Московский государственный университет имени М.В.Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра системного программирования

Егоров Илья Георгиевич
группа 527

Параллельные высокопроизводительные вычисления

Многопоточная реализация операций с сеточными данными на неструктурированной смешанной сетке, решение системы линейных алгебраических уравнений

ОТЧЁТ

Москва, 2024, 7 Ноября

Содержание

1 Описание задания и программной реализации

1.1 Постановка задачи

Работа программы состоит из 4 этапов:

1. **Generate** — генерация графа/портрета по тестовой сетке;
2. **Fill** — заполнение матрицы по заданному портрету;
3. **Solve** — решение СЛАУ с полученной матрицей;
4. **Report** — проверка корректности программы и выдача измерений.

1.1.1 Генерация портрета на основе тестовой сетки

Будем иметь дело с двухмерной неструктурированной смешанной сеткой, состоящей из треугольников и четырехугольников. Решетка состоит из $N = N_x * N_y$ клеточек. У каждой клетки есть позиция (x, y), где i — номер строки в решетке, j — номер столбца. Нумерация клеток в решетке построчно слева направо, сверху вниз. Номер клеточки в единой общей нумерации: $I = i * N_x + j$. Далее часть квадратных клеток делятся на треугольники следующим образом: K_1 и K_2 — количество идущих подряд треугольников и четырехугольников соотвественно. На данном этапе нужно сгенерировать "топологию"связей ячеек: построить графа, и по нему сгенерировать портрет матрицы смежности, дополнив этот портрет главной диагональю, где вершины графа — элементы сетки (вариант Б2).

Входные данные:

- N_x, N_y — число клеток в решетке по вертикали и горизонтали;
- K_1, K_2 — параметры для количества треугольных и четырехугольных элементов.

Выходные данные:

- N — размер матрицы (число вершин в графе);
- IA, JA — портрет разреженной матрицы смежности графа, (в формате CSR).

1.1.2 Построение СЛАУ по заданному портрету матрицы

Входные данные:

- N — размер матрицы (число вершин в графе);
- IA, JA — портрет разреженной матрицы смежности графа, (в формате CSR).

Выходные данные:

- A — массив ненулевых коэффициентов матрицы (размера $IA[N]$);
- b — вектор правой части (размера N).

Правила заполнения:

- $a_{ij} = \cos(i * j + i + j), i \neq j, j \in Col(i)$
- $a_{ii} = 1.234 \sum_{j,j \neq i} |a_{ij}|$
- $b_i = \sin(i)$

1.1.3 Решение СЛАУ итерационным методом

Входные данные:

- N — размер матрицы (число вершин в графе);
- IA, JA — портрет разреженной матрицы смежности графа, (в формате CSR).
- A — массив ненулевых коэффициентов матрицы (размера $IA[N]$);
- b — вектор правой части (размера N).
- eps — критерий остановки (ϵ), которым определяется точность решения;
- $maxit$ — максимальное число итераций.

Выходные данные:

- x — вектор решения (размера N);
- n — количество выполненных итераций;

- r — L2 норма невязки (невязка – это вектор $r = Ax - b$).

Будем использовать такой алгоритм предобусловленного метода CG.

Для решателя понадобится несколько вычислительных функций для базовых операций:

- Матрично-векторное произведение с разреженной матрицей (sparse matrix-vector);
- Скалярное произведение;
- Поэлементное сложение двух векторов с умножением одного из них на скаляр.

1.1.4 Проверка корректности программы и выдача измерений

На этом этапе нужно проверить, что невязка системы удовлетворяет заданной точности, выдать значение фактической невязки, и распечатать табличку таймирования, в которой указано, сколько времени в секундах затрачено на:

1. Этап генерации;
2. Этап заполнения;
3. Этап решения СЛАУ;
4. Каждую из трех кернел-функций, базовых алгебраических операций решателя СЛАУ.

1.2 Программная реализация

Программа реализована на языке C++ (c++11/c++17).

На листинге 1 представлен интерфейс программы.

Для хранения матриц векторов использовался стандартный *std::vector*, для возврата результата часто использовался *std::pair* и *std::tuple* (удобно для c++17 и выше). Основные релизованные функции:

- gen — функция генерации портрета матрицы;
- fill — функция построения СЛАУ по портрету матрицы;

- solve — функция-решатель СЛАУ;
- scalar — функция, вычисляющая скалярное произведение;
- sum_vvc — функция, вычисляющая поэлементное сложение двух векторов с умножением одного из них на скаляр;
- mul_mv — функция, вычисляющая матрично-векторное произведение с разреженной матрицей (sparse matrix-vector).

Более подробное описание функций можно найти в приложении 3.

Листинг 1 Интерфейс реализованной программы

```
Usage: ../solver Nx Ny K1 K2 Maxit Eps Tn Ll
Where:
Nx is positive int that represents grid hieght
Ny is positive int that represents grid width
K1 is positive int that represents square cells sequence length
K2 is positive int that represents triangular cells sequence length
Maxit is positive int that represents maximum iteration number
Eps is positive float that represents accuracy
Tn is tread numberLl is log level:
        <=0 - no logs
        >=1 - show time
        >=2 - show info
        >=3 - show arrays
```

2 Исследование производительности

2.1 Характеристики вычислительной системы

Имя: ПВС «IBM Polus»

Пиковая производительность: 55.84 TFlop/s

Производительность (Linpack): 40.39 TFlop/s

Вычислительных узлов: 5

На каждом узле:

Процессоры IBM Power 8: 2

NVIDIA Tesla P100: 2

Число процессорных ядер: 20

Число потоков на ядро: 8

Оперативная память: 256 Гбайт (1024 Гбайт узел 5)

Коммуникационная сеть: Infiniband / 100 Gb

Система хранения данных: GPFS

Операционная система: Linux Red Hat 7.5

2.2 Результаты измерений производительности

Все измерения проводились со следующими параметрами:

- $K_1 = 101$
- $K_2 = 57$
- $maxit = 1000$
- $eps = 0.0001$

Параметры K_1 и K_2 были выбраны взаимнопростыми друг с другом и с размерностями сетки для большей хаотичности, выбор $maxit$ особо не на что не влияет, так как при заданной точности eps алгоритм сходится достаточно быстро (при общем размере сетки порядка $9 * 10^9$ вершин алгоритм сходится всего за 25 итераций).

Используемые опции компиляции: *-std=c++11 -O1 -fopenmp*

2.2.1 Последовательная производительность

Линейный линейный рост потребления памяти и времени работы в зависимости от размера системы видны на на таб. 1, а также рис. 1 и рис. 2 соответственно.

Таблица 1: Зависимость потребления памяти и времени работы от размера системы

Data size	Memory (MB)	Time (seconds)
1000000	253	0.574201
2000000	503	1.69145
3000000	751	1.82295
4000000	1000	2.42671
5000000	1248	3.73934

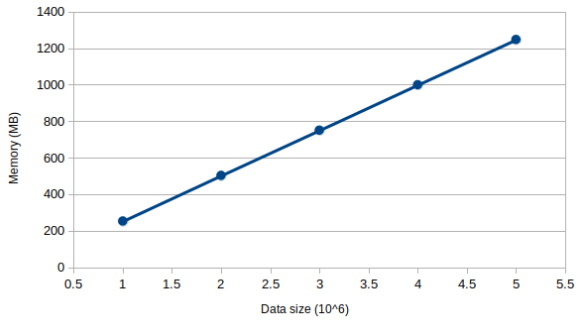

Рис. 1: Зависимость потребления памяти от размера системы

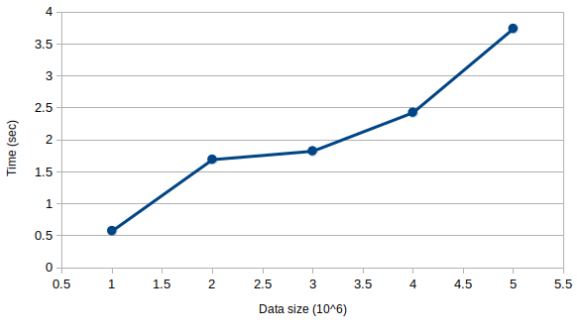

Рис. 2: Зависимость времени работы от размера системы

На таб. 2 и рис. 3 показана зависимость достигаемой производительности от размера системы для всего алгоритма решателя.

Таблица 2: Зависимость достигаемой производительности от размера системы

N	Time (sec)	FLOP	FLOPS	GFLOPS
10000	0.00265486	3852420	1451082166	1.451082166
100000	0.0288859	42150480	1459206049	1.459206049
1000000	0.350449	492786840	1406158500	1.4061585
10000000	4.03278	5633019840	1396808117	1.396808117
100000000	65.0932	88033963500	1352429493	1.352429493

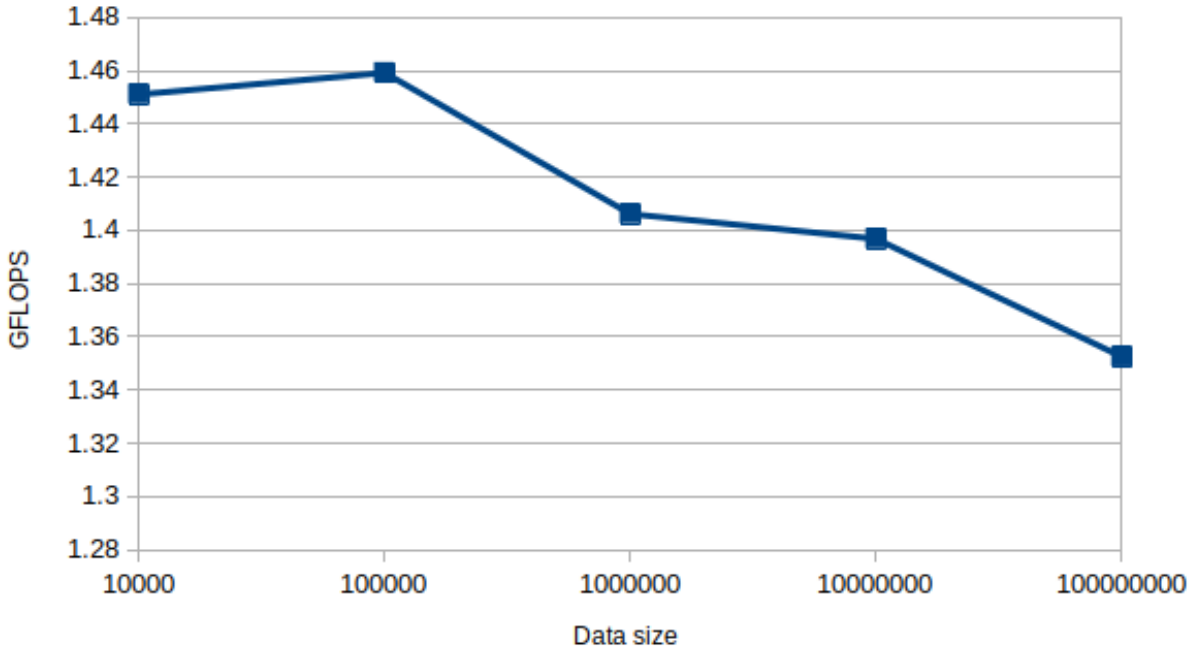

Рис. 3: Зависимость достигаемой производительности от размера системы

2.2.2 Параллельное ускорение

На таб. 3 и рис. 4 и 5 хорошо видно наличие параллельного ускорения для разных этапов программы и основных функций соответственно. Замеры производились с параметрами $N_x = 8000$ и $N_y = 8000$ и привязкой к узлам *polus-c3-ib* и *polus-c4-ib* и ядрам.

Таблица 3: Зависимость времени работы этапов программы и основных функций от числа параллельных процессов

Thread number	Time (sec)					
	Generation	Filling	Solving	Scalar	Addiction	Multiplication
1	8.31451	41.2727	71.4609	5.53869	7.04687	21.4265
2	4.69452	22.5375	25.0211	2.1435	2.18539	6.1266
4	2.4795	11.524	12.5424	1.06713	1.10095	3.05322
8	1.43452	5.60599	6.09411	0.501688	0.554672	1.5036
16	1.17973	2.85349	3.57233	0.3576	0.355618	1.24018
32	0.953992	1.63569	2.49932	0.235401	0.274155	1.04491

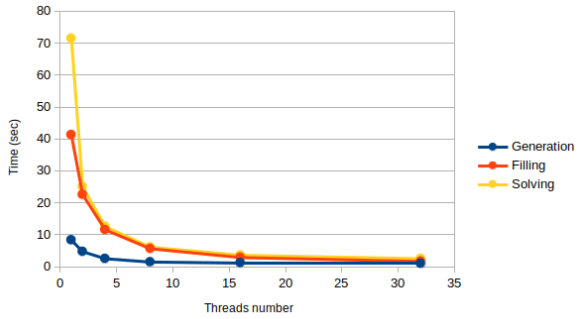

Рис. 4: Зависимость времени работы этапов программы от числа параллельных процессов

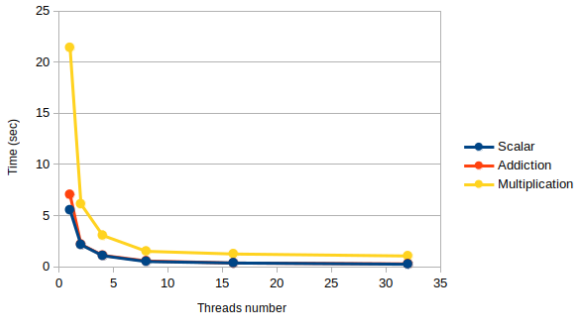

Рис. 5: Зависимость времени работы основных функций от числа параллельных процессов

3 Анализ полученных результатов

Рассчитаем TBP (Theoretical Bounded Performance) — теоретически достижимую производительность — и получаемый процент от неё для каждой из трех базовых операций. TBP рассчитывается по следующей формуле: $TBP = min(TPP, BW * AI)$, где TPP (Theoretical Peak Performance) – теоретическая пиковая производительность, BW (Bandwidth) — пропускная способность памяти, AI (Arithmetic Intensity) — арифметическая интенсивность. Для Polus $TPP = 55.84TFlop/s$, $BW = 153.6GB/s$, а AI будет разной для разных функций. Для скалярного произведения и сложения AI будут фиксированными:

- $AI_{scal} = \frac{2*n}{2*sizeof(float)*n} = \frac{1}{4}$
- $AI_{add} = \frac{2*n}{3*sizeof(float)*n} = \frac{1}{6}$

AI для умножения матриц будет иметь более сложную зависимость. Путь $ia.size() = n$, а $ja.size() = m$. Тогда $ia.size() = res.size() = v.size() = n$ и $ja.size() = a.size() = m$, где все перечисленные вектора соответствую аргументам функции mul_mv (подробнее в листинге 7). Отсюда имеем следующую формулу:

- $AI_{mul} = \frac{2m}{sizeof(float)*(2m+3n)} = \frac{m}{4m+6n}$

n и m будут зависеть от количества вершин в графе, т.е. от каждого из параметров N_x, N_y, K_1, K_2. Для значений параметров $N_x = 8000$, $N_y = 8000$, $K_1 = 101$, $K_2 = 57$ получается следующий результат:

$n = 87088592$, $m = 389233773$, $AI_{mul} = \frac{389233773}{2079466644} \approx 0.187$

Получаем следующий результат:

- $TBP_{scal} = 38.4GFlops$, $\frac{TBP_{scal}}{TPP} \approx 0.067\%$
- $TBP_{add} \approx 25.6GFlops$, $\frac{TBP_{add}}{TPP} \approx 0.045\%$
- $TBP_{mul} \approx 28,75GFlops$, $\frac{TBP_{mul}}{TPP} \approx 0.05\%$

Приложение

В листингах 2, 3, 4, 5, 6 и 7 приведены прототипы и описания основных реализованных функций.

Листинг 2 Функция генерации портрета матрицы

```
/* Generate CSR portrait by grid params
 * # Arguments:
 * * nx - grid hieght
 * * ny - grid width
 * * k1 - square cells sequence length
 * * k2 - triangular cells sequence length
 * * ll - log level
 * # Return values:
 * * ia - row CSR array
 * * ja - col CSR array
 * * t - time
 */
std::tuple<std::vector<size_t>, std::vector<size_t>, double>
gen(
    size_t nx,
    size_t ny,
    size_t k1,
    size_t k2,
    LogLevel ll
);
```

Листинг 3 Функция построения СЛАУ по портрету матрицы

```
/* Fill val CSR array by given row/col arrays and right side array
 * # Arguments:
 * * ia - row CSR array
 * * ja - col CSR array
 * # Return values:
 * * a - val CSR array
 * * b - right side array
 * * t - time
 */
std::tuple<std::vector<float>, std::vector<float>, double>
fill(
    std::vector<size_t>& ia,
    std::vector<size_t>& ja
);
```

Листинг 4 Функция-решатель СЛАУ

```
/* Solve Ax=b system
 * # Arguments:
 * * n - size
 * * ia - A row CSR array
 * * ja - A col CSR array
 * * a - A val CSR array
 * * b - right side array
 * * eps - accuracy
 * * maxit = maximum iteration number
 * * ll - log level
 * # Return values:
 * * x - solution
 * * k - iteration number
 * * r - residual
 * * t - operation time tuple
 */
std::tuple<
  std::vector<float>,
  size_t,
  std::vector<float>,
  std::tuple<double, double, double, double>
>
solve(
    size_t n,
    std::vector<size_t>& ia,
    std::vector<size_t>& ja,
    std::vector<float>& a,
    std::vector<float>& b,
    float eps,
    size_t maxit,
    LogLevel ll
);
```

Листинг 5 Функция, вычисляющая скалярное произведение

```
/* Compute scalar product of two vectors with equal sizes
 * # Arguments:
 * * a - first vector
 * * b - second vector
 * * n - size
 * # Return value:
 * * scalar product value
 * * computing time
 */
std::pair<float, double>
scalar(
    std::vector<float>& a,
    std::vector<float>& b,
    size_t n
);
```

Листинг 6 Функция, вычисляющая поэлементное сложение двух векторов с умножением одного из них на скаляр

```
/* Store sum of two vectors with equal sizes (second with coef)
 * to preallocated vector
 * # Arguments:
 * * res - allocated result vector
 * * a - first vector
 * * b - second vector
 * * c - second vector coefficient
 * * n - size
 * # Return value:
 * * computing time
 */
double sum_vvc(
    std::vector<float>& res,
    std::vector<float>& a,
    std::vector<float>& b,
    float c,
    size_t n
);
```

Листинг 7 Функция, вычисляющая матрично-векторное произведение с разреженной матрицей

```
/* Store multiply square CSR matrix by vector to preallocated vector
 * # Arguments:
 * * res - allocated result vector
 * * ia - row CSR array
 * * ja - col CSR array
 * * a - val CSR array
 * * v - vector
 * * n - size
 * # Return value:
 * * computing time
 */
double mul_mv(
    std::vector<float>& res,
    std::vector<size_t>& ia,
    std::vector<size_t>& ja,
    std::vector<float>& a,
    std::vector<float>& v,
    size_t n
)
```