МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Компьютерный практикум по курсу:

“Распределенные системы”

Задания:

Моделирование алгоритма процессорной консистентности в DSM

Разработка отказоустойчивой версии программы

Отчет

студента 421 группы

факультета ВМК МГУ

Есенина Егора Евгеньевича

Оглавление

1 Постановка задачи

Требуется разработать и реализовать:

1. Процессорная консистентность памяти и алгоритм ее реализации в DSM с полным размножением. Сколько времени потребует модификация 10 различных переменных, если все 10 процессов (каждый процесс модифицирует одну переменную), находящихся на разных ЭВМ сети с шинной организацией (без аппаратных возможностей широковещания), одновременно выдали запрос на модификацию своей переменной. 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.
   Для разработанного алгоритма реализовать программу, осуществляющую все необходимые рассылки значений модифицируемых переменных при помощи пересылок MPI типа точка-точка.
2. 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 Структура проекта

2.1 FirstTask

Директория FirstTask содержит модель алгоритма процессорной консистентности в DSM. Для запуска модели требуется проделать следующие команды:

- docker pull abouteiller/mpi-ft-ulfm (Для корректного использования типа bool)
- ./bulid.sh — для сборки
- ./start… для запуска двух вариантов теста описанных в пункте 3.3

2.2 SecondTask

Директория SecondTask содержит реализацию отказоустойчивой программы Gauss (main.c), Makefile, а также два скрипта, реализующие сборку и запуск программы. Обязательно требуется проделать следующую команду:

- docker pull abouteiller/mpi-ft-ulfm

3 Модель алгоритма процессорной консистентности в DSM

3.1 Постановка задачи

Реализовать программу, моделирующую алгоритм процессорной консистентности в DSM для 10 процессов и 10 переменных. Каждый процесс модифицирует отличную(от других процессов) переменную.

Процессорная консистентность памяти и алгоритм ее реализации в DSM с полным размножением. Сколько времени потребует модификация 10 различных переменных, если все 10 процессов (каждый процесс модифицирует одну переменную), находящихся на разных ЭВМ сети с шинной организацией (без аппаратных возможностей широковещания), одновременно выдали запрос на модификацию своей переменной(размер переменной считаю I = 1 байт). 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.

3.2 Алгоритм процессорной консистентности

Процессорная консистентность сочетает в себе Pram консистентность дополненную когерентностью памяти на каждом из процессов.

Таким образом должны быть выполнены следующие условия:

1. Команды записи выполненные на одном процессоре должны быть видны другим процессорам в том порядке, в котором они были выполнены эти процессором(Pram консистентность)
2. Для каждой переменной(ячейки памяти) есть общее соглашение относительно порядка в котором процессы модифицируют эту переменную(Когерентность памяти).

3.3 Возможные алгоритмы реализации.

- Идеальный случай - каждый из процессов является координатором себя(в рамках PRAM консистентности), а также координатор переменной которую он изменяет в таком случае количество времени затраченное на выполнение данного алгоритма ввиду отсутствия возможности широковещания составит:

  $N * (N - 1) * (T_s + T_b * I) = 10*9*(100+1) = 9090\ mc$

  В данном случае, временные затраты будут наименьшими.

- Худший случай - каждый из процессов является координатором себя(в рамках PRAM консистентности), а координатором переменной, которую он изменяет является другой процесс(то есть номер переменной и номер процесса попарно различны(не ограничивая общности))

  В таком случае, процесс отправит сообщение об изменении переменной координатору и только затем произойдет рассылка всем(включая и сам процесс, который делал запрос к координатору) процессам в порядке очереди(если до этого процесса модификацию этой переменной

проводил другой процесс и его запрос дошел к координатору раньше). Таким образом будет достигнуто условие когерентности памяти.

В таком случае, каждый процесс отправит 1 запрос(будем считать что размер запроса также равен 1 байту), а также каждый процессор выступит в роли координатора, который отправит 9 сообщений о модификации.

В таком случае количество затраченного времени

$N * (T_s + T_b * l) \quad + N * (N - 1) * (T_s + T_b * l) = 10 * (100 + 1) + 10 * 9 * (100 + 1) =$10100 mc

4 Реализация отказоустойчивой программы Gauss

Для реализации отказоустойчивой версии был выбран метод, который создает дополнительные процессы, чтобы в случае ошибки заменить ими “испорченные” процессы и продолжить вычисления тем же количеством процессов.

Для этого рассмотрим основные функции

4.1 verbose_errhandler

Данная функция описывает реакцию процессов в случае сбоя. В результате сбоя каждый процесс должен обновить свою рабочую группу (удалить из неё мёртвый процесс с помощью MPIX_Comm_shrink) и после этого сделать loadData, чтобы иметь правильные данные на момент начала чекпоинта.

4.2 loadData

В случае сбоя, данные загружаются из файлов в память каждого процесса. Это сделано для того, чтобы никакой процесс не имел на каком-то шаге наполовину обновленную матрицу. Матрицы должны быть сброшены к предыдущему шагу. После загрузки процессы ждут друг друга с помощью MPI_Barrier.

4.3 saveData

Данная функция отвечает за запись данных в файлы. Так как все процессы имеют одинаковые данные, то записью занимается процесс с рангом 0. Остальные же процессы ждут завершения процесса записи с помощью MPI_Barrier.

4.4 Организация чекпоинта

- saveData();
- ...
- flag = true; //флаг для входа в цикл
- while(flag || err_fl){
    - err_fl = false;
    - flag = false;
    - ...
    - if(err_fl == false){
        - MPI_BARRIER(comm)
- }
- saveData();

В случае если в данном блоке возникает ошибка, то флаг err_fl будет true. Тогда блок будет выполнен ещё раз с данными, которые были на последнем чекпоинте (при последнем saveData). Если блок будет завершён успешно всеми процессами,, то идёт обновление бинарного файла(checkpoint).

5 Основные ссылки

1. https://github.com/enotnadoske/DistSys2020 - репозиторий на github.
2. https://fault-tolerance.org/2018/11/08/ulfm-2-1a1-docker-package/ - docker ulfm, там же можно найти tutorials.
3. https://www.youtube.com/watch?v=Ky8Zfqd4h2k&list=PLMMEMWeUUML9sVaMfbaq16Mys-EKDnRp-&index=11 - Лекция 11. 1.03.08 - определение и реализация процессорной консистентности.