Московский Государственный Универститет им. М. В. Ломоносова

Факультет Вычислительной Математики и Кибернетики

Практикум по курсу Распределенные системы

Реализация MPI_ALLREDUCE

Реализация устойчивой версии программы из курса "Суперкомпьютеры и параллельная обработка данных"

Выполнил:

[redacted]

Москва, 2024

Содержание

1. Часть 1. Постановка задачи

1.1. Условие задачи:

В транспьютерной матрице размером 5×5, в каждом узле которой находится один процесс, необходимо выполнить операцию нахождения максимума среди 25 чисел (каждый процесс имеет свое число). Найденное максимальное значение должно быть получено на каждом процессе.
Реализовать программу, моделирующую выполнение операции MPI_ALLREDUCE на транспьютерной матрице при помощи пересылок MPI типа точка-точка.
Оценить сколько времени потребуется для выполнения операции MPI_ALLREDUCE, если все процессы выдали эту операцию редукции одновременно. 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.

1.2. Формулировка задания:

Доработать MPI-программу, реализованную в рамках курса “Суперкомпьютеры и параллельная обработка данных”. Используя параллельный ввод-вывод (MPI-IO), 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. Часть 2. Описание алгоритма

2.1. MPI_ALLREDUCE

Инициализируем транспьютерную матрицу случайным образом с помощью функции `init_values()`, для удобства чтения матрицы элементы инициализируются числами от 0 до 99. Печатаем матрицу для наглядности, чтобы можно было проверить, что найденный максимум действительно является максмумом этой матрицы.

Каждый процесс обрабатывает один элемент матрицы, соответствующий его рангу. Процесс с рангом i обрабатывает элемент $values[i]$. Каждый процесс получает свое число в результате вызова функции `reduce()`, которая и реализует основные шаги алгоритма. Начальные данные выглядят следующим образом:

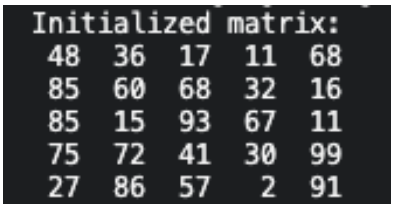

Рис. 1: Пример одной из возможных матриц

Основная идея алгоритма: Собираем максимальное значение в "центре" матрицы, а затем из центра рассылаем остальным процессам найденное максимальное значение.

1. Каждый процесс вычисляет свою "координату".

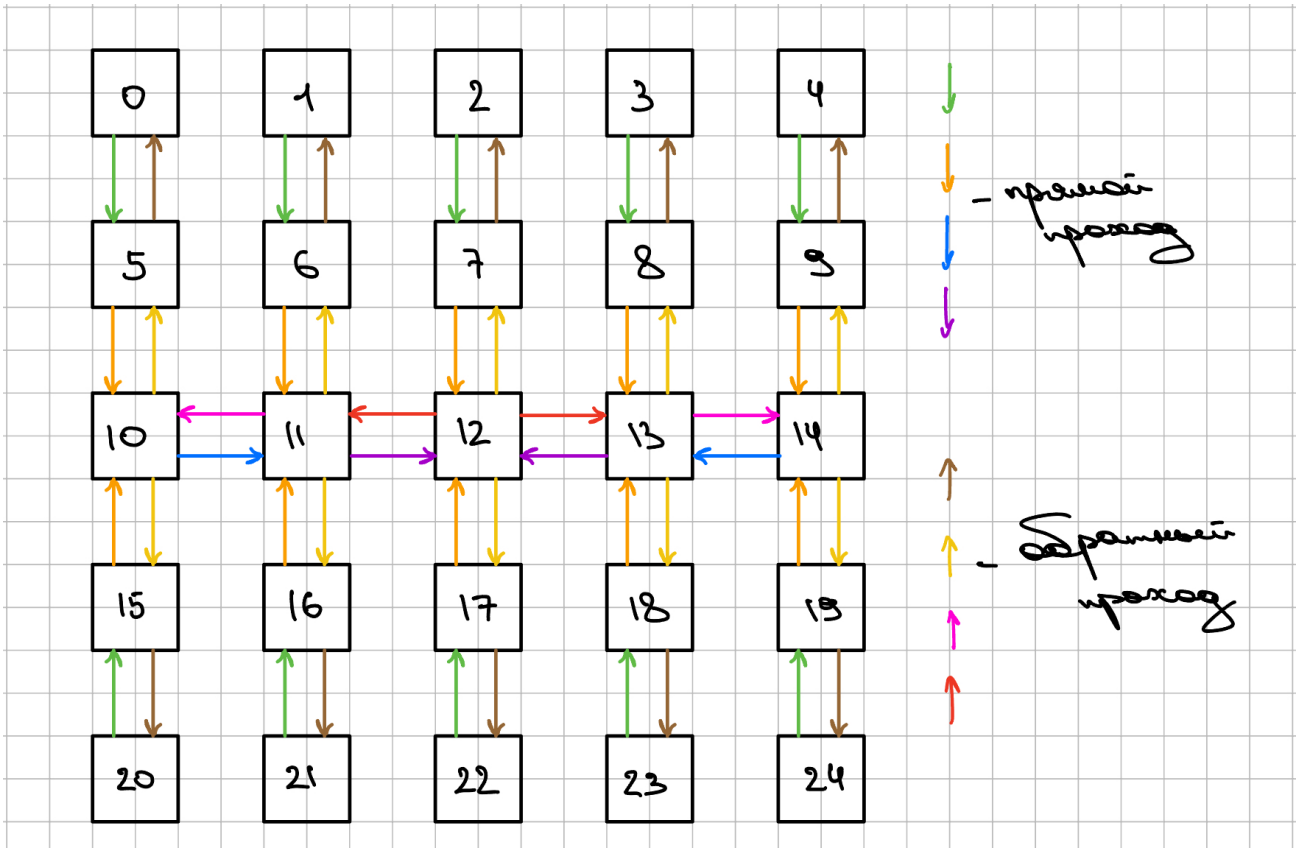

Рис. 2: Иллюстрация того, как общаются процессы

2. Каждый процесс вычисляет от скольких соседних процессов он должен получить число с помощью MPI_Recv.

3. Процесс по очереди получает числа от соседних процессов и вычисляет максимум из числа хранящегося в этом процессе и вновь полученного и обновляет локальный максимум.

4. Процесс определяет направление, в котором он отправляет свое текущее значение. Если он находится на центральной строке, то данные передаются влево или вправо в зависимости от расположение относительно центра. Если процесс находится не в центральной строке, то данные передаются вверх или вниз.

5. Передача осуществляется с использованием MPI_Isend.

6. Процесс, не находящийся в центральной точке, ожидает новое значение от своего соседа. Процесс являющийся "центром"матрицы вычисляет максимум по всей матрице и начинает обратную рассылку.

7. Обратная рассылка осуществляется по схожей логике, но в обратном направлении.

8. При обратной рассылке каждый процесс сравнивает полученное максимальное значение с эталонным значением max, которе было вычислено изначально. Если результат не совпадает, то выводится сообщение об ошибке.

9. Каждый процесс выводит сообщение с найденным максимальным значением и завершается через MPI_Finalize.

2.2. Устойчивая программа

Программа выполняет параллельное вычисление методом релаксации в 3D-сетке, распределенной по процессам. Основная цель — поддерживать отказоустойчивость: при сбое процесса программа сохраняет свое состояние (чекпоинт) и восстанавливается, чтобы продолжить выполнение с момента последней сохраненной итерации на оставшихся процессах.

Устанавливаем пользовательский обработчик ошибок через `MPI_Comm_create_errhandler` и `MPI_Comm_set_errhandler`. Итерации сбоя `failure_iteration` задается через аргумент командной строки. Если пользователь ввел некорректное значение (≤ 0 или > 100), используется значение по умолчанию 23. Затем каждый процесс получает свою часть тензора, размер данной части вычисляется исходя из количества процессов с помощью функции `calculate_bounds(int rank, int size, int startrow, int`

nrow), за назначение части тензора отвечает функция initialize(). Логика этих функций взята из прошлогодней программы. Далее в цикле запускаем метод релаксации (relax()), реализация также из прошлогодней программы, который остановится либо в момент, когда метод сойдется, либо в момент, когда будет достигнуто предельное значение итераций.

В цикле на определенной итерации (определяется пользователем) будет вызвана функция simulate_failure(), которая выполнится один раз и вызовет raise(SIGKILL) для всех процессов с четным номером. Эта функция спровоцирует вызов обработчика ошибок error_handler(), который сделан по аналогии с рассмотренным на лекции. В обработчике ошибок с помощью MPIX_Comm_revoke(*pcomm) отзывается старый коммуникатор и создается новый с помощью MPIX_Comm_shrink(*pcomm, communicator). Поскольку количество процессов уменьшилось, то необходимо пересчитать границы для процессов, так как оставшиеся процессы должны перераспределить между собой тензор. Также проверяется наличие контрольной точки, если контрольная точка существует, то выполняется восстановление из нее через load_checkpoint(), иначе инициализируем матрицу начальными значениями и начинаем вычисления заново.

Контрольные точки создаются в том же цикле, в котором вызывается функция relax(), они создаются каждые 10 итераций с помощью функции save_checkpoint(). Функция save_checkpoint() открывает на чтение файл FILE_NAME, если такого файла нет, то он создается. Процесс с рангом 0 записывает в начало файла текущий номер итерации, чтобы после сбоя восстановить точку начала. Каждый процесс записывает свою локальную часть тензора. Чтобы каждый процесс записывал данные в определенную область файла высчитывается смещение для каждого процесса. Также в функции save_checkpoint() обновляется глобальный флаг checkpoint_exists, кото-

рый нужен, чтобы проверять, была ли в ходе программы создана хотя бы одна контрольная точка. Функция `load_checkpoint()` работает по схожей логике, но только в ней данные считываются процессами из файла.

После цикла вызывается функция `verify()`, которая вычисляет суммарное значение S по всему тензору. В конце работы программы удаляется файл, хранящий контрольную точку, это сделано для того, чтобы при повторном запуске не было ситуации, когда файл с контрольной точкой существует, до первого сохранения в данном запуске, со старыми данными.

3. Часть 3. Временная оценка

3.1. MPI_ALLREDUCE

Алгоритм выполняется в 8 шагов, на каждом шаге пересылаем найденный максимум, то есть время работы алгоритма составляет $8 \cdot (T_s + 4 \cdot T_b) = 8 \cdot (100 + 4 \cdot 1) = 832$ единицы времени.

4. Часть 4. Запуск программы

4.1. MPI_ALLREDUCE

Для компиляции и запуска программы использовался OpenMPI 5.0.6 .

Компиляции производилась следующий образом: `mpicc MPI_ALLREDUCE.c -O2 -Wextra -Wall -o allreduce`

Запуск: `mpirun -np 25 -oversubscribe allreduce`

4.2. Устойчивая программа

Для компиляции и запуска программы использовался OpenMPI 5.0.6 .

Компиляции производилась следующий образом: `mpic++ -O2 mpi2.cpp -o mpi_programm`

Запуск: `mpirun -n 4 -with-ft=ulfm mpi_programm <failure_iteration>`

5. Часть 5. Исходный код

Коды программ приложены к файлу с отчетом.