Параллельные вычисления

Параллельная реализация операций с сеточными данными на неструктурированной смешанной сетке

Лазарев Владимир Александрович

528 группа

17.10.2020

Описание задачи и программной реализации

Краткое описание задания

Целью задания является построения портера разреженной матрицы на основе представления неструктурированной сетки, построение СЛАУ на основе полученного портрета и решения СЛАУ.

Краткое описание программной реализации

В ходе разработки программы были реализованы следующие методы:

- *void createNodesOfGraph* – метод генерации портрета матрицы. Присутствует распараллеливание. Аргументами метода являются:
  - *int Nx* – ширина сетки;
  - *int Ny* – высота сетки;
  - *int k1* – количество несоединенных вершин графа;
  - *int k2* – количество соединенных вершин графа;
  - *map<int, vector<int>> resultGraph* – представление графа в виде справочника. Для каждого элемента по его индексу можно получить список его соседей;
  - *vector<int> IA, JA* – портрет матрицы;
  - *int sizeOfResultGraph* – количество вершин.
- *string vectorToString* – метод выводит на печать в консоль структуру типа vector<int>;
- *vector<string> createAdjacencyList* – метод выводит в консоль список смежности для каждой вершины;
- *void printResult* – печать результатов полученного портрета;
- *void makeSLAE* – метод построения СЛАУ на основе ранее созданного портера матрицы. Выполняется расчет ненулевых коэффициентов и вектора правой части. Присутствует распараллеливание. Аргументы:
  - *vector<int> IA, vector<int> JA* – портрет матрицы;
  - *vector<double> A* – ненулевые коэффициенты матрицы;
  - *vector<double> b* – вектор правой части;
  - *int countOfNodes* – количество вершин графа.
- *void printSLAE* – печать ненулевых коэффициентов каждой строки и значения вектора правой части;

- *double scalar* – принимает на вход две структуры типа vector<double> и рассчитывает их скалярное произведение (скалярное произведение векторов). Присутствует распараллеливание. Аргументы:
    - *vector<double> x1, x2* – вектора для перемножения;
    - *double allTime* – общее время, потраченное на текущий метод;
    - *double countOfCalls* – количество раз, сколько вызывался текущий метод.
- *double normalizeVector* – принимает на вход один вектор и возвращает его норму. Присутствует распараллеливание.
- *vector<double> spMV* – реализация матрично-векторного произведения. Присутствует распараллеливание. Список аргументов:
    - *int countOfNodes* – количество вершин графа;
    - *vector<int> IA, JA* – портрет матрицы;
    - *vector<double> A* – ненулевые коэффициенты матрицы;
    - *vector<double> x* – вектор, на который умножать;
    - *double allTime* – общее время, потраченное на текущий метод;
    - *double countOfCalls* – количество раз, сколько вызывался текущий метод.
- *vector<double> linearCombination* – реализация линейной комбинации двух векторов. Присутствует распараллеливание. Содержит следующие аргументы:
    - *vector<double> x1, x2* – вектора для кобинации;
    - *double a1, a2* – коэффициенты линейной комбинации;
    - *double allTime* – общее время, потраченное на текущий метод;
    - *double countOfCalls* – количество раз, сколько вызывался текущий метод.
- *void createMatrixFromA* – создание матрицы *M*, состоящей только из элементов главной диагонали матрицы *A*. Присутствует распараллеливание.
- *void reverseMatrix* – возведение матрицы в степень (-1). Получение обратной матрицы иными словами.
- *void solveSLAE* – реализация алгоритма решения СЛАУ итерационным методом с использованием метода сопряженных градиентов с предобуславливателем Якоби. Имеет следующие агрументы:
    - *vector<int> IA, JA* – портрет матрицы;
    - *vector<double> A* – ненулевые коэффициенты матрицы;
    - *vector<double> b* – вектор правой части;
    - *int countOfNodes* – количество вершин графа;
    - *double tol* – требуемая точность решения;

    - *int n* – количество итераций до получения решения;
    - *double res* – L2 норма невязки;
    - *vector<double> xRes* – вектор *x*, при котором удалось получить решение необходимой точности.
- *void printSolveVector* – печать вектора-решения СЛАУ *x*;
- *void doTask* – метод, выполняющий этап за этапом. Предполагалось, что можно вызвать несколько таких друг за другом с разным количеством потоков для наглядного представления преимуществ распараллеливания;
- *int main* – входная точка программы с инициализацией всех необходимых переменных.

Параметры запуска программы

Программа запускается со следующими параметрами:

1. *Nx* – ширина сетки. Целое число.
2. *Ny* – высота сетки. Целое число.
3. *k1* – количество несоединенных вершин графа. Целое число.
4. *k2* – количество соединенных вершин графа. Целое число.
5. *T* – количество нитей (поток), которые участвуют в распараллеленных участках программы. Целое число.
6. *tol* – необходимая точность решение СЛАУ. Число с плавающей точкой.
7. *isPrint* – необходимость печати детальной информации (таблица смежности, портрет матрицы, вектор решения СЛАУ). 0 – печать отключена, 1 – печать включена.

Опробованные методы оптимизации

В ходе выполнения практической работы возникали трудности с потреблением памяти и времени работы. Возникали проблемы, т. к. изначально вершина графа являлась объектов, хранящей вектор своих соседей (также объекты). В итоге программа отказывалась работать на размерах сетки более 15*15. Было принято решение реорганизовать вектор соседей вершины. Вместо объекта стал хранить всего лишь его индекс. Это позволило запускать программу на большем размере сетки. Заметив такой прирост производительности, убрал совсем объекты из программы, заменив на *map<int, vector<int>>*. Это также улучшило потребление памяти и скорость работы приложения. Детальнее рефакторинг кода можете наблюдать в репозитории

https://github.com/vlazarew/Parallel_Lab_1_Graph. При дальнейшем тестировании программы проведена оптимизация переменных, многие большие вектора были упразднены, чтоб во время выполнения в памяти хранились только необходимые данные без устаревших и утративших актуальность. Это позволило существенно уменьшить требуемую память для успешной работы приложения. Для лучшего параллелизма метода скалярного произведения был создан отдельный вектор с уже подготовленным размером. В него записывались результаты на каждой итерации, а затем уже вектор с готовыми решениями складывался. Это позволило улучшить параллелизм dot кернела. Была проблема, что на больших размерах сетки портрет матрицы долго генерировался. Это возникало из-за того, что для каждого элемента производился поиск в векторе вершин, которые необходимо соединить с верхней левой или нижней правой. Решением проблемы послужило сделать не поиск элемента в векторе, а небольшая реструктуризация вектора. Программа начинает требовать чуть больше памяти, но экономим существенное количество времени.

Исследование производительности

Характеристики вычислительной системы

ЭВМ 1 (ПК):

- Процессор – Intel Core i7-7700K;
- Количество ядер – 4;
- Базовая тактовая частота – 4,2 GHz;
- Максимальная тактовая частота – 4,5 GHz;
- GFLOPS – 249,6;
- Кеш-память – 8 MB;
- Частота системной шины – 8 GT/S;
- Количество общей оперативной памяти – 32 GB;
- ОС – *Linux Mint 20*;
- BW – 2666 * 2* 8 = 42 Гб/с.

ЭВМ 2 (Ноутбук):

- Процессор – Intel Core i5-8520U;
- Количество ядер – 4;
- Базовая тактовая частота – 1,6 GHz;
- Максимальная тактовая частота – 3,4 GHz;
- GFLOPS – 163,2;
- Кеш-память – 6 MB;
- Частота системной шины – 4 GT/S;
- Количество общей оперативной памяти – 4 GB;
- ОС – *Windows 10*;
- BW – 1866 * 1* 8 = 15 Гб/с.

Описание параметров компиляции:

В случае *Windows 10* для сборки проекта используется *gcc 8.1.0*, для отладки *gdb* из пакета *mingw-w64*.

В случае с *Linux Mint 20* для сборки используется *Clang 10.0.1*, для отладки *gdb*.

При использовании иных компляторов приложение может показывать *nan* в результатах решения. К отчету и исходному коду прилагается *CMakeFile*, в котором

указывается, что проект собирается под стандарт *C++ 11* с флагами *-O3, -fopenmp*.

AI для линейной комбинации

```
for (int i = 0; i < vectorSize; i++)
        {
            result.at(i) = x1.at(i) * a1 + x2.at(i) * a2;
        }
```

Число флопов – 3N. 2 операции умножения + 1 операция сложения.

N даблов на чтение вектора *x1*. N даблов на чтение *x2*. 2 дабла на чтение *a1* и *a2*. N даблов на запись в вектор *result*. 1 дабл = 8 байт. Итого: 8*(3N + 2).

Получаем AI для линейной комбинации: $\frac{3N}{8*(3N+2)}$.

AI для скалярного произведения

```
for (int i = 0; i < vectorSize; i++)
        {
            Result.at(i) = x1.at(i) * x2.at(i);
        }
```

Число флопов – 2N -1. Операция умножения + сложение.

N даблов на чтение вектора *x1*. N даблов на чтение *x2*. 1 дабл = 8 байт. Итого: 8*2N.

Получаем AI для скалярного произведения: $\frac{2N-1}{8*2N}$.

AI для матрично-векторного произведения

```
for (int i = 0; i < countOfNodes; i++)
        {
            for (int k = IA.at(i); k < IA.at(i + 1); k++)
            {
                result.at(i) += A.at(k) * x.at(JA.at(k));
            }
        }
```

Число флопов – 2N.

N даблов на чтение вектора *A*. N даблов на чтение *x*. N даблов на чтение *IA*. N даблов на чтение *JA*. N даблов на запись в вектор *result*. 1 дабл = 8 байт. Итого: 8*5N.

Получаем AI для матрично-векторного произведения: $\frac{2N}{8*5N}$, что равно 1/20.

Результаты измерения производительности

Последовательная производительность

Таблица для ЭВМ 1

Статистика ПО/Количество вершин N	**10^4**	**10^5**	**10^6**	**10^7**
Первый этап (генерирование портрета)				
Время выполнения, с	0,005	0,0708	0,973	11,2391
Потребляемая память, MB	1,2	14	156	1400
Второй этап (построение СЛАУ)				
Время выполнения, с	0,00157	0,0346	0,355965	3,5692
Третий этап (решение СЛАУ)				
Время выполнения, с	0,010865	0,1399	1,51488	24,5951
Скалярное произведение векторов				
Среднее время выполнения, с	0,000034	0,000376	0,004247	0,061764
TBP (AI = $\frac{2N-1}{8*2N}$)	5,249737 GLOPS, 2,1032 % TPP	5,249973 GLOPS, 2,1033 % TPP	5,249997 GLOPS, 2,1033 % TPP	5,249997 GLOPS, 2,1033 % TPP
Линейная комбинация				
Среднее время выполнения, с	0,000024	0,000281	0,003017	0,049655
TBP (AI = $\frac{3N}{8*(3N+2)}$)	5,249650 GLOPS, 2,1032 % TPP	5,249965 GLOPS, 2,1033 % TPP	5,249996 GLOPS, 2,1033 % TPP	5,249999 GLOPS, 2,1033 % TPP
Матрично-векторное произведение				
Среднее время выполнения, с	0,000101	0,001147	0,011663	0,139695

TBP (AI = 1/20)	1,05 GLOPS, 0,42 % TPP	1,05 GLOPS, 0,42 % TPP	1,05 GLOPS, 0,42 % TPP	1,05 GLOPS, 0,42 % TPP

Таблица для ЭВМ 2

Статистика ПО/Количество вершин N	**10^4**	**10^5**	**10^6**	**10^7**
Первый этап (генерирование портрета)				
Время выполнения, с	0,033	0,432	3,411	63,698
Потребляемая память, МВ	1,2	14	156	1400
Второй этап (построение СЛАУ)				
Время выполнения, с	0,003	0,028	0,282	2,991
Третий этап (решение СЛАУ)				
Время выполнения, с	0,142	0,368	2,224	24,049
Скалярное произведение векторов				
Среднее время выполнения, с	0,000155	0,000499	0,006085	0,050378
ТВР (AI = $\frac{2N-1}{8*2N}$)	1,874906 GLOPS, 1,148 % TPP	1,874991 GLOPS, 1,148 % TPP	1,874999 GLOPS, 1,148 % TPP	1,874999 GLOPS, 1,148% TPP
Линейная комбинация				
Среднее время выполнения, с	0,000057	0,000434	0,004590	0,037577
ТВР (AI = $\frac{3N}{8*(3N+2)}$)	1,874875 GLOPS, 1,148 % TPP	1,874988 GLOPS, 1,148 % TPP	1,874999 GLOPS, 1,148 % TPP	1,874999 GLOPS, 1,148% TPP
Матрично-векторное произведение				
Среднее время выполнения, с	0,000118	0,001179	0,01269	0,116879
ТВР (AI = 1/20)	0,75 GLOPS, 0,459 % TPP	0,75 GLOPS, 0,459 % TPP	0,75 GLOPS, 0,459 % TPP	0,75 GLOPS, 0,459 % TPP

Параллельное ускорение

Таблица для ЭВМ 1

Статистика ПО/Количество вершин N	**10^4**	**10^5**	**10^6**	**10^7**
Первый этап (генерирование портрета)				
T = 2, Время выполнения, с	0,00464	0,05459	0,756838	8,56875
T = 4. Время выполнения, с	0,003886	0,04746	0,623742	7,15642
T = 8. Время выполнения, с	0,027	0,059419	0,576877	6,69101
Второй этап (построение СЛАУ)				
T = 2, Время выполнения, с	0,000782	0,01939	0,177797	1,78528
T = 4. Время выполнения, с	0,000573	0,00873	0,082325	0,825234
T = 8. Время выполнения, с	0,00128	0,01314	0,061823	0,604726
Третий этап (решение СЛАУ)				
T = 2, Время выполнения, с	0,00707	0,0818	0,9709	17,6711
T = 4. Время выполнения, с	0,00602	0,05643	0,716301	14,8858
T = 8. Время выполнения, с	0,01	0,103	1,00241	14,814
Скалярное произведение векторов				
T = 2, Время выполнения, с	0,000025	0,000251	0,003078	0,051618
T = 4. Время выполнения, с	0,0000220119	0,000220243	0,00275571	0,049052
T = 8. Время выполнения, с	0,000177157	0,00093477	0,00453566	0,050920
Линейная комбинация				
T = 2, Время выполнения, с	0,0000147153	0,000153723	0,00182947	0,039245
T = 4. Время выполнения, с	0,0000100578	0,000129563	0,00151128	0,037004
T = 8. Время выполнения, с	0,000022619	0,000245685	0,00223275	0,037818
Матрично-векторное произведение				
T = 2, Время выполнения, с	0,0000604461	0,000627318	0,00669869	0,084801
T = 4. Время выполнения, с	0,0000474953	0,000551549	0,00436277	0,060396
T = 8. Время выполнения, с	0,0000472697	0,00115728	0,00590556	0,058222

Таблица для ЭВМ 2

Статистика ПО/Количество вершин N	**10^4**	**10^5**	**10^6**	**10^7**
Первый этап (генерирование портрета)				
T = 2, Время выполнения, с	0,0439	0,515	4,423	108,451
T = 4. Время выполнения, с	0,056	0,609	5,524	128,51
T = 8. Время выполнения, с	0,06699	0,868	7,931	174,448
Второй этап (построение СЛАУ)				
T = 2, Время выполнения, с	0,003	0,028	0,282	2,991
T = 4. Время выполнения, с	0,000999	0,016	0,14	1,646
T = 8. Время выполнения, с	0,001	0,008	0,078	0,948
Третий этап (решение СЛАУ)				
T = 2, Время выполнения, с	0,153	0,389	1,795	19,193
T = 4. Время выполнения, с	0,139	0,354	1,795	18,435
T = 8. Время выполнения, с	0,154	0,368	1,767	19,513
Скалярное произведение векторов				
T = 2, Время выполнения, с	0,0001206	0,000803023	0,00585715	0,0466097
T = 4. Время выполнения, с	0,000206898	0,00080304	0,00588572	0,0517317
T = 8. Время выполнения, с	0,00029309	0,00101515	0,0064	0,0485122
Линейная комбинация				
T = 2, Время выполнения, с	0,000103454	0,000444436	0,00416191	0,0371057
T = 4. Время выполнения, с	0,000137935	0,000525253	0,00421904	0,383252
T = 8. Время выполнения, с	0,000137932	0,000474747	0,00431428	0,0382114
Матрично-векторное произведение				
T = 2, Время выполнения, с	0,000135596	0,001	0,00788732	0,0747711
T = 4. Время выполнения, с	0,00010168	0,000999	0,00685915	0,659398
T = 8. Время выполнения, с	0,000288135	0,0011194	0,00694366	0,0671446

Анализ производительности

ЭВМ 1

Статистика ПО/Количество вершин N	**10^4**	**10^5**	**10^6**	**10^7**
Скалярное произведение векторов				
T = 1, GFLOPS (N)	0,29135	0,265253	0,23546	0,161905
T = 2, GFLOPS (N)	0,386018	0,398051	0,324864	0,193729
T = 4, GFLOPS (N)	0,4543	0,454044	0,362883	0,203864
T = 8, GFLOPS (N)	0,056447	0,106978	0,220475	0,196385
Линейная комбинация				
T = 1, GFLOPS (N)	0,408619	0,355148	0,331365	0,20139
T = 2, GFLOPS (N)	0,679565	0,650521	0,546606	0,254806
T = 4, GFLOPS (N)	0,994253	0,771825	0,661691	0,270234
T = 8, GFLOPS (N)	0,442106	0,407025	0,447878	0,264424
Матрично-векторное произведение				
T = 1, GFLOPS (N)	0,098712	0,087157	0,085738	0,071585
T = 2, GFLOPS (N)	0,165437	0,159409	0,149283	0,117922
T = 4, GFLOPS (N)	0,210547	0,181308	0,229212	0,165572
T = 8, GFLOPS (N)	0,211552	0,08641	0,169332	0,171756
3-ий этап (алгоритм решения)				
T = 1, GFLOPS (N)	0,015646	0,012152	0,011222	0,006912
T = 2, GFLOPS (N)	0,024045	0,020782	0,01751	0,00962
T = 4, GFLOPS (N)	0,028239	0,030126	0,023733	0,01142
T = 8, GFLOPS (N)	0,017	0,016505	0,016959	0,011476

ЭВМ 2

Статистика ПО/Количество вершин N	**10^4**	**10^5**	**10^6**	**10^7**
Скалярное произведение векторов				
T = 1, GFLOPS (N)	0,128943	0,400001	0,328639	0,396998
T = 2, GFLOPS (N)	0,165829	0,249058	0,341463	0,429095
T = 4, GFLOPS (N)	0,096661	0,249052	0,339805	0,38661
T = 8, GFLOPS (N)	0,068235	0,197014	0,3125	0,412267
Линейная комбинация				
T = 1, GFLOPS (N)	0,522021	0,690675	0,653526	0,798354
T = 2, GFLOPS (N)	0,289984	0,675013	0,720823	0,808501
T = 4, GFLOPS (N)	0,217494	0,571153	0,711062	0,078277
T = 8, GFLOPS (N)	0,217498	0,631916	0,695365	0,785106
Матрично-векторное произведение				
T = 1, GFLOPS (N)	0,168577	0,169622	0,157603	0,171117
T = 2, GFLOPS (N)	0,147497	0,2	0,253572	0,267483
T = 4, GFLOPS (N)	0,196696	0,2002	0,291581	0,303307
T = 8, GFLOPS (N)	0,069412	0,178667	0,288033	0,297865
3-ий этап (алгоритм решения)				
T = 1, GFLOPS (N)	0,001197	0,00462	0,007644	0,007069
T = 2, GFLOPS (N)	0,001111	0,00437	0,009471	0,008857
T = 4, GFLOPS (N)	0,001223	0,004802	0,009471	0,009222
T = 8, GFLOPS (N)	0,001104	0,00462	0,009621	0,008712

Анализ полученных результатов

Исходя из данных, приведенных выше видно, что на 2-ух рабочих станциях ТВР операций dot, axpby, spMV находится в диапазоне от 0,75% до 2% ТРР. То есть теоретически достижимая производительность всего до 1% на *Windows* платформе и до 2% на *Linux* платформах. На деле достигается еще меньшие показатели. Исходя из полученных результатов можно предполагать, что на малом размере сетки выгоднее запускать приложение на 1-ом потоке (т. к. оба экземпляра оснащены технологией *Intel Turbo Boost*), либо на 4-ех потоках. На большом размере сетки преимущество 8-ми потоков полностью

раскрывается. Исходя из приведенных выше сведений о производительности программы от разного количество N и T (количества потоков) можно сделать вывод, что на матрицах больше 10^7 лучше использовать 8 поток, если размер меньше –целесообразнее пробовать 2 или 4 потока. Если же матрица совсем мала, то процессоры, оснащенные технологией *Intel Turbo Boost*, используя всего 1 поток, могут превосходить в производительности многопоточное выполнение той же самой программы.