

Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной
математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Численные методы»
на тему
«Распараллеливание алгоритмов решения задач линейной алгебры»

Студентка: А. А. Довженко
Преподаватель: Д. Л. Ревизников
Группа: М8О-307Б
Дата:
Оценка:
Подпись:

Москва, 2019

1 Описание

В качестве алгоритма для распараллеливания был выбран метод вращений для нахождения собственных значений симметричных матриц.

Для реализации параллелизма был выбран модуль mpi4py, который является оберткой над MPI — API для передачи сообщений между процессами, выполняющими одну задачу.

Есть два способа параллелить эту задачу: первый — распараллелить вспомогательные функции (нахождение максимального элемента матрицы, перемножение матриц, транспонирование матрицы), второй — разделить задачу на подзадачи, каждая из которых будет вычисляться независимо от другой. Мной был выбран второй способ.

Сначала опишем сам метод, потом использованные технологии распараллеливания (MPI), а затем рассмотрим получившуюся реализацию.

1 Метод

Суть метода заключается в том, чтобы для заданной симметрической матрицы $A = A^{(0)}$ построить последовательность ортогонально подобных матриц $A^{(1)}, A^{(2)}, \dots, A^{(m)}$, сходящуюся к диагональной матрице, на диагонали которой стоят собственные значения A . Для построения этой последовательности применяется специально подобранная матрица вращения U , такая что норма наддиагональной части $\|A^{(i)}\| = \sqrt{\sum_{1 \leq j < k \leq n} (a_{jk}^{(i)})^2}$ уменьшается при каждом двустороннем вращении матрицы $A^{(i+1)} = U_i^T A^{(i)} U_i$.

Это достигается выбором максимального по абсолютной величине элемента матрицы $A^{(i)}$ и его обнулением в матрице $A^{(i+1)}$. Если он расположен в j -й строке и k -м столбце, то

$$U_i = \begin{matrix} & & & & & & & & \\ & & & & & & & & \\ & & & & & & & & \\ & & & & & & & & \\ j & & & & & & & & \\ & & & & & & & & \\ k & & & & & & & & \\ & & & & & & & & \end{matrix} \begin{bmatrix} 1 & & & & & & & & \\ & 1 & & & & & & & \\ & & \ddots & & & & & & \\ & & & \cos(\theta) & & -\sin(\theta) & & & \\ & & & & \ddots & & & & \\ & & & \sin(\theta) & & \cos(\theta) & & & \\ & & & & & & \ddots & & \\ & & & & & & & 1 & \\ & & & & & & & & 1 \end{bmatrix}$$

Если обозначить $s = \sin \theta$ и $c = \cos \theta$, то матрица $A^{(i+1)}$ состоит из следующих элементов, отличающихся от элементов $A^{(i)}$:

$$a_{jj}^{(i+1)} = c^2 a_{jj}^{(i)} - 2sc a_{jk}^{(i)} + s^2 a_{kk}^{(i)} \quad (1)$$

$$a_{kk}^{(i+1)} = s^2 a_{jj}^{(i)} + 2sc a_{jk}^{(i)} + c^2 a_{kk}^{(i)} \quad (2)$$

$$a_{jk}^{(i+1)} = a_{kj}^{(i+1)} = (c^2 - s^2) a_{jk}^{(i)} + sc (a_{kk}^{(i)} - a_{jj}^{(i)}) \quad (3)$$

$$a_{jm}^{(i+1)} = a_{mj}^{(i+1)} = c a_{jm}^{(i)} - s a_{km}^{(i)} \quad m \neq j, k \quad (4)$$

$$a_{km}^{(i+1)} = a_{mk}^{(i+1)} = s a_{jm}^{(i)} + c a_{km}^{(i)} \quad m \neq j, k \quad (5)$$

$$a_{ml}^{(i+1)} = a_{ml}^{(i)} \quad m, l \neq j, k \quad (6)$$

Если $a_{jj}^{(i)} = a_{kk}^{(i)}$, то выбирается $\theta = \frac{\pi}{4}$, в противном случае вводится $t = \frac{s}{c} = \text{tg}(\theta)$ и тогда $t^2 - 2t\tau + 1 = 0$. Решение квадратного уравнения даёт $t = \frac{\text{sign}(\tau)}{|\tau| + \sqrt{1 + \tau^2}}, c = \frac{1}{\sqrt{1 + t^2}}, s = tc$.

Вычисление останавливается, когда выполняются критерии близости к диагональной матрице. Это малость максимального по абсолютной величине внедиагонального элемента матрицы.

2 MPI

MPI расширяется как "Message passing interface" ("Взаимодействие через передачу сообщений"). MPI предоставляет программисту единый механизм взаимодействия ветвей внутри параллельного приложения независимо от машинной архитектуры (однопроцессорные/многопроцессорные с общей/раздельной памятью), взаимного расположения ветвей (на одном процессоре/на разных) и API операционной системы.

Программа, использующая MPI, легче отлаживается (сужается простор для совершения стереотипных ошибок параллельного программирования) и быстрее переносится на другие платформы (в идеале, простой перекомпиляцией).

Параллельное приложение состоит из нескольких ветвей, или процессов, или задач, выполняющихся одновременно. Разные процессы могут выполняться как на разных процессорах, так и на одном и том же — для программы это роли не играет, поскольку в обоих случаях механизм обмена данными одинаков. Процессы обмениваются друг с другом данными в виде сообщений. Сообщения проходят под идентификаторами, которые позволяют программе и библиотеке связи отличать их друг от друга. Для совместного проведения тех или иных расчетов процессы внутри приложения объединяются в группы. Каждый процесс может узнать у библиотеки связи свой

номер внутри группы, и, в зависимости от номера приступает к выполнению соответствующей части расчетов.

Термин «процесс» используется также в Unix: в MPI ветвь запускается и работает как обычный процесс Unix, связанный через MPI с остальными процессами, входящими в приложение. В остальном процессы следует считать изолированными друг от друга: у них разные области кода, стека и данных, процессы имеют отдельную память.

3 Реализация

Как было сказано ранее, я попыталась разбить задачу на подзадачи и вычислять их отдельно друг от друга. Алгоритм реализован следующим образом: в нулевом процессе вычисляются значения матрицы поворота и аккумулируются данные из других процессов, оставшиеся процессы разбиваются на две группы (с четными и нечетными номерами), группа нечетных процессов считает строку j , а группа четных процессов – строку k . На каждой итерации изначально нулевой процесс считает значения косинуса и синуса для поворота, и отправляет другим процессам эти значения, индексы наибольшего элемента и соответствующие им строки. Все процессы, кроме нулевого, разбиваются на две группы. Группы разбивают j -ую и k -ую строки на блоки по количеству процессов в группе. Каждый процесс пересчитывает свой блок. После пересчета все блоки объединяются и получается новая строка, которая отправляется нулевому процессу. Нулевой процесс получает эти данные и помещает их в матрицу A . Каждая итерация заканчивается пересчетом параметра t , который считает нулевой процесс и рассылает по другим процессам.

2 Исходный код

```
1 import argparse
2 import json
3
4 from utils import save_to_file
5
6 from mpi4py import MPI
7 import numpy as np
8
9
10 def read_data(filename, need_args):
11     init_dict = {}
12     with open(filename, 'r') as json_data:
13         data = json.load(json_data)[0] # !
14
15         for arg in need_args:
16             if arg not in data:
17                 raise ValueError('No "{0}" in given data'.format(arg))
18
19             if arg == 'matrix':
20                 init_dict[arg] = np.array(data[arg], dtype=np.float64)
21             else:
22                 init_dict[arg] = data[arg]
23
24     return init_dict
25
26
27 def sign(n):
28     return 1 if n > 0 else -1
29
30
31 def t(A):
32     return np.sqrt(sum([A[i, j] ** 2 for i in range(A.shape[0])
33                     for j in range(i + 1, A.shape[0])]))
34
35
36 def indexes_max_elem(A):
37     i_max = j_max = 0
38     a_max = A[0, 0]
39     for i in range(A.shape[0]):
40         for j in range(i + 1, A.shape[0]):
41             if abs(A[i, j]) > a_max:
42                 a_max = abs(A[i, j])
43                 i_max, j_max = i, j
44     return i_max, j_max
45
46
47 def parallel_jacobi_rotate(comm, A, ind_j, ind_k):
```

```

48     sz = A.shape[0]
49     rank = comm.Get_rank()
50     pool_size = comm.Get_size()
51     c = s = 0.0
52     j = k = 0
53     row_j, row_k = np.zeros(sz), np.zeros(sz)
54     if rank == 0:
55         j, k = ind_j, ind_k
56
57         if A[j, j] == A[k, k]:
58             c = np.cos(np.pi / 4)
59             s = np.sin(np.pi / 4)
60         else:
61             tau = (A[j, j] - A[k, k]) / (2 * A[j, k])
62             t = sign(tau) / (abs(tau) + np.sqrt(1 + tau ** 2))
63             c = 1 / np.sqrt(1 + t ** 2)
64             s = c * t
65
66         for i in range(sz):
67             row_j[i] = A[j, i]
68             row_k[i] = A[k, i]
69
70     j = comm.bcast(j, root=0)
71     k = comm.bcast(k, root=0)
72     c = comm.bcast(c, root=0)
73     s = comm.bcast(s, root=0)
74     comm.Bcast(row_j, root=0)
75     comm.Bcast(row_k, root=0)
76
77     row_j_comm = comm.Create_group(comm.group.Incl([i for i in range(1, pool_size) if i
78         % 2 == 1]))
79
80     row_k_comm = comm.Create_group(comm.group.Incl([i for i in range(1, pool_size) if i
81         % 2 == 0]))
82
83     row_j_rank = row_j_size = -1
84     row_j_new = np.zeros(sz)
85     if MPI.COMM_NULL != row_j_comm:
86         row_j_rank = row_j_comm.Get_rank()
87         row_j_size = row_j_comm.Get_size()
88         size = int(sz / row_j_size)
89         row_j_part = np.zeros(size)
90         row_k_part = np.zeros(size)
91         row_j_new_part = np.zeros(size)
92
93         row_j_comm.Scatter(row_j, row_j_part, root=0)
94         row_k_comm.Scatter(row_k, row_k_part, root=0)
95
96         for i in range(size):
97             row_j_new_part[i] = c * row_j_part[i] + s * row_k_part[i]

```

```

95
96     row_j_comm.Gather(row_j_new_part, row_j_new, root=0)
97     if row_j_rank == 0:
98         comm.Send([row_j_new, sz, MPI.FLOAT], dest=0, tag=0)
99     row_j_comm.Free()
100
101     row_k_rank = row_k_size = -1
102     row_k_new = np.zeros(sz)
103     if MPI.COMM_NULL != row_k_comm:
104         row_k_rank = row_k_comm.Get_rank()
105         row_k_size = row_k_comm.Get_size()
106         size = int(sz / row_k_size)
107         row_j_part = np.zeros(size)
108         row_k_part = np.zeros(size)
109         row_k_new_part = np.zeros(size)
110
111         row_k_comm.Scatter(row_j, row_j_part, root=0)
112         row_k_comm.Scatter(row_k, row_k_part, root=0)
113
114         for i in range(size):
115             row_k_new_part[i] = s * row_j_part[i] - c * row_k_part[i]
116
117         row_k_comm.Gather(row_k_new_part, row_k_new, root=0)
118         if row_k_rank == 0:
119             comm.Send([row_k_new, sz, MPI.FLOAT], dest=0, tag=0)
120         row_k_comm.Free()
121
122     if rank == 0:
123         status = MPI.Status()
124         comm.Recv([row_j_new, sz, MPI.FLOAT], source=1, tag=0, status=status)
125         comm.Recv([row_k_new, sz, MPI.FLOAT], source=2, tag=0, status=status)
126
127         A[j, k] = A[k, j] = (c ** 2 - s ** 2) * row_j[k] + s * c * (row_k[k] - row_j[j
128             ])
129         A[j, j] = c ** 2 * row_j[j] + 2 * s * c * row_j[k] + s ** 2 * row_k[k]
130         A[k, k] = s ** 2 * row_j[j] - 2 * s * c * row_j[k] + c ** 2 * row_k[k]
131
132         for i in range(sz):
133             if i != j and i != k:
134                 A[j, i] = A[i, j] = row_j_new[i]
135                 A[k, i] = A[i, k] = row_k_new[i]
136
137     return A
138
139
140 def jacobi_parallel(comm, A, eps):
141     elapsed_time = 0
142     i, j = indexes_max_elem(A)

```

```

143     norm = t(A)
144     rank = comm.Get_rank()
145     eps = comm.bcast(eps, root=0)
146     norm = comm.bcast(norm, root=0)
147
148     k = 1
149     while norm > eps:
150         elapsed_time -= MPI.Wtime()
151         A = parallel_jacobi_rotate(comm, A, j, i)
152         if rank == 0:
153             norm = t(A)
154             elapsed_time += MPI.Wtime()
155             norm = comm.bcast(norm, root=0)
156             i, j = indexes_max_elem(A)
157             k += 1
158
159     return np.diag(A).tolist()
160
161
162 if __name__ == "__main__":
163     parser = argparse.ArgumentParser()
164     parser.add_argument('--input', required=True, help='Input file')
165     parser.add_argument('--output', required=True, help='Output file')
166     args = parser.parse_args()
167
168     elapsed_time = 0
169     need_args = ('matrix', 'eps')
170     init_dict = read_data(args.input, need_args)
171     A, eps = init_dict['matrix'], init_dict['eps']
172
173     comm = MPI.COMM_WORLD
174     rank = comm.Get_rank()
175
176     elapsed_time -= MPI.Wtime()
177     eig = jacobi_parallel(comm, A, eps)
178     elapsed_time += MPI.Wtime()
179
180     if rank == 0:
181         save_to_file(args.output, eigenvalues=eig)
182         print("Dimension {0}, time elapsed {1}\n".format(A.shape[0], elapsed_time))
183
184     MPI.Finalize()

```

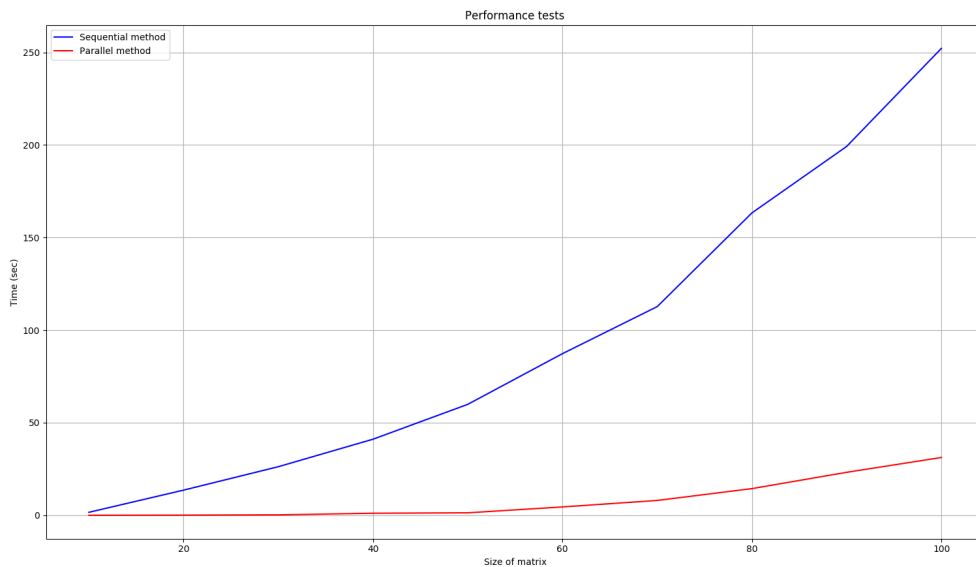

3 Тест производительности

Для бенчмарка было выбрано два кейса: тесты производительности и тесты с разным количеством процессов.

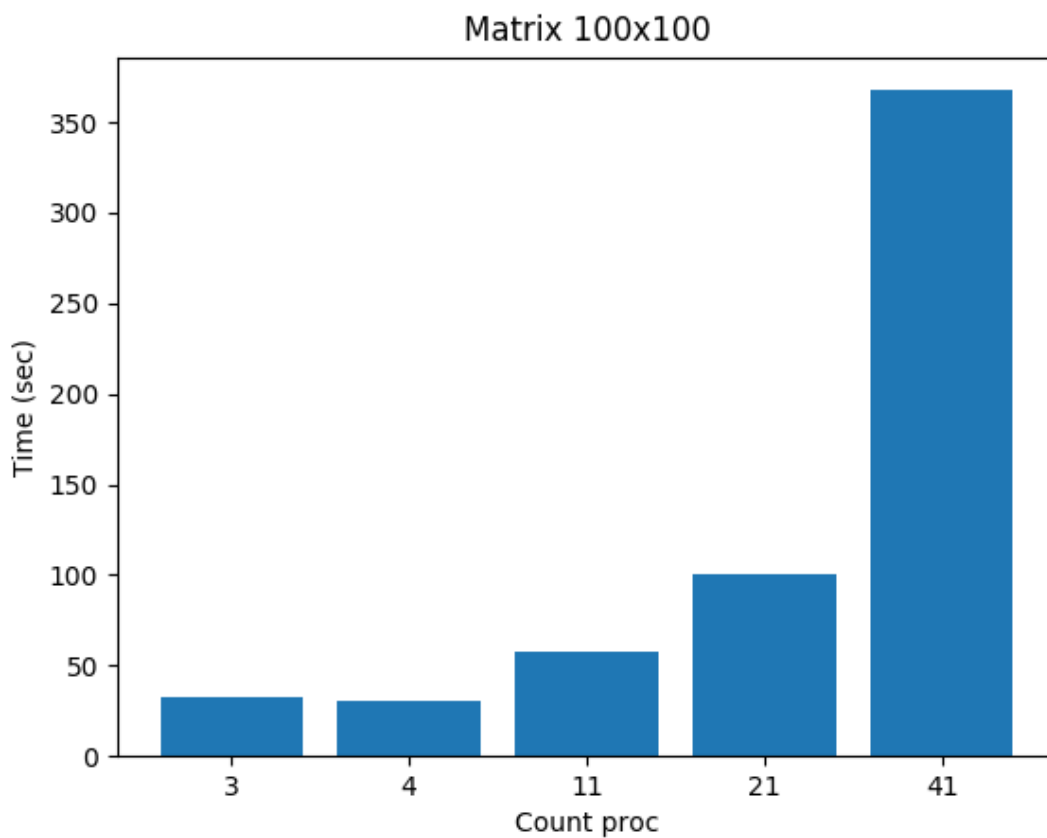
Тесты производительности представляют собой 10 матриц размерами от 10 до 100 с шагом 10, все значения матриц лежат в отрезке $[-2000, 2000]$. Сначала тесты были запущены на однопоточной версии программы, а затем на многопроцессорной.

Тесты с разным количеством процессов представляют собой один тестовый файл с матрицей размерами 10 на 10, для которой вычисление собственных значений запускается на 3, 4, 11, 21 и 41 процессах.

Полученные результаты отражены на графиках.



Как видно, распараллеливание сильно ускорило работу программы. Но также стоит отметить, что однопоточная версия была написана на чистом Python, без использования `numru`, тогда как многопроцессорная версия использует обертку над MPI, который реализован на Fortran и C, языках которые значительно превосходят Python по скорости вычислений.



Количество процессов напрямую влияет на скорость работы распараллеленного алгоритма. При 4 процессах время имеет наименьшее значение. Это связано с тем, что на моей машине 4 ядра, то есть в один момент времени могут выполняться максимум 4 процесса. С ростом количества процессов растет и время, потому что, во-первых, тратится время на переключение контекста между процессами, во-вторых, процессы дольше разбивают данные на блоки и синхронизируют полученные результаты.

4 Выводы

Метод Якоби является самым медленным из имеющихся алгоритмов вычисления собственных значений симметричной матрицы. Кроме того, он не детерминирован, так как является итерационным алгоритмом с выходом по точности: число итераций зависит от входных данных и значения точности. Метод Якоби практически не применяется в библиотеках для параллельных вычислительных систем.

Недостаток моей реализации в том, что количество процессов – подбираемый параметр, зависящий от размера матрицы. Так как создаются две группы процессов, которые пересчитывают строки, длина строки должна делиться на количество процессов (n) в одной группе, потому что строка будет разбита на n равных блоков.

В этом курсовом проекте я познакомилась с технологией распараллеливания MPI, с которой мне придется работать на 4 курсе. Хорошо, что я ее «потрогала» уже сейчас.

Список литературы

- [1] *Методические материалы, Раздел 1. Численные методы линейной алгебры.*
URL: <https://mainfodotru.files.wordpress.com/2017/09/numeric-methods-part1.pdf> (дата обращения: 05.06.2019).
- [2] *Классический метод Якоби (вращений) для симметричных матриц с выбором по всей матрице — AlgoWiki.*
URL: [https://algowiki-project.org/ru/Классический_метод_Якоби_\(вращений\)_для_симметричных_матриц_с_выбором_по_всей_матрице](https://algowiki-project.org/ru/Классический_метод_Якоби_(вращений)_для_симметричных_матриц_с_выбором_по_всей_матрице) (дата обращения: 07.06.2019).