Московский государственный университет имени М.В.Ломоносова
Факультет вычислительной математики и кибернетики

Отчёт по учебному курсу «Распределенные системы»

Автор: Богатенкова Анастасия Олеговна
гр. 428

Москва, 2020

Содержание

1 Постановка задачи

Требуется сделать следующее:

- Реализовать операцию *MPI_Reduce* (нахождение максимума) на транспьютерной матрице при помощи пересылок MPI типа точка-точка.
- Доработать MPI-программу, реализованную в рамках курса "Суперкомпьютеры и параллельная обработка данных". Добавить контрольные точки для продолжения работы программы в случае сбоя.

Для продолжения работы программы после сбоя использована следующая стратегия: при запуске программы на счет сразу запускается некоторое дополнительное количество MPI-процессов, которые используются в случае сбоя.

После реализации операции *MPI_Reduce* необходимо оценить сколько времени потребуется для её выполнения, 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.

2 Реализация операции MPI_Reduce и оценка её сложности

В транспьютерной матрице размером 4*4, в каждом узле которой находится один процесс, необходимо выполнить операцию нахождения максимума среди 16 чисел (каждый процесс имеет свое число). Найденное максимальное значение должно быть получено на процессе с координатами (0,0).

Минимальное время оценивается через минимальное расстояние между двумя самыми дальними процессами в матрице. В нашем случае, чтобы пройти от процесса с координатами (0, 0) к процессу с координатами (3, 3), необходимо сделать 6 шагов. Это количество шагов является минимальным, так как есть алгоритм, реализующий операцию нахождения максимума за 6 шагов. Один из способов пересылки (который был реализован) показан на рисунке 1:

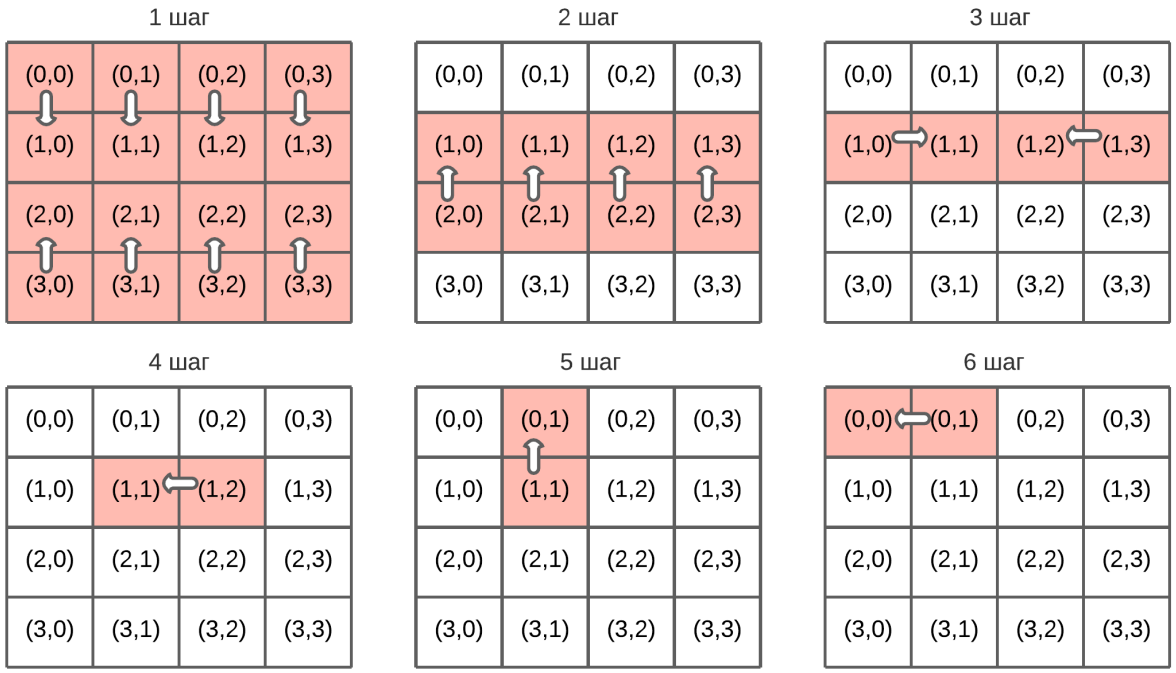

Рис. 1: Алгоритм нахождения максимума на транспьютерной матрице

Данный алгоритм был реализован с помощью функций *MPI_Send* и *MPI_Recv*. По-

лучение топологии в виде транспьютерной матрицы произведено с помощью функции *MPI_ Cart_ rank*.

Оценим время работы алгоритма. Если время старта равно 100, время передачи байта равно 1 (Ts=100,Tb=1), то время выполнения операции рассчитывается следующим образом:

$$time = num\_steps \cdot (Ts + n \cdot Tb)$$

где n - размер передаваемого сообщения в байтах. В нашем случае сообщением является число, размер которого может быть равен, например, 4 байтам.

Таким образом, при $n = 4$, получаем:

$$time = 6 \cdot (100 + 4 \cdot 1) = 624$$

3 Добавление в программу возможности её продолжения в случае сбоя

Для того, чтобы при сбое одного из процессов программа не завершалась с ошибкой, а продолжала своё выполнение, необходимо написать обработчик ошибок, который будет срабатывать в таких ситуациях. Для этого в стандарте MPI существует специальные функции *MPI_ Comm_ create_ errhandler* и *MPI_ Comm_ set_ errhandler*. Однако стандарт не позволяет определить, в каком именно процессе произошла ошибка. Это можно сделать, используя расширение MPI – ULFM [1].

Программа была реализована в учебных целях, поэтому запускать её, используя ULFM, рекомендуется через Docker. Однако, можно установить себе расширение, следуя инструкции на сайте [1].

Программа была доработана следующим образом:

1. с помощью фунцкций *MPI_ Comm_ create_ errhandler* и *MPI_ Comm_ set_ errhandler* добавлен обработчик ошибок *err_ handler*, о реализации которого будет сказано ниже;

2. в качестве резервного процесса используется последний процесс;

3. для каждого работающего процесса сформировано имя файла для записи данных контрольных точек;

4. в начале, в соответствии с выбранным алгоритмом сортировки, данные распределяются по процессам с помощью коллективной операции *MPI_ Scatterv*, при этом последнему процессу данные не отправляются;

5. далее следует $tasks - 1$ итераций цикла ($tasks$ - число процессов), на каждой итерации соседние процессы обмениваются данными друг с другом. В начале каждой итерации процессы считывают данные из файла, работают с ними и в конце снова записывают в файл. Если произошла какая-то ошибка (это выясняется с помощью специального флага), то все процессы считывают данные заново и итерация начинается сначала.

6. Для того, чтобы процессы находились на итерациях с одинаковым номером, были расставлены «барьеры» с помощью функции *MPI_ Barrier*.

7. В коде один из процессов убивается – после этого во всех процессах управление переходит в функцию-обработчик ошибок.

8. В обработчике ошибок на базе старого коммуникатора создается новый, не включающий в себя вышедшие из строя процессы (в нашем случае один процесс). Это делается с помощью функции, не входящей в стандарт MPI, *MPIX_ Comm_ shrink*.

9. После создания нового коммуникатора, каждый процесс получает новый номер и возможно работает с другим файлом, однако это не влияет на результат работы программы.

10. Работа программы продолжается на оставшихся процессах и результат собирается на процессе с номером 0 с помощью функции *MPI_ Gather*.

Заключение

Таким образом, была реализована операция *MPI_Reduce* на транспьютерной матрице при помощи пересылок MPI типа точка-точка и оценено время ее работы. MPI-программа, реализующая алгоритм чет-нечетной сортировки, была доработана так, чтобы работа программы продолжалась после выхода из строя одного из процессов. Для этого один из процессов изначально считается резервным и используется в случае необходимости.

Код двух программ с инструкцией по запуску доступен на сайте [2].

Список литературы

[1] User Level Failure Mitigation. http://fault-tolerance.org/.

[2] Github repository with code. https://github.com/NastyBoget/msu_parallel_programming/tree/main/7_semester.