Лабораторная работа №2

Задание выполнила: Самсонова Карина, группа 20ПИ-1

Выполнение лабораторной работы происходило в команде.

Участники команды:

* Самсонова Карина, группа 20ПИ-1
* Корнев Егор, группа 20ПИ-1
* Лисунов Никита, группа 20ПИ-2

Отчет по заданию 1

Оглавление

[1. Формат ввода/вывода 1](#_Toc153652275)

[2. Умножение матрицы на вектор по строкам 2](#_Toc153652276)

[2.1. Алгоритм: 2](#_Toc153652277)

[2.2. Исследование производительности 2](#_Toc153652278)

[3. Умножение матрицы на вектор по столбцам 4](#_Toc153652279)

[3.1. Алгоритм: 4](#_Toc153652280)

[3.2. Исследование производительности 4](#_Toc153652281)

[4. Умножение матрицы на вектор по блокам 6](#_Toc153652282)

[4.1. Алгоритм: 6](#_Toc153652283)

[4.2. Исследование производительности 7](#_Toc153652284)

# Формат ввода/вывода

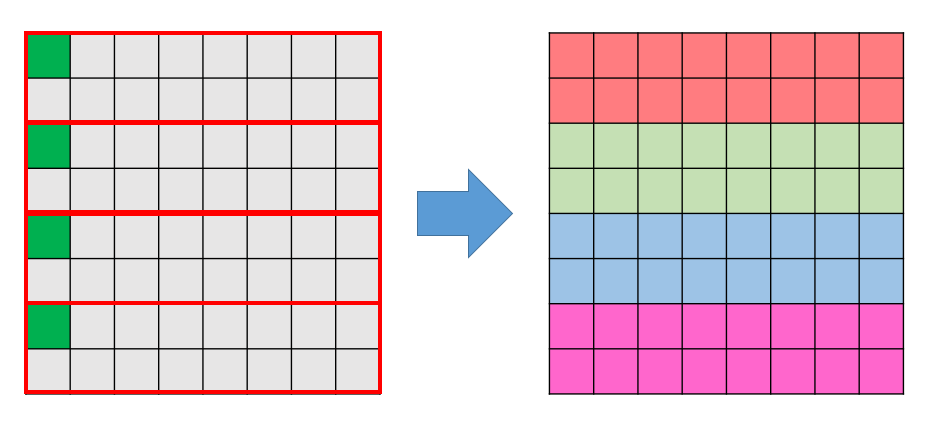
Матрица, исходный вектор и результат – массивы типа double. Для всех алгоритмов входные данные генерируются рандомно (с использованием rand из ‘math.h’, seed зависит от времени). Можно задать размерность матрицы с помощью макросов M (число строк) и N (число столбцов) и указать число процессов при запуске программы. Также есть функция fillToCheck, которая делает матрицу единичной, а исходный вектор заполняет последовательными числами (1, 2, 3…). С помощью этой функции легко проверять правильность подсчетов программы. После выполнения подсчетов, программа выводит в консоль время их выполнения, также можно напечатать исходную матрицу, вектор и результат умножения, выставив #define PRINT.

# 2. Умножение матрицы на вектор по строкам

# 2.1. Алгоритм:

1) Разделяем вектор *х* на равномерные части и отдаем потокам с помощью MPI\_Scatter.

2) Разделяем матрицу *a* на блоки строк *a\_local* и отдаем процессам с помощью MPI\_Scatterv. Элементы, входящие в displs, выделены зеленым, sendcounts и recvcounts рассчитаны так, чтобы в блок вошли все ячейки матрицы в пределах красной рамки.



3) Собираем исходный вектор в массив *x\_full* с помощью MPI\_Allgather.

4) На каждом процессе идем по строкам внутри блока *a\_local*,каждую строку умножаем на вектор *x\_full* и результат прибавляем к *y\_local*.

5) Собираем *y\_local* в *y* с помощью MPI\_Gather.

# 2.2. Исследование производительности

Время

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размер матрицы | 1 процесс | 2 процесса | 4 процесса | 8 процессов |
| 1000 | 0,006242 | 0,005309 | 0,005153 | 0,005335 |
| 2000 | 0,02417 | 0,018841 | 0,013913 | 0,014133 |
| 3000 | 0,05554 | 0,041795 | 0,032183 | 0,032191 |
| 4000 | 0,100048 | 0,077285 | 0,061096 | 0,09124 |
| 5000 | 0,156391 | 0,123679 | 0,104835 | 0,13322 |
| 10000 | 0,623137 | 0,458041 | 0,378901 | 0,567118 |

Ускорение

|  |  |  |  |
| --- | --- | --- | --- |
| Размер матрицы | 2 процесса | 4 процесса | 8 процессов |
| 1000 | 1,175739311 | 1,211333204 | 1,170009372 |
| 2000 | 1,282840614 | 1,737224179 | 1,710181844 |
| 3000 | 1,328867089 | 1,725755834 | 1,725326955 |
| 4000 | 1,294533221 | 1,637554013 | 1,096536607 |
| 5000 | 1,264491142 | 1,491782325 | 1,173930341 |
| 10000 | 1,360439349 | 1,644590539 | 1,098778385 |

Эффективность

|  |  |  |  |
| --- | --- | --- | --- |
| Размер матрицы | 2 процесса | 4 процесса | 8 процессов |
| 1000 | 0,587869655 | 0,302833301 | 0,146251172 |
| 2000 | 0,641420307 | 0,434306045 | 0,21377273 |
| 3000 | 0,664433545 | 0,431438958 | 0,215665869 |
| 4000 | 0,647266611 | 0,409388503 | 0,137067076 |
| 5000 | 0,632245571 | 0,372945581 | 0,146741293 |
| 10000 | 0,680219675 | 0,411147635 | 0,137347298 |

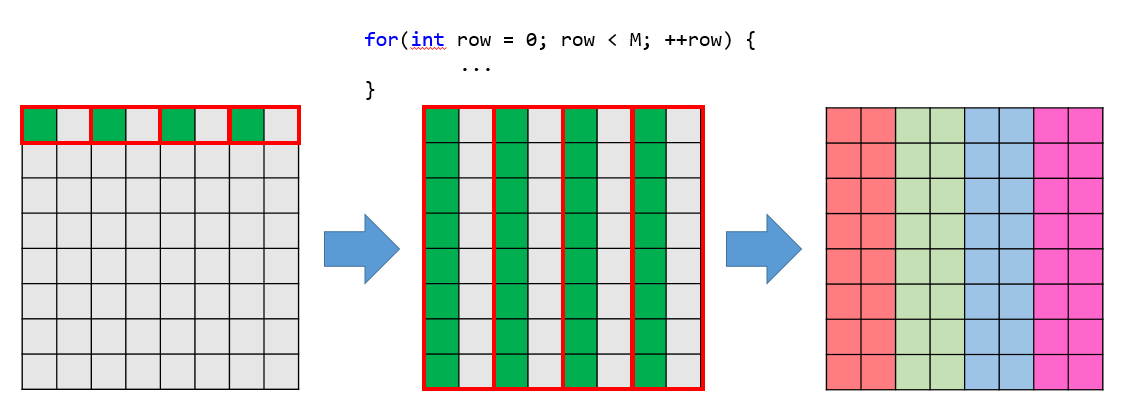
Вывод: можно заметить, что время выполнения падает с ростом количества процессов, но к 8 процессам начинает увеличиваться, это нормально, учитывая то, что у меня 4-ядерный процессор. Только на маленьких данных ускорение не столь большое, но в целом оно достигает 1.74 на 4 процессах, M=N=2000. Пик эффективности наблюдается на 2 процессах, но на 4 она падает не так сильно.

# 3. Умножение матрицы на вектор по столбцам

# 3.1. Алгоритм:

1) Разделяем вектор *х* на равномерные части *x\_local* и отдаем потокам с помощью MPI\_Scatter.

2) Разделяем матрицу *a* на блоки столбцов *a\_local* и отдаем процессам с помощью MPI\_Scatterv. Для этого мы проходимся по строкам исходной матрицы и разделяем элементы каждой строки так, что их ширина равна ширине итоговых столбцов. Элементы, входящие в *displs*, выделены зеленым, sendcounts и recvcounts рассчитаны так, чтобы в блок вошли все ячейки матрицы в пределах красной рамки. По мере прохождения по строкам, получившиеся блоки выстраиваются в столбцы.



3) Далее проходим по всем столбцам в *a\_local* и умножаем элементы в каждом столбце на соответствующий элемент в *x\_local*, результат прибавляем к *y\_local*.

4) Суммируем соответствующие элементы *y\_local* в *y* с помощью MPI\_Reduce.

# 3.2. Исследование производительности

Время

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размер матрицы | 1 процесс | 2 процесса | 4 процесса | 8 процессов |
| 1000 | 0,00796 | 0,005145 | 0,00382 | 0,005207 |
| 2000 | 0,040204 | 0,022138 | 0,016307 | 0,022141 |
| 3000 | 0,098176 | 0,055343 | 0,036451 | 0,040609 |
| 4000 | 0,18344 | 0,095601 | 0,057541 | 0,066602 |
| 5000 | 0,29328 | 0,159655 | 0,101067 | 0,155542 |
| 10000 | 1,430591 | 0,681157 | 0,435946 | 0,506429 |

Ускорение

|  |  |  |  |
| --- | --- | --- | --- |
| Размер матрицы | 2 процесса | 4 процесса | 8 процессов |
| 1000 | 1,547133139 | 2,083769634 | 1,52871135 |
| 2000 | 1,816062878 | 2,465444288 | 1,81581681 |
| 3000 | 1,773955152 | 2,693369181 | 2,417592159 |
| 4000 | 1,918808381 | 3,187987696 | 2,754271643 |
| 5000 | 1,836960947 | 2,901837395 | 1,88553574 |
| 10000 | 2,100236803 | 3,281578452 | 2,824859951 |

Эффективность

|  |  |  |  |
| --- | --- | --- | --- |
| Размер матрицы | 2 процесса | 4 процесса | 8 процессов |
| 1000 | 0,773566569 | 0,520942408 | 0,191088919 |
| 2000 | 0,908031439 | 0,616361072 | 0,226977101 |
| 3000 | 0,886977576 | 0,673342295 | 0,30219902 |
| 4000 | 0,95940419 | 0,796996924 | 0,344283955 |
| 5000 | 0,918480474 | 0,725459349 | 0,235691967 |
| 10000 | 1,050118401 | 0,820394613 | 0,353107494 |

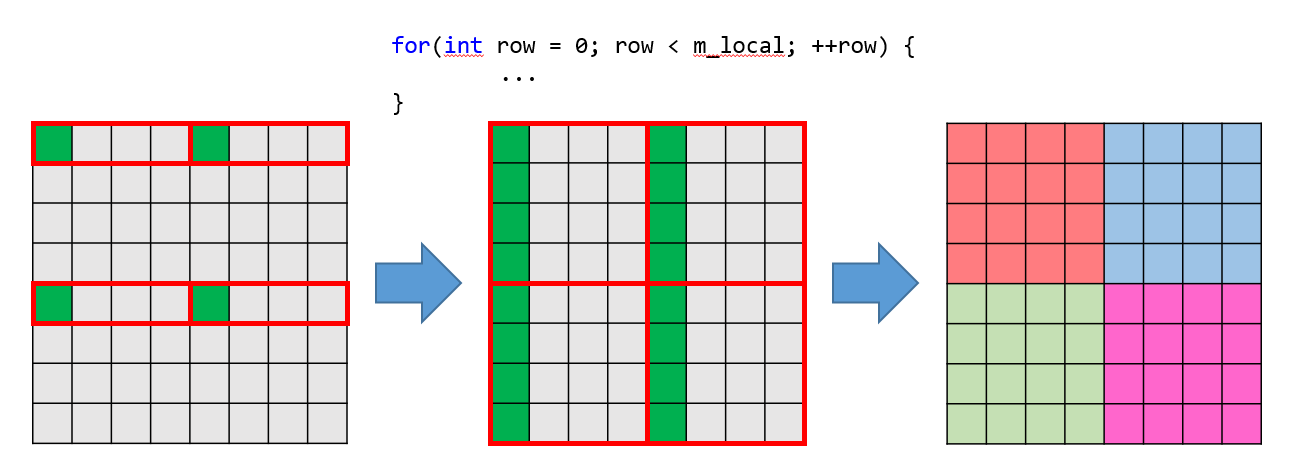
Вывод: время так же падает до 8 процессов, ускорение уже достигает максимума в 3.3, на 4 процессах при матрице 10000\*10000. Эффективность на матрице того же размера, но на 2 процессах превышает 1. Мне лично кажется, что это очень хороший показатель.

# 4. Умножение матрицы на вектор по блокам

# 4.1. Алгоритм:

1) Находим размерность блока с помощью отдельной функции. Для этого берем переменную *s*, которая содержиткорень из числа процессов и двигаемся от корня к 0 в поиске множителей *s* и *q=comm\_sz/s*, которым кратны стороны исходной матрицы. Если таких множителей при определенных входных данных найти нельзя, программа выводит сообщение об ошибке и завершается.  
2) Разделяем вектор *х* на равномерные части *x\_local* и отдаем потокам с помощью MPI\_Scatter.

3) Разделяем матрицу a на блоки *a\_local* и отдаем процессам с помощью MPI\_Scatterv. Этот алгоритм похож на тот, что используется при разбиении на столбцы, только здесь displs равномерно разбросаны не вдоль первой строки, а вдоль всей матрицы, и проходим мы не по всем строкам, а только в пределах высоты блока *m\_local*.



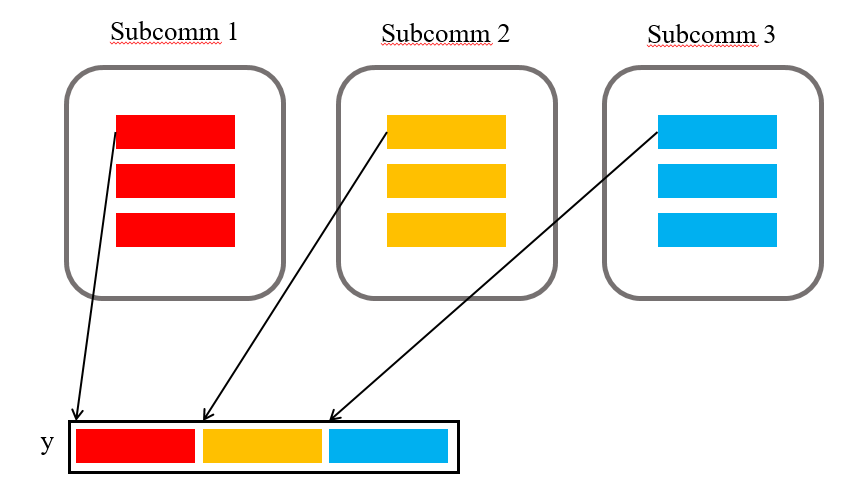
4) Собираем исходный вектор в массив *x\_full* с помощью MPI\_Allgather.

5) На каждом процессе идем по строкам внутри блока *a\_local*,каждую строку умножаем на соответствующую часть вектора *x\_full* и результат прибавляем к *y\_local*.

6) Теперь нужно сложить между собой промежуточные результаты. Для этого нужно суммировать те *y\_local*, которые относятся к одному ряду блоков (на картинке сверху красный и синий блоки относятся к ряду 0, а зеленый и фиолетовый – к ряду 1). Чтобы это сделать, нужно создать отдельные коммуникаторы с помощью MPI\_Comm\_split и за параметр для разделения взять ряд блока.

7) Суммируем соответствующие элементы *y\_local* в *y\_group* с помощью MPI\_Allreduce в отдельных коммуникаторах.

8) Соединяем получившиеся векторы *y\_group* с помощью MPI\_Gatherv в общем коммуникаторе MPI\_COMM\_WORLD, результат сохраняется в массиве *y*.



# 4.2. Исследование производительности

Время

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размер матрицы | 1 процесс | 2 процесса | 4 процесса | 8 процессов |
| 1000 | 0,010818 | 0,006394 | 0,006432 | 0,004907 |
| 2000 | 0,044696 | 0,024425 | 0,02279 | 0,016017 |
| 3000 | 0,100244 | 0,055203 | 0,032223 | 0,034038 |
| 4000 | 0,176689 | 0,096941 | 0,056201 | 0,055952 |
| 5000 | 0,272421 | 0,149593 | 0,086629 | 0,09798 |
| 10000 | 1,095557 | 0,596248 | 0,343263 | 0,320315 |

Ускорение

|  |  |  |  |
| --- | --- | --- | --- |
| Размер матрицы | 2 процесса | 4 процесса | 8 процессов |
| 1000 | 1,691898655 | 1,681902985 | 2,204605665 |
| 2000 | 1,829928352 | 1,961211057 | 2,790535057 |
| 3000 | 1,815915802 | 3,110945598 | 2,945061402 |
| 4000 | 1,822644701 | 3,143876443 | 3,157867458 |
| 5000 | 1,8210812 | 3,14468596 | 2,780373546 |
| 10000 | 1,837418323 | 3,191596531 | 3,420248818 |

Эффективность

|  |  |  |  |
| --- | --- | --- | --- |
| Размер матрицы | 2 процесса | 4 процесса | 8 процессов |
| 1000 | 0,845949327 | 0,420475746 | 0,275575708 |
| 2000 | 0,914964176 | 0,490302764 | 0,348816882 |
| 3000 | 0,907957901 | 0,777736399 | 0,368132675 |
| 4000 | 0,911322351 | 0,785969111 | 0,394733432 |
| 5000 | 0,9105406 | 0,78617149 | 0,347546693 |
| 10000 | 0,918709161 | 0,797899133 | 0,427531102 |

Вывод: в половине случаев ускорение продолжило расти даже на 8 процессах и достигло своего пика в 3.4 при размерах матрицы 10000\*10000. На больших данных этот алгоритм показал себя лучше всего, даже эффективность падала с меньшей скоростью.

Все рассмотренные алгоритмы показали ускорение при росте числа процессов. На малых данных все 3 алгоритма показали себя примерно одинаково, но на больших данных лучше всего себя показали именно построчный и блочный алгоритмы. Можно предположить, что это как раз связано с кэшем и с тем, что при умножении по столбцам, мы вынуждены переписывать *y\_local* длины M, а не *M/comm\_sz*.