Московский государственный университет имени М. В. Ломоносова

Факультет вычислительной математики и кибернетики

Компьютерный практикум по учебному курсу

Распределенные системы

Задания

Алгоритм **MPI_Scan** для транспьютерной матрицы

Доработка MPI-программы, реализованной в рамках курса
"Суперкомпьютеры и параллельная обработка данных"

Отчет

о выполненном задании

студента 428 группы факультета ВМК МГУ
Макеева Виталия Олеговича

Москва

2020

Содержание

1. Постановка задачи

В рамках данного задания требуется:

- Реализовать программу, моделирующую выполнение операции MPI_Scan для транспьютерной матрицы размером 4x4 при помощи пересылок MPI типа точка-точка. Оценить сколько времени потребуется для выполнения операции MPI_Scan, если все процессы выдали эту операцию редукции одновременно. Считать, что время старта равно 100, время передачи байта равно 1 (Ts=100,Tb=1). Процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.;
- Доработать MPI-программу, реализованную в рамках курса "Суперкомпьютеры и параллельная обработка данных". Добавить контрольные точки для продолжения работы программы в случае сбоя. Реализовать один из 3-х сценариев работы после сбоя: а) продолжить работу программы только на "исправных"процессах; б) вместо процессов, вышедших из строя, создать новые MPI-процессы, которые необходимо использовать для продолжения расчетов; в) при запуске программы на счет сразу запустить некоторое дополнительное количество MPI-процессов, которые использовать в случае сбоя.

2. Обзор файлов проекта

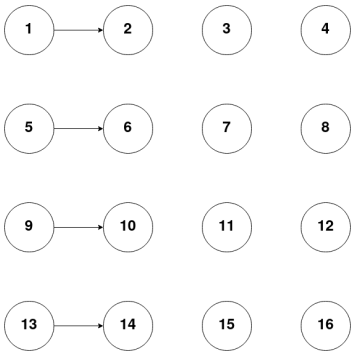
Код с реализацией обоих заданий доступен на github. В репозитории так же описана структура файлов и доступна инструкция по сборке и запуску.

3. Реализация операции MPI_Scan

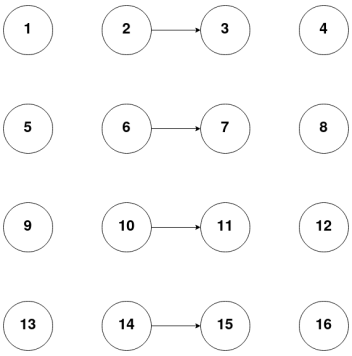
Наивный алгоритм

В каждом узле транспьютерной матрицы размера 4x4 запущен один MPI-процесс. Каждый процесс имеет свое число. В результате выполнения операции MPI_Scan каждый i-ый процесс должен получить сумму чисел, которые находятся у процессов с номерами 0, ... i включительно.

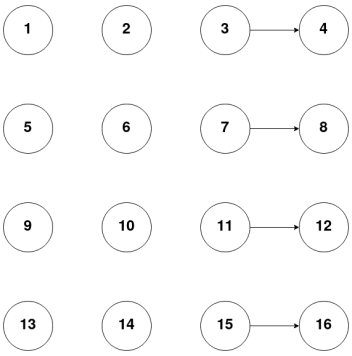
Наивный алгоритм можно визуально описать следующей схемой:



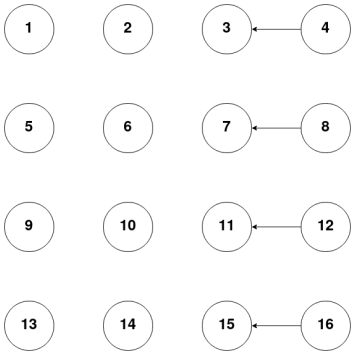
шаг 1



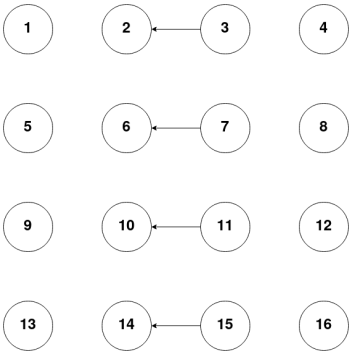
шаг 2



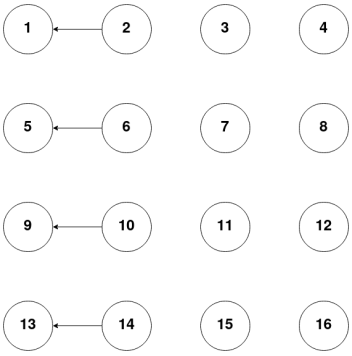
шаг 3



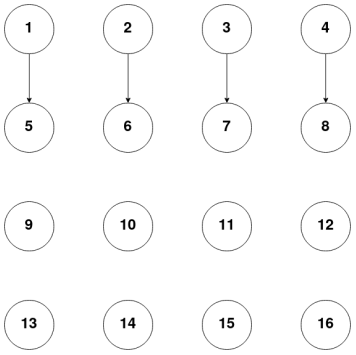
шаг 4



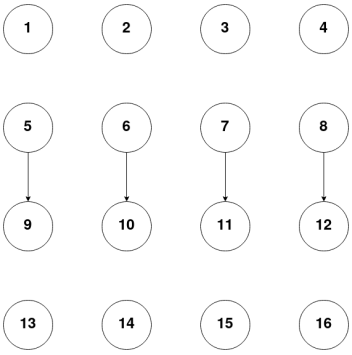
шаг 5



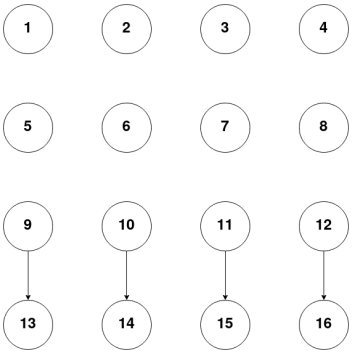
шаг 6



шаг 7



шаг 8



шаг 9

Пусть каждый процесс с номером i должен вычислить два числа: sum_to_i – сумма чисел на всех процессах от начала строки до i; sum_row – сумма чи-

сел на всех процессах в строке. Для процессов 1, 5, 9, 13 sum_to_i известна и равна соответствующим числам на этих процессах. Эти вычисления можно проводить независимо для каждой строки матрицы. Алгоритм действует следующим образом:

- На шагах 1-3 процессы построчно передают sum_to_i процессам справа. При получении соседи увеличивают свое значение sum_to_i на полученное число;
- Полагаем на процессах 4, 8, 12, 16 $sum\_row = sum\_to\_i$ и на шагах 4-6 происходит передача sum_row назад по очереди всем процессам в строке;
- Для процессов из первой строки ответом будет число sum_to_i. На шагах 7-9 каждый процесс передает sum_row нижним процессам. При получении соседи вычисляют ответ, равный $upper\_sum\_row + sum\_to\_i$, и увеличивают свой sum_row, который затем передают ниже.

Данный алгоритм для матрицы размера MxN можно оценить следующим образом (учитывая, что все передаваемые числа имеют размер B):

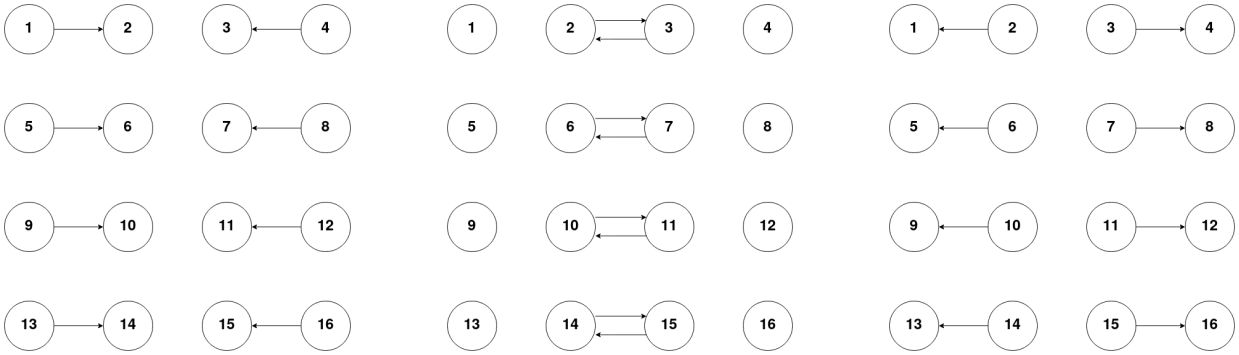
$$T_{naive}(N, M) = (2{\cdot}(N-1) + M - 1) * (T_s + B{\cdot}T_b)$$

Или же для транспьютерной матрицы размера 4x4 и $B = 4$ байта:

$$T_{naive}(4, 4) = (2{\cdot}3 + 4 - 1) * (100 + 4{\cdot}1) = 936$$

Улучшенный алгоритм

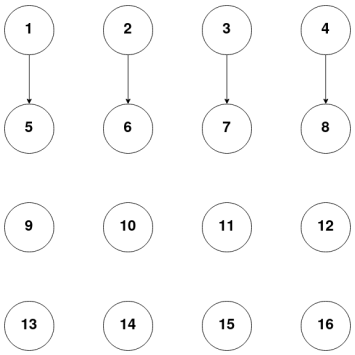
Очевидно, что описанный выше подход не оптимален. Его можно улучшить, проводя построчные вычисления на шагах 1-6 параллельно. Например, собирать sum_row в "серединных" элементах строки, а затем рассылать обратно к крайним процессам. Этот процесс схематично изображен ниже:



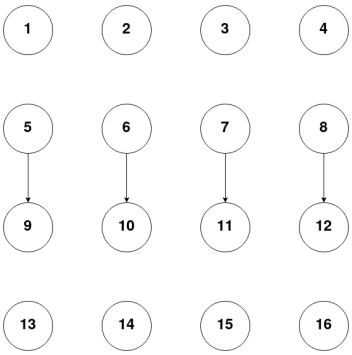
шаг 1

шаг 2

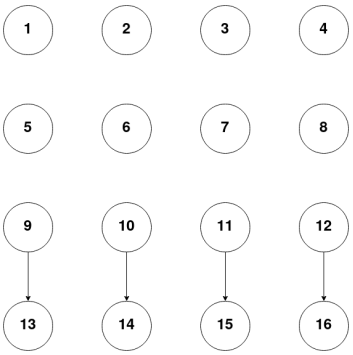
шаг 3



шаг 4



шаг 5



шаг 6

На шаге 1 происходит подсчет частичных сумм чисел на процессах. Для процессов из левой половины матрицы аналогично наивному алгоритму вычисляется sum_to_i. Для процессов из правой половины вычисляются такие же суммы, но справа налево. Шаг 2 необходим только при четном N. На нем два центральных процесса параллельно обмениваются полученными суммами и вычисляют sum_row. В случае нечетного N числа собираются на одном процессе и нужды в таком обмене нет. На шаге 3 происходит рассылка sum_row остальным процессам в строке. На процессах из правой половины $sum\_to\_i = sum\_row - sum\_to\_i$. Шаги 4-6 повторяют наивный алгоритм.

Оценим модифицированную версию алгоритма для матрицы размера NxM. Для каждой строки будет выполнено $\lfloor \frac{N-1}{2} \rfloor$ пересылок при сборе чисел к центральным элементам и еще столько же при пересылке от центральных элементов к краям. Если N четное, то дополнительно будет две пересылки между

центральными элементами. Для распространения сумм сверху вниз будет выполнено $M-1$ пересылка (аналогично наивному алгоритму). Таким образом:

$$T_{improved}(N,M) = (M - 1 + 2{\cdot}(\lfloor\tfrac{N-1}{2}\rfloor + (1 - N\%2))) * (T_s + B{\cdot}T_b)$$

где $N\%2$ может быть вычислено как $N - 2{\cdot}\lfloor\frac{N}{2}\rfloor$.

Или для транспьютерной матрицы размера 4x4 и $B = 4$ байта:

$$T_{improved}(4,4) = (3 + 2{\cdot}(\lfloor\tfrac{3}{2}\rfloor + (1 - 0)) * (100 + 4) = 728 < 936 = T_{naive}(4,4)$$

4. Доработка программы, реализованной в рамках курса СКиПОД

MPI-программа, которую необходимо было улучшить, вычисляет

$$C = \alpha * A * B + \beta * C,$$

где A, B, C – матрицы размера $ni * nj, ni * nk, nk * nj$ соответственно.

Программа была распараллелена следующим образом: пусть имеется N процессов, тогда можно разбить ni на примерно одинаковые части и для каждого процесса вычислить индексы i_{min} и i_{max}, в пределах которых будут производиться вычисления. В конце работы у каждого процесса получен результат для строк $[i_{min}; i_{max})$ матрицы C, и этот результат нигде не собирается.

В момент работы каждый процесс вычисляет 3 вложенных цикла (по i, затем по j, затем по k). В качестве контрольных точек были выбраны моменты в конце кажой итерации самого внешнего цикла. Сами контрольные точки реализованы следующим образом: во время вычислений у каждого процесса есть открытый дескриптор файла, имя которого может быть вычислено по номеру процесса. В начале работы процесс записывает туда вычисленные индексы i_{min} и i_{max}. Затем после каждой итерации цикла записывает вычисленную строку матрицы C. После каждой записи файл сливается на диск, чтобы предотвратить потерю изменений в случае отказа процесса. Для восстановления необходимо считать из файла индексы и уже вычисленные строки, и продолжить выисления с нужной строки.

При запуске программы на счет несколько процессов резервируются для использования в случае сбоя. Числом этих процессов можно управлять с помощью отдельного аргумента командной строки. Кроме того, один процесс всегда является главным и следит за состоянием других процессов, ничего не вычисляя.

Всего в программе есть 3 вида процессов, ниже подробнее описан каждый из них:

- **рабочий процесс** в начале работы инициализирует матрицы числамии и вычисляет индексы, с которыми ему предстоит работать. Затем создает файл для сброса данных и начинает вычисления, сохраняя результат на каждой итерации. В конце работы процесс отправляет мастеру неблокирующий *send*, который уведомляет о том, что процесс завершается;

- **резервный процесс** в начале работы также инициализирует матрицы, но затем блокируется до тех пор, пока мастер не пришлет процессу число. Это число либо является номером процесса, который вышел из строя, либо -1 (которая означает, что все в порядке и можно завершать работу). В случае положительного числа резервный процесс вычисляет имя файла, из которого нужно считать данные, считывает их и дальше работает как рабочий процесс, т. е. производит вычисления и отправляет мастеру *send*;

- **главный процесс (мастер)** поддерживает два списка: процессов, которые в данный момент являются рабочими и резервных процессов, которые еще не были использованы. С каждым процессом из первого списка ассоциирована структура $MPI\_Request$, которая необходима для ожидания процессов. В начале работы мастер с помощью функции $Irecv$ начинает ожидать от каждого из рабочих процессов уведомления о завершении. Затем он входит в бесконечный цикл, на каждой итерации которого он блокируется, начиная ожидать выполнения $Irecv$ от какого-либо процесса с помощью функции $Waitany$ (которая дополнительно возвращает индекс того процесса в массиве, которого удалось дождаться). При разблокировке возможны 3 случая:

  1. Был получен положительный индекс и статус соответствующего элемента имеет значение $SUCCESS$. Это является уведомлением об успешном завершении рабочего процесса с соответствующим индексом. Мастер помечает процесс как завершенный и больше не ожидает его;
  2. Была получена какая либо ошибка, при этом индекс в массиве отрицательный. Это означает, что мастер сразу вышел из $Waitany$, потому что все процессы успешно завершились, и можно прервать цикл;
  3. Был получен положительный индекс и статус соответствующего элемента сигнализирует об ошибке. Это означает, что в процессе с соответствующим индексом произошел сбой. Мастер берет любой резервный процесс, удаляет его из списка резервных процессов. Затем в списке рабочих процессов заменяет неисправный процесс на резервный и отправляет резервному процессу его индекс в этом списке. Затем мастер с помощью неблокирующей функции $Irecv$ начинает ожидать от этого процесса сигнала о завершении.

После завершения цикла главный процесс отправляет -1 все резервным процессам, которые еще не были использованы и завершается сам.

Пример описанного процесса изображен на снимке экрана ниже. После первой итерации отказывает процесс 1, и работу продолжает процесс 5. Он также отказывает после первой итерации и работу заканчивает процесс 6.

```
 $ mpirun -np 7 mult 10 20 30 2
There are 7 processes total
        Proc #0-#3 are workes
        Proc #4 is master
        Proc #5-#6 are backup
i = 10, j = 20, k = 30

|       |       #2: working in range [6; 8)
|       |       |       |       #4: received status 0 and index 2 from wait
|       |       |       |       |       |       #6: waiting as backup
|       |       #2: finished
#0: working in range [0; 3)
|       |       |       #3: working in range [8; 10)
|       |       |       #3: finished
|       |       |       |       |       #5: waiting as backup
|       |       |       |       #4: received status 0 and index 3 from wait
|       |       |       |       #4: received status 0 and index 0 from wait
|       #1: working in range [3; 6)
#0: finished
|       #1: failed on iteration 1!
|       |       |       |       #4: received status 74 and index 1 from wait
|       |       |       |       #4: found issues in process #1
|       |       |       |       #4: sending backup data to process #5
|       |       |       |       |       #5: received 1
|       |       |       |       |       #5: opening file 1_res
|       |       |       |       |       #5: read 1 rows, calculating rows from 4 to 5
|       |       |       |       |       #5: failed on iteration 1!
|       |       |       |       #4: received status 74 and index 1 from wait
|       |       |       |       #4: found issues in process #5
|       |       |       |       #4: sending backup data to process #6
|       |       |       |       #4: received status 0 and index 1 from wait
|       |       |       |       |       |       #6: received 1
|       |       |       |       |       |       #6: opening file 1_res
|       |       |       |       |       |       #6: read 2 rows, calculating rows from 5 to 5
|       |       |       |       |       |       #6: finished
|       |       |       |       #4: received status 0 and index -32766 from wait
|       |       |       |       #4: master finishing
```

Главным минусом такого подхода является незащищенность программы от сбоев в мастере. Это можно решить, например, реплицируя главный процесс и запуская несколько его копий сразу. Кроме того, программа устойчива лишь к конечному числу сбоев, так как число резервных процессов задается заранее при запуске. Для решения этой проблемы можно после успешного завершения рабочего процесса добавлять его в список резервных. Но этот подход не оптимальный, поскольку может произойти сбой во всех процессах сразу и выполнять вычисления будет некому.