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

Практикум по курсу
“Распределенные системы”

Задания:

1. Последовательная консистентность памяти и алгоритм ее реализации в DSM с полным размножением.
2. Доработка MPI-программы, реализованной в рамках курса “Суперкомпьютеры и параллельная обработка данных”. Добавление контрольных точек для продолжения работы программы в случае сбоя.

Отчет
о выполненном задании
студента 420 группы факультета ВМК МГУ
Налетова Илья Вячеславовича

Moscow
2021

Оглавление

Задание №1

Постановка задачи

Последовательная консистентность памяти и алгоритм ее реализации в DSM с полным размножением. Сколько времени потребует модификация 10 различных переменных 10-ю процессами (каждый процесс модифицирует одну переменную), находящимися на разных ЭВМ сети с шинной организацией (без аппаратных возможностей широковещания) и одновременно выдавшими запрос на модификацию. 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.

Описание алгоритма

Последовательная консистентность – это определение означает, что при параллельном выполнении, все процессы должны видеть одну и ту же последовательность записей в память.

Последовательная консистентность не гарантирует, что операция чтения возвратит значение, записанное другим процессом наносекундой или даже минутой раньше, в этой модели только точно гарантируется, что все процессы знают последовательность всех записей в память. Результат повторного выполнения параллельной программы в системе с последовательной консистентностью (как, впрочем, и при строгой консистентности) может не совпадать с результатом предыдущего выполнения этой же программы, если в программе нет регулирования операций доступа к памяти с помощью механизмов синхронизации.

W(x)1 – запись значения 1 в переменную x;
R(x)0 – чтение значения 0 из переменной x.

P1:	W(x)1			W(y)1	
P2:			W(z)1		
P3:		R(x)0	R(y)0	R(z)1	R(y)0
P4:		R(x)0	R(y)1	R(z)1	R(x)1

Программная реализация

1. Программа запускается с 10 процессами и они должны произвести каждый по одной модификации с разными переменными.
2. 10 переменных оформлены, как массив с 10 значениями, в коде можно их по-разному проинициализировать.

3. Изначально задача была понята иначе, поэтому количество модификаций, которые произведут каждый процесс это параметр, который можно менять в коде программы.
4. Видов модификаций 5: +, -, *, /, %
5. Задается модификация массивом из 3 элементов: [с какой переменной, что сделать, на сколько].
   Например [3,2,4]: умножь третью переменную на 4.
6. Каждый процесс посылает координатору запрос на модификацию, получает в ответ номер своей модификации и отсылает эту модификацию (сделано через два этапа, для лучшего понимания процесса) координатору. Координатор же рассылает ее всем.
7. После выполнения всех модификаций процесс координатор выполняет локальную модификацию, рассылает всем информацию о ней, говорит, что все, все модификации закончились и собирает значения переменных со всех процессов и выводит на экран.
   Ожидается, что в итоге все массивы (как в координаторе, так и во всех рабочих процессах) будет равны.
8. Все общение происходит через MPI_Send, MPI_Recv. Все взаимодействие происходит в main, отдельно реализована функция operation, которая применяет модификацию.
9. В коде много комментариев, почти под каждую строку, поэтому в нем легко разобраться.

Инструкция для запуска

1. mpicc dsm.c -o dsm
2. mpirun -np 10 ./dsm

Временная оценка работы алгоритма

Считаем, что всего 10 процессов.

1 из них – координатор.

10 процессов модифицируют 10 переменных.

Каждый процесс посылает координатору запрос на модификацию. 9 сообщений (1 сообщение будет локальным). В ответ на каждый запрос на модификацию координатор вышлет 9 номеров и 9 модификаций. Координатору будут переданы 9 модифицированных блоков данных.

Итого получается:

$9 * (18 * Ts + 9 * Tb * Ln + 9 * Tb * Lm)$,

где $Ln = Lm = 4$ байта

$Ts = 100$

$Tb = 1$

Задание №2

Постановка задачи

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:

a) продолжить работу программы только на “исправных” процессах;
b) вместо процессов, вышедших из строя, создать новые MPI-процессы, которые необходимо использовать для продолжения расчетов;
c) при запуске программы на счет сразу запустить некоторое дополнительное количество MPI-процессов, которые использовать в случае сбоя.

Исходная версия

Данная программа производит вычисления для задачи *"Уравнение теплопроводности в трехмерном пространстве"* (*"Heat equation over 3D data domain"*).

В исходной версии программы есть 3 основные функции, а также функция замера времени.

init_array - функция, которая инициализирует массивы A и B.

kernel_heat_3d - основная функция вычисления, в которой t раз обновляются значения значения элементов массивов A и B. Ниже представлен алгоритм вычисления:

```
static
void kernel_heat_3d(int tsteps,
        int n,
        double A[ n][n][n],
        double B[ n][n][n])
{
    int t, i, j, k;

    for (t = 1; t <= TSTEPS; t++) {
        for (i = 1; i < n-1; i++) {
            for (j = 1; j < n-1; j++) {
                for (k = 1; k < n-1; k++) {
                    B[i][j][k] = 0.125 * (A[i+1][j][k] - 2.0 * A[i][j][k] + A[i-1][j][k])
                                + 0.125 * (A[i][j+1][k] - 2.0 * A[i][j][k] + A[i][j-1][k])
                                + 0.125 * (A[i][j][k+1] - 2.0 * A[i][j][k] + A[i][j][k-1])
                                + A[i][j][k];
                }
            }
        }
        for (i = 1; i < n-1; i++) {
           for (j = 1; j < n-1; j++) {
                for (k = 1; k < n-1; k++) {
                    A[i][j][k] = 0.125 * (B[i+1][j][k] - 2.0 * B[i][j][k] + B[i-1][j][k])
                                + 0.125 * (B[i][j+1][k] - 2.0 * B[i][j][k] + B[i][j-1][k])
                                + 0.125 * (B[i][j][k+1] - 2.0 * B[i][j][k] + B[i][j][k-1])
                                + B[i][j][k];
                }
            }
        }
    }

}
```

print_array - выводит значения массивов на экран.

Параллельная версия

В рамках курса "Суперкомпьютеры и параллельная обработка данных" была реализована параллельная версия исходной программы с использованием MPI.

1. Нулевой процесс является координатором, его основная задача - заниматься координацией работы и вычислений остальных рабочих процессов, в которых в свою очередь уже и проходят все рассчеты.
2. Массивы A и B делятся между рабочими процессами по первому измерению.

3. Упрощены выражения, для уменьшения количества вычислений.
4. Для обмена информацией между процессами используется MPI_Irecv() и MPI_Isend().
5. Для замера времени используем MPI_Wtime().
6. Также используется MPI_Waitall() для ожидания выполнения всех отправок.

Устойчивая версия (задание)

Особенности работы:

1. MPI_Comm_set_errhandler - используется (так же запуск происходит с особыми флагами) для того, чтобы при убийстве процесса программа продолжала работать, а не завершалась аварийно.
2. Было решено реализовать сценарий c), то есть создать резервный процесс, который, в случае аварийного завершения одного из процессов, продолжит вычисления за него.
3. Программа разделена на блоки, в каждом из блоков может произойти смерть процесса.
4. Перемещение между блоками с помощью goto.
5. Ранк, блок и шаг на котором произойдет завершение процесса - параметры.
6. Убийство происходит с помощью raise(SIGKILL).
7. Процессы могут понять, в работе какого процесса произошла ошибка по коду возвратов функций MPI_Recv. Первый процесс, который заметил сбой, отправляет сообщение с номером процесса, который следует заменить резервному.
8. С помощью цикла с MPI_Irecv от всех процессов по очереди и функции MPI_Waitany после цикла резервный процесс понимает какой именно умер, и, как следствие, от кого ждать этой информации, что он и делает с помощью MPI_Irecv.
9. Подмена происходит так. Резервный процесс загружает данные из последней контрольной точки этого ранка, подменяет свой ранк номером умершего

процесса и перемещается в соответствующий блок, в котором произошел сбой.

Функции:

save_control_point - функция, которая сохраняет контрольные точки в формате rank_block_step.txt

init_array - функция инициализации массивов

kill_process - функция, которая убивает процесс, если его ранк, блок и шаг совпадают с указанными. Вызывается в каждом блоке.

kernel_heat_3d - основная функция, в которой происходят все вычисления. Здесь находятся почти все блоки, в которых может умереть процесс.

all_mas_to_one - функция сбора информации с процессов и объединения их значений в единые массивы A и B.

print_array - функция печати массива в файл. Файл matrix.txt будет хранить значения.

send_results - отправка значений от рабочих процессов координатору

load_control_point - загрузка контрольной точки, в случае падения процесса

Инструкция для запуска

1. docker pull abouteiller/mpi-ft-ulfm
2. alias make='docker run -v $PWD:/sandbox:Z abouteiller/mpi-ft-ulfm make' alias mpirun='docker run -v $PWD:/sandbox:Z abouteiller/mpi-ft-ulfm mpirun --oversubscribe -mca btl tcp,self'
3. make heat-3d

В Makefile уже прописаны все необходимые флаги и запуски.

Заключение

В результате данной работы были выполнены задачи:

1. Последовательная консистентность памяти и алгоритм ее реализации в DSM с полным размножением.

2. Доработка MPI-программы, реализованной в рамках курса “Суперкомпьютеры и параллельная обработка данных”. Добавление контрольных точек для продолжения работы программы в случае сбоя.

Была изучена архитектура последовательной консистентности памяти, налажены обмен информацией и общение между процессами, а также получена временная оценка работы алгоритма, при заданных параметрах, а именно 10 процессов одновременно проводят 10 модификаций.

Был изучен подход, при котором, для создания устойчивой MPI-программы, используется принцип избыточности, а конкретнее создается дополнительный процесс, который в случае необходимости, готов продолжить работу вместо завершившегося аварийно. Также изучена работа с контрольными точками, их сохранением и восстановлением из них.

Код выложен: https://github.com/Dantes4u/skpod2021