

## 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

# Содержание

1. Постановка задачи	2
2. Обзор файлов проекта	2
3. Реализация операции MPI_Scan	3
Наивный алгоритм . . . . .	3
Улучшенный алгоритм . . . . .	5
4. Доработка программы, реализованной в рамках курса СКИ-ПОД	7

# 1. Постановка задачи

В рамках данного задания требуется:

- Реализовать программу, моделирующую выполнение операции `MPI_Scan` для транспьютерной матрицы размером  $4 \times 4$  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 ( $T_s=100, T_b=1$ ). 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

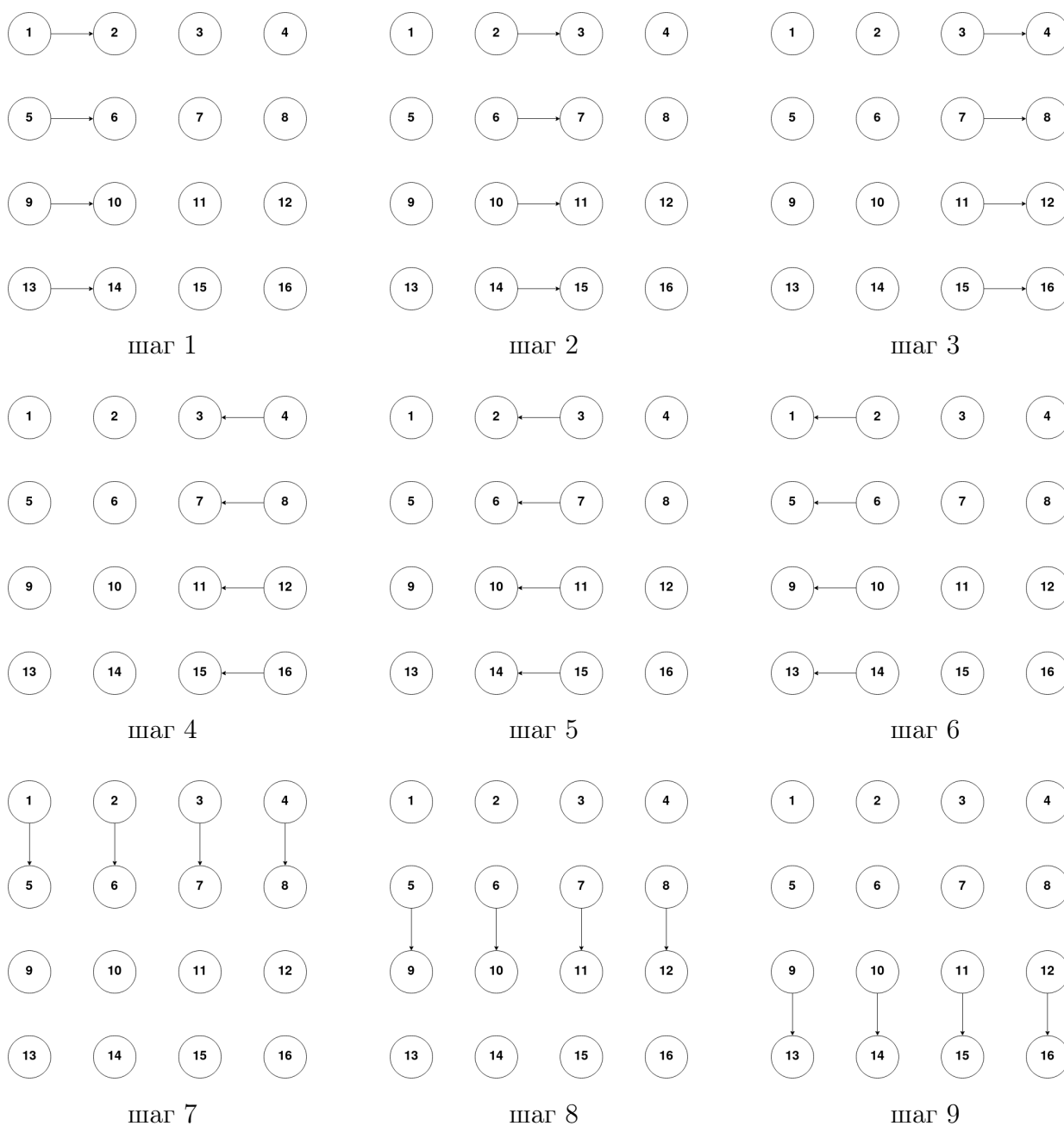
Код с реализацией обоих заданий доступен на [github](#). В репозитории также описана структура файлов и доступна инструкция по сборке и запуску.

### 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:



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  $M \times N$  можно оценить следующим образом (учитывая, что все передаваемые числа имеют размер  $B$ ):

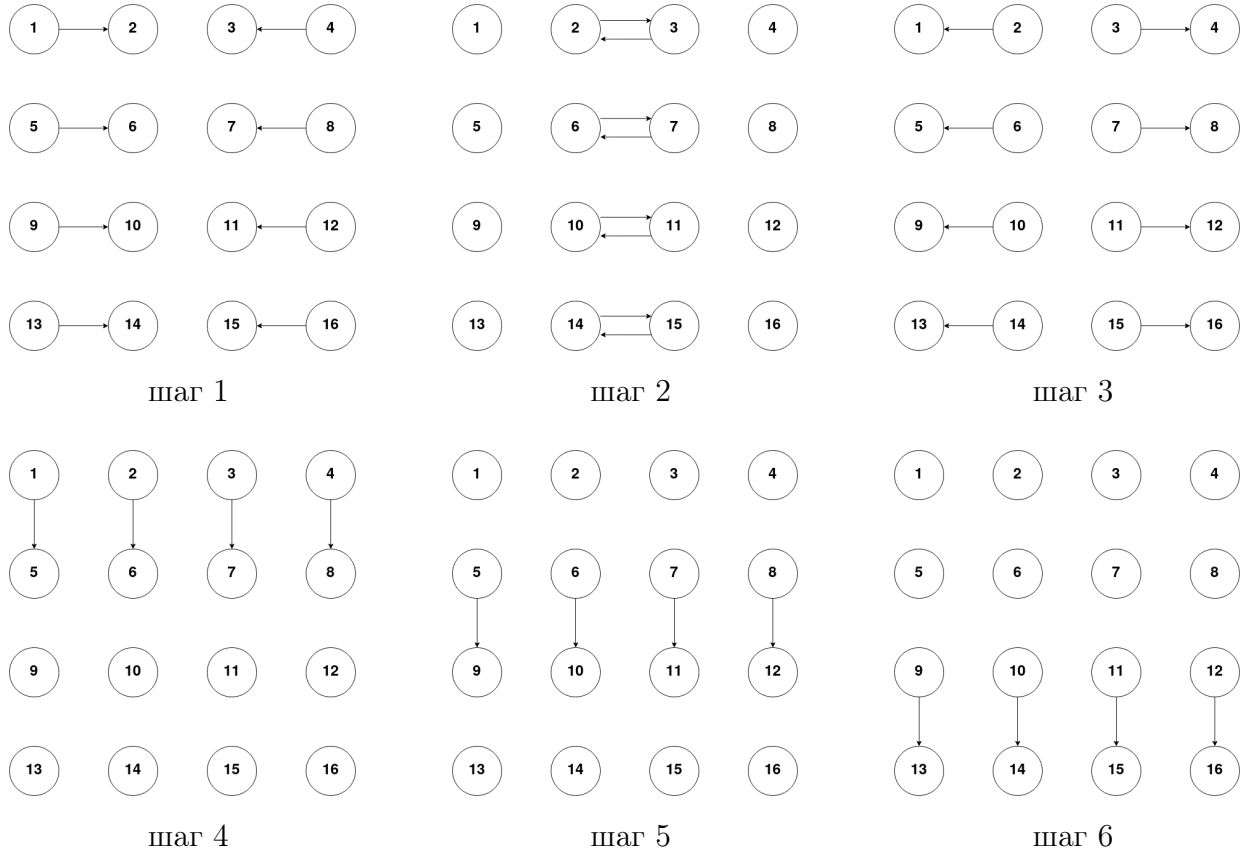
$$T_{naive}(N, M) = (2 \cdot (N - 1) + M - 1) * (T_s + B \cdot T_b)$$

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:



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  $N \times M$ . 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 (с учетом того, что пересылки выполняются параллельно для пра-

вой и левой частей). Если  $N$  четное, то дополнительно будет пересылка между центральными элементами. Для распространения сумм сверху вниз будет выполнена  $M - 1$  пересылка (аналогично наивному алгоритму). Таким образом:

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

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:

$$T_{improved}(4, 4) = (3 + 2 \cdot \lfloor \frac{3}{2} \rfloor + 1) * (100 + 4) = 624 < 936 = T_{naive}(4, 4)$$

## 4. Доработка программы, реализованной в рамках курса СКИПОД

МРІ-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 каждой 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 вычисления 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 числами 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*. Если открыть файл не удалось, процесс создает его и сам вычисляет нужные для работы индексы;
- **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 дожждаться). 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 заверченный 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

