
- ★ 3: $nt = 8, N = 8 * 10^5$;
- ★ 4: $nt = 16, N = 16 * 10^5$.

$\frac{N}{P} = 10^6$				
Operation	N	nt	timeMPI	accelerationMPI
dot()	$N = 2 * 10^6$	2	1.14s	0.09X
	$N = 4 * 10^6$	4	5.88s	0.03X
	$N = 8 * 10^6$	8	2.01s	0.15X
	$N = 16 * 10^6$	16	3.51s	0.06X
axpy()	$N = 2 * 10^6$	2	1.38s	0.91X
	$N = 4 * 10^6$	4	1.44s	0.29X
	$N = 8 * 10^6$	8	1.5s	0.44X
	$N = 16 * 10^6$	16	1.68s	0.81X
update() +spmv()	$N = 2 * 10^6$	2	19.0s	0.14X
	$N = 4 * 10^6$	4	22.1s	0.19X
	$N = 8 * 10^6$	8	23.6s	0.36X
	$N = 16 * 10^6$	16	25.3s	0.69X
solver()	$N = 2 * 10^6$	2	17.39s	0.95X
	$N = 4 * 10^6$	4	17.67s	1.9X
	$N = 8 * 10^6$	8	20.18s	3.3X
	$N = 16 * 10^6$	16	21.31s	6.2X

График зависимости ускорения от различных N и P при фиксированном $\frac{N}{P} = 10^6$



На графике:

- ★ 1: $nt = 2, N = 2 * 10^6$;
- ★ 2: $nt = 4, N = 4 * 10^6$;
- ★ 3: $nt = 8, N = 8 * 10^6$;
- ★ 4: $nt = 16, N = 16 * 10^6$.