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 для кластерных систем с распределенной памятью

ОТЧЁТ

Москва, 2024, 16 Декабря

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. **Comm** — построение схемы обменов;
4. **Solve** — решение СЛАУ с полученной матрицей;
5. **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;
- P_x, P_y — параметры декомпозиция по двум осям (Программа запускается через MPI с числом процессов $P = P_x * P_y$).

Выходные данные:

- N — размер матрицы (число вершин в графе);
- N_o — число собственных вершин в локальной подобласти;
- IA, JA — портрет разреженной матрицы смежности графа, (в формате CSR);
- $Part$ — массив с номерами владельцев по каждой вершине в локальном графе;
- $L2G$ — массив с номерами вершин локального графа в глобальном графе, то есть отображение из локальной нумерации в глобальную.

1.1.2 Построение СЛАУ по заданному портрету матрицы

Входные данные:

- N — размер матрицы (число вершин в графе);
- N_o — число собственных вершин в локальной подобласти;
- IA, JA — портрет разреженной матрицы смежности графа, (в формате CSR).
- $L2G$ — массив с номерами вершин локального графа в глобальном графе, то есть отображение из локальной нумерации в глобальную.

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 Построение схемы обмена данными

Входные данные:

- N — размер матрицы (число вершин в графе);
- N_o — число собственных вершин в локальной подобласти;
- IA, JA — портрет разреженной матрицы смежности графа, (в формате CSR).
- $Part$ — массив с номерами владельцев по каждой вершине в локальном графе;

Выходные данные:

- $Comm$ — схема обменов, в данном случае массив специальных структур (подробнее в приложении 3 в листинге 2).

1.1.4 Решение СЛАУ итерационным методом

Входные данные:

- N — размер матрицы (число вершин в графе);
- N_o — число собственных вершин в локальной подобласти;
- IA, JA — портрет разреженной матрицы смежности графа, (в формате CSR).
- A — массив ненулевых коэффициентов матрицы (размера $IA[N]$);
- b — вектор правой части (размера N);
- $Comm$ — схема обменов;
- 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.

1.1.5 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;

Листинг 1 Интерфейс реализованной программы

```
Usage: ../solver Nx Ny K1 K2 Px Py 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
Px is positive int that represents x axis decomposition param
Py is positive int that represents y axis decomposition param
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 arrays
        >=3 - show info
```

- build_comm — функция построения схемы обменов;
- 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.

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 -O3 -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

n	Data size	Memory (MB)	Time (seconds)
1	1000000	255	1.81
2	2000000	638	3.16
3	3000000	918	4.69
4	4000000	1219	6.02
5	5000000	1505	7.68

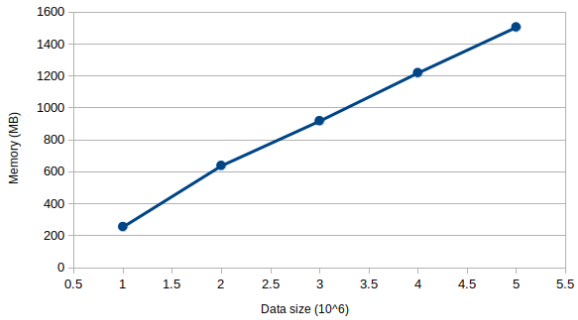

Рис. 1: Зависимость потребления памяти от размера системы

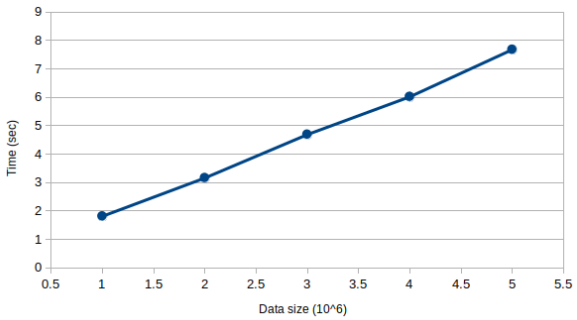

Рис. 2: Зависимость времени работы от размера системы

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	0.00507376	7004400	1380514648	1.380514648
100000	0.0548021	77275880	1410089759	1.410089759
1000000	0.575652	809578380	1406367701	1.406367701
10000000	6.48869	9153657240	1410709595	1.410709595
100000000	120.427	165503851380	1374308514	1.374308514

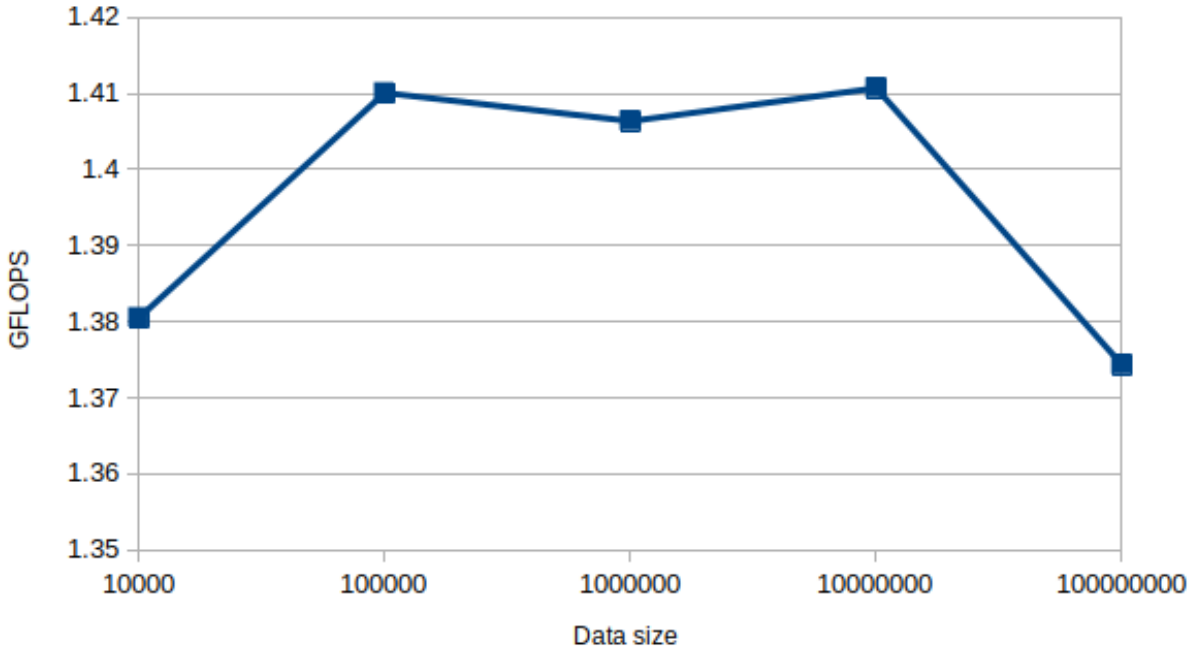

Рис. 3: Зависимость достигаемой производительности от размера системы

2.2.2 Параллельное ускорение

По таб. 3 и рис. 4 и 5 хорошо видно наличие параллельного ускорения для разных этапов программы и основных функций соответственно при увеличении числа процессов (каждый процесс использовал при этом одну нить). На таб. 4 и рис. 6 и 7 приведены аналогичные результаты про росте числа нитей для 2 процессов (указано число нитей каждого процеса). Замеры производились с параметрами $N_x = 8000$ и $N_y = 8000$ на узлах *polus-c3-ib* и *polus-c4-ib* и ядрам.

Исходя из значений в таблицах можно сделать вывод, что данное решение гораздо лучше распараллеливается при параллелизме с распределённой памятью, чем с общей.

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

Process number	Time (sec)					
	Generation	Filling	Solving	Scalar	Addiction	Multiplication
1	14.9495	39.9364	35.3001	6.58842	3.84681	22.994
2	7.97041	20.3477	18.0691	3.3094	1.95556	11.5381
4	4.2798	9.10192	11.8547	1.69546	1.26462	6.02451
8	3.03957	4.58143	7.8764	1.09015	1.13784	3.88704
16	2.53211	1.79083	5.81112	0.838099	0.633418	2.2245

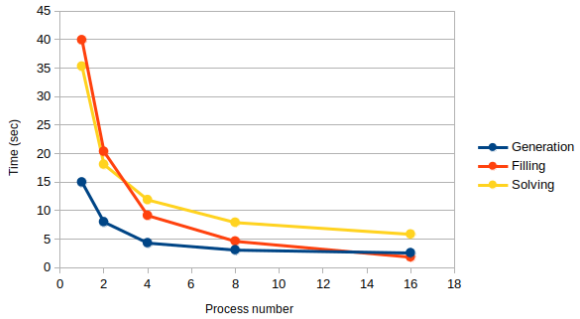

Рис. 4: Зависимость времени работы этапов программы от числа параллельных процессов

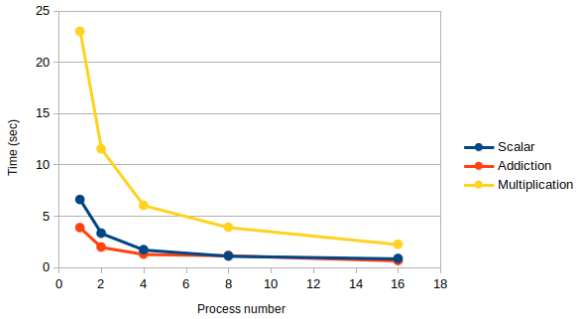

Рис. 5: Зависимость времени работы основных функций от числа параллельных процессов

Таблица 4: Зависимость времени работы этапов программы и основных функций от числа нитей процесса

Thread number	Time (sec)					
	Generation	Filling	Solving	Scalar	Addiction	Multiplication
1	7.73392	18.339	22.1044	3.3067	2.0031	11.5658
2	6.2181	11.0741	10.1256	1.75116	1.26054	6.04317
4	5.24675	5.66123	7.42345	1.02451	1.15399	4.03253
8	4.77103	3.21805	6.6462	0.99346	1.10461	3.93224

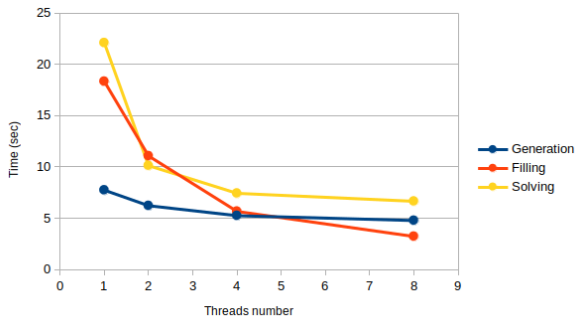

Рис. 6: Зависимость времени работы этапов программы от числа нитей процесса

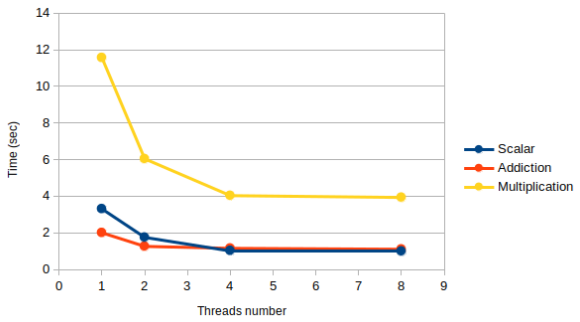

Рис. 7: Зависимость времени работы основных функций от числа нитей процесса

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 9). 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 листинге 2 дано описание структуры для обмена между процессами. В листингах 3, 4, 5 6, 7, 8 и 9 приведены прототипы и описания основных реализованных функций.

Листинг 2 Структура для обмена между процессами

```
// Interprocess communication structure
struct Comm {
    int pr;
    std::vector<size_t> send;
    std::vector<size_t> recv;
};
```

Листинг 3 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
 * * px - x axis decomposition param
 * * py - y axis decomposition param
 * * pr - MPI process rank
 * * ll - log level
 * # Return values:
 * * ia - row CSR array
 * * ja - col CSR array
 * * l2g - local to global vertices numbers array
 * * part - partition array
 * * sections - vertex sections info
 * * sends - interface vertex send info
 * * t - time
 */
std::tuple<
    std::vector<size_t>,
    std::vector<size_t>,
    std::vector<size_t>,
    std::vector<int>,
    Sections,
    Sends,
    double
> gen(
    size_t nx,
    size_t ny,
    size_t k1,
    size_t k2,
    int px,
    int py,
    int pr,
```

Листинг 4 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
 * * l2g - local to global vertices numbers 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,
    std::vector<size_t>& l2g
);
```

Листинг 5 Функция построения схемы обменов

```
/* Build interprocess communication scheme
 * # Arguments
 * * sections - vertex sections info
 * * part - partition array
 * * l2g - local to global vertices numbers array
 * # Return values
 * * comms - interprocess communication array
 * * t - time
 */
std::tuple<
    std::vector<Comm>,
    double
> build_comm(
    Sections& sections,
    std::vector<int>& part,
    std::vector<size_t>& l2g
);
```

Листинг 6 Функция-решатель СЛАУ

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

Листинг 7 Функция, вычисляющая скалярное произведение

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

Листинг 8 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
 * # Return value:
 * * computing time
 */
double sum_vvc(
    std::vector<float>& res,
    std::vector<float>& a,
    std::vector<float>& b,
    float c
);
```

Листинг 9 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
 * * comms - interprocess communication array
 * # 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,
    std::vector<Comm>& comms
);
```