Отчёт по курсу «Распределённые системы»

Коротков Борис Сергеевич, 424 группа

**Задача 1.**

1. Постановка задачи

В транспьютерной матрице размером 4\*4, в каждом узле которой находится один процесс, необходимо выполнить операцию рассылки данных (длиной 4 байта) всем процессам от одного (MPI\_SCATTER) - процесса с координатами (0,0).  
Реализовать программу, моделирующую выполнение операции MPI\_SCATTER на транспьютерной матрице при помощи пересылок MPI типа точка-точка с использованием синхронного режима.  
Получить временную оценку работы алгоритма. Оценить сколько времени потребуется для выполнения операции MPI\_SCATTER, если все процессы выдали ее 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.

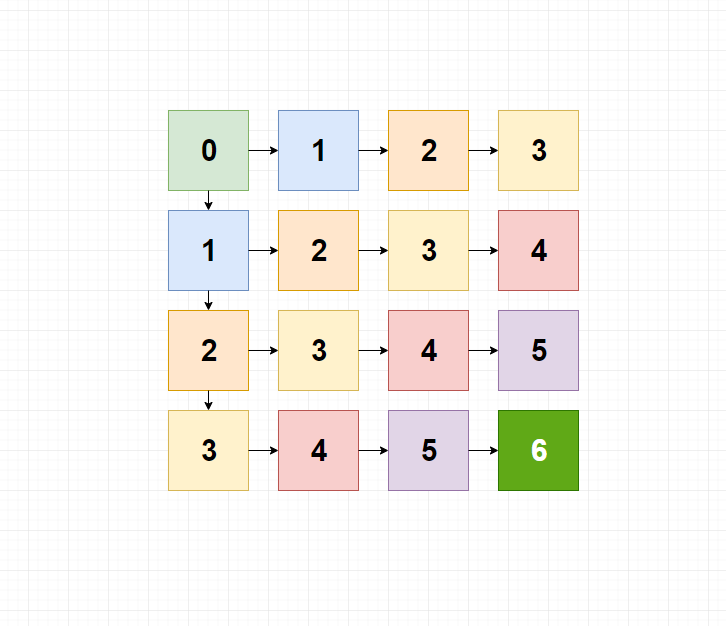
1. Решение

Пусть не предусмотрено буферизации, сообщение не разбивается на равные части при пересылке, а отправляется целиком. При выполнении операции MPI\_Scatter, процесс (в нашем случае (0,0)) отсылает сообщение всем узлам транспьютерной матрицы. Чтобы снизить временные затраты, будем передавать сообщение по 0-ой колонке матрицы. Получивший сообщение узел, находящийся в этой колонке, передаёт его по горизонтали до последнего звена. Таким образом требуется P-1 такт передачи по вертикали и Q-1 такт передачи по горизонтали, что на самом длинном пути приводит к P+Q-2 тактам передачи.

Один такт посылки сообщения состоит из инициализации (Ts) и передачи сообщения из M байт (Tb\*M). Получаем (P+Q-2)\*(Ts+M\*Tb). P,Q – размеры транспьютерной матрицы, M = 4.

Общее время = 6\*(Ts+M\*Tb) = 624

Получаем такое распределение тактов:



1. Замечание

На скриншоте Task1(1) мы можем увидеть, как последовательно передаётся сообщение по узлам транспьютерной матрицы (при помощи обычного MPI\_Send). На скриншоте Task1(2) мы видим работу нашей программы, которая в синхронном режиме (при помощи MPI\_Ssend) передаёт сообщение. Порядок печати нарушен, потому что крайние и угловые узлы тратят меньше времени на приём и передачу сообщений, чем те, что находятся в центре или в 0-ой колонке. Из-за этого печать их результата появляется быстрее.

**Задача 2.**

1. Постановка задачи

Все 25 процессов, находящихся на разных ЭВМ сети, одновременно выдали запрос на вход в критическую секцию. Реализовать программу, использующую древовидный маркерный алгоритм для прохождения всеми процессами критических секций.  
Критическая секция:  
<проверка наличия файла “critical.txt”>;  
if (<файл “critical.txt” существует>) {  
<сообщение об ошибке>;  
<завершение работы программы>;  
} else {  
<создание файла “critical.txt”>;  
sleep (<случайное время>);  
<уничтожение файла “critical.txt”>;  
}  
Для передачи маркера использовать средства MPI.  
Получить временную оценку работы алгоритма. Оценить сколько времени потребуется, если маркером владеет нулевой процесс. 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. Решение

Будем считать, что корнем двоичного дерева является 0-ой процесс, у которого находится маркер в начальный момент времени. Проведём эмуляцию одновременного входа в критическую секцию при помощи функции MPI\_Barrier. Инициализация общей памяти проводится для определения сколько ещё процессов не прошло критическую секцию. Когда все процессы прошли её, работа программы завершается, чтобы не ждать время, которое мы потратим на бесполезный обмен сообщениями между процессами.

Для оценки времени составляется сбалансированное дерево. Где бы не находился маркер, нам необходимо N-1 посылок запросов для инициализации (по каждому ребру либо в одну, либо в другую сторону). Начинаем обход с вершины, где есть маркер, и обходим его в глубину, начиная сначала с правых поддеревьев, переходя к левым. Тогда по каждому ребру придётся пройти 2 раза (вперёд и назад), кроме последней самой левой цепочки. Ещё стоит учесть M запросов после того, как маркер покидает вершину, в направлении ушедшего маркера (они выполняются, если очередь запросов в узле не пуста). Эту оценку можно улучшить, посылая запрос сразу же вместе с маркером в одном сообщении. Итого: (25 + приблизительно 46)(Ts+1\*Tb) + M\*Tb = 71\*101 + приблизительно 21 = 7171 + 21 = 7192 для 25 процессов. Если маркер в начальный момент времени будет находиться не в 0-ом процессе, то нам потребуется меньше операций, и следовательно время уменьшится.

Древовидный маркерный алгоритм:

* Вход в критическую секцию:

Если есть маркер, то процесс выполняет критическую секцию, если нет маркера, то помещает свой запрос в FIFO-очередь запросов и посылает сообщение «ЗАПРОС» в направлении владельца маркера и ждёт сообщений.

* Поведение процесса при приёме сообщений:

Процесс, не находящийся внутри критической секции, должен реагировать на сообщения двух видов – «МАРКЕР» (-1) и «ЗАПРОС» (номер узла, от 0 до N-1).

**А)** Если пришло сообщение «МАРКЕР»

М1. Взять 1-ый запрос из очереди. Если это свой запрос, то исполнить вход в критическую секцию, изъять его из очереди и вернуться к пункту М1. Иначе послать маркер его автору.

М2. Поменять значение указателя в сторону маркера. (marker\_pointer)

М3. Исключить запрос из очереди.

М4. Если в очереди остались запросы, то послать сообщение «ЗАПРОС» в сторону маркера.

**Б)** Если пришло сообщение «ЗАПРОС»

Поместить запрос в очередь. Если нет маркера, то послать сообщение «ЗАПРОС» в сторону маркера, иначе (если есть маркер) – перейти на пункт М1.

* Если очередь запросов пуста, то при выходе ничего не выполняется, иначе выполняется пункт М1.

Начало работы – запрос входа в критическую секцию (request\_permisson), ожидание маркера (wait\_marker). При получении маркера – возврат к работе, прохождение критической секции. При завершении критической секции процессы продолжают «слушать» (check\_query), отвечая на запросы от соседних процессов, пока не наступит период, за который не будет принято ни одного сообщения (проверка состояния осуществляется при помощи MPI\_Test через равные промежутки времени).

1. Замечание

На скриншотах Task2(<количество потоков>) мы можем увидеть временную оценку работы программы. Для 24 и больше потоков у меня не получилось дождаться выполнения программы, так как компьютер начинал подтормаживать. Скорее всего это связано с нехваткой мощности моего процессора (2 ядра, 4 потока), нехваткой оперативной памяти (запущено на виртуальной машине с 4 Гб. ОЗУ) или же переполнением очереди, когда процессы синхронно закидывают много запросов и не успевают отключиться (необходимо подобрать оптимальное значение count при котором нижележащие процессы будут успевать пройти критическую секцию, в то время как вышележащие будут ещё активны).