МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное
образовательное учреждение высшего образования
«Самарский национальный исследовательский университет
имени академика С.П. Королева»
(Самарский университет)

Институт информатики, математики и электроники
Факультет информатики
Кафедра технической кибернетики

ЛАБОРАТОРНАЯ РАБОТА №2

Поиск суммы элементов с использованием технологий OpenMP и MPI

по курсу
Параллельное программирование

Студент гр. 6407 _____ В.С. Гринина
Преподаватель,
к.ф.-м.н. _____ Е.С. Козлова

САМАРА 2018

ЗАДАНИЕ

Необходимо произвести запуск программ для поиска суммы элементов, которые используют технологии MPI и OpenMP, на различном количестве процессоров (потоков). Для корректной работы программ в шаблоны для запуска добавить код инициализации входных данных.

На основе технологии OpenMP реализовать три варианта программы с использованием:

1. Опции reduction;
2. Директивы critical;
3. Директивы atomic.

На основе технологии MPI было реализовать два варианта сбора информации на 0-ой процессор с использованием:

1. Операций «точка-точка»;
2. Коллективной операции.

В программе с использованием технологией MPI для корректной декомпозиции входных данных использовать широковещательную рассылку исходного массива, который генерируется только 0-ым процессом.

Для упрощения процесса написания программы допускается использование размерности вектора, кратной количеству процессоров. В ходе анализа работы программы следовало оценить время ее выполнения на различном количестве исполняющих нитей (процессов). Также нужно было оценить влияние различных функций и директив на скорость работы приложений. [1]

ВВЕДЕНИЕ

В основе параллельных вычислений лежит тот факт, что вычисления на нескольких процессорах происходят одновременно. Не всегда очевидно, каким образом организовать свою программу, чтобы в ней не было различных критических ситуаций. Например, одновременное чтение из файла или запись в него, изменение какой-либо общей переменной.

Для обучения одной из классических задач распараллеливания является задача поиска суммы элементов массива, так как алгоритм параллельных вычислений в ней очевиден, а количество дополнительных строк в коде минимально, но все-таки в ней присутствуют проблемы, описанные выше.

Данная задача имеет несколько решений как с применением технологии OpenMP, так и с MPI. С несколькими из них мы и познакомимся, узнаем их тонкости и различия между собой.

1 Ход выполнения работы

Для удаленного доступа к командной строке сервера будем использовать протокол SSH. Для простоты будем использовать командную строку. Для синхронизации файлов между компьютером и сервером используем протокол SSHFS, который позволяет примонтировать удаленную директорию. Редактировать код будем в SublimeText.

1.1 Технология OpenMP

В соответствии с заданием представленный код с технологией OpenMP был изменен с использованием:

1. Опции reduction;
2. Директивы critical;
3. Директивы atomic. [3]

Для реализации параллельного сложения элементов массива в технологии OpenMP достаточно было добавить лишь одну строчку с использованием директивы parallel с различными параметрами.

Код программ, использующих технологию OpenMP, приведен в приложении.

1.2 Технология MPI

Для программ MPI было написано два варианта работы:

1. с операциями "точка-точка";
2. с коллективной операции.

Оба варианта для передачи исходного массива использовали широковещательную рассылку всего массива. Первый вариант работы программы MPI был реализован с помощью непосредственной пересылки текущей суммы между процессами. Второй – с помощью операции глобальной редукции. [4]

На рисунке 1 представлены запуск программы на кластере с использованием скрипта и отслеживание статуса выполнения программы.

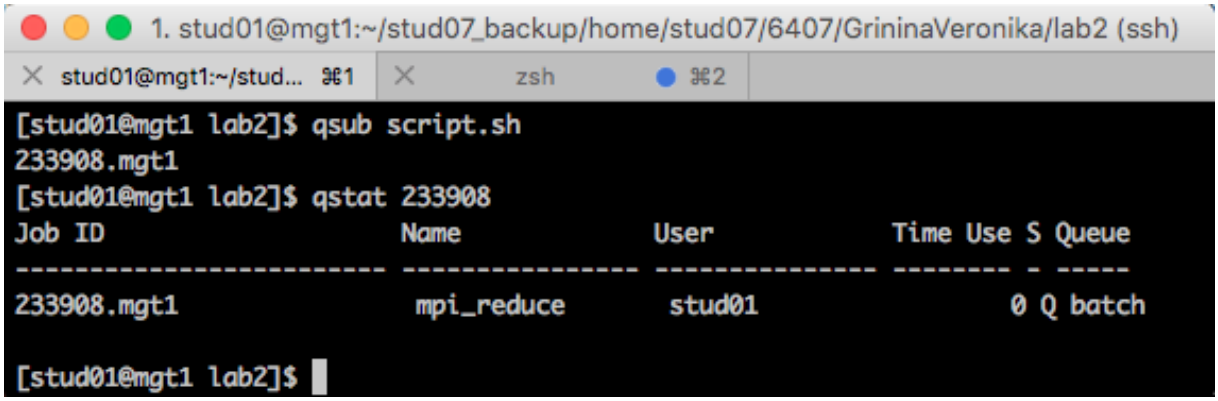

Рисунок 1 — Запуск программы MPI на кластере

Для реализации параллельного алгоритма с использованием технологии MPI пришлось написать уже заметно больше строк, так как помимо пересылки и принятия данных между процессорами есть необходимость писать обязательные функции MPI (инициализацию среды и ее различных переменных).

Код программ, использующих технологию MPI, приведен в приложении.

1.3 Время выполнения программ

Для дополнительного анализа работы программ засечем время их выполнения при различных параметрах и внесем эти данные в таблицу 1.

Таблица 1 – Время выполнения программ при различных параметрах

Количество нитей (процессов)	OpenMP			MPI	
	Reduction	Critical	Atomic	P2P	Reduce
1	0.005426	**0.027288**	**0.027361**	**0.003775**	**0.004082**
8	**0.001531**	0.818311	0.271905	0.034990	0.026409
16	0.024689	0.777127	0.279582	0.039967 (2n/8ppn) 0.025458 (4n/4ppn)	0.041518 (2n/8ppn) 0.026668 (4n/4ppn)

Взглянув на таблицу 1, можно обратить внимание, что распараллеливание данных программ не привело к большому выигрышу по

времени, а также, что полученные результаты сложно оценить объективно, так как скорость выполнения программы зависит не только от количества арифметических операций, но и от создания новых нитей/процессов, передачи/сбора данных и множества особенностей поведения кластера.

Однако, в целом, можно сказать, что для технологии OpenMP в среднем более быстрым способом распараллеливания оказалась опция reduction. Это может быть связано с тем, что опция reduction выполняет операцию более хитро, не блокируя явно общие переменные, в отличие от директив critical и atomic. Однако данная опция может выполнять только лишь ограниченный набор операций, в то время как с помощью директив critical и atomic можно написать более сложную логически программу.

Для технологии MPI по полученным результатам сложно сделать какой-либо вывод.

ЗАКЛЮЧЕНИЕ

В ходе проведения данной лабораторной работы на кластере были запущены программы OpenMP и MPI с различными реализациями параллельного блока. Были изучены директивы OpenMP и функции передачи сообщений MPI.

Были частично изучены стандарты технологий OpenMP и MPI. А именно: некоторые директивы OpenMP, их различия и их возможные опции, обязательные функции MPI, функции передачи сообщений и их различия. [1]

Был сделан вывод, что скорость выполнения программы зависит не только от количества арифметических операций, но и от множества технических факторов: создание новых нитей/процессов, передача/сбор данных и различные особенности поведения кластера. Было выявлено, что распараллеливание программы не всегда приходит к более быстрому ее выполнению.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Козлова Е.С. Методические указания к лабораторной работе №2 по курсу «Параллельное программирование» Методические указания к лабораторным работам [Текст]/ Сост. Козлова Е.С. – Самара, 2017. 11 с.

[2] Параллельные вычисления [Электронный ресурс] // Википедия: свободная энцикл. – Электрон. дан. – [Б. м.], 2017. – URL: https://en.wikipedia.org/wiki/Concurrent_computing (дата обращения: 10.09.2018).

[3] Справочное руководство по OpenMP [Электронный ресурс] // OpenMP: – Электрон. дан. – [Б. м.], 2015. – URL: https://www.openmp.org/resources/refguides/ (дата обращения: 08.10.2018).

[4] Справочное руководство в примерах по MPI [Электронный ресурс] // MPI Tutorials: – Электрон. дан. – [Б. м.], 2018. – URL: http://mpitutorial.com/tutorials/ (дата обращения: 10.10.2018).

[5] Страуструп, Б. Язык программирования C++ [Текст]/ / Б. Страуструп. – М:Бином, 2011 –1136 с

ПРИЛОЖЕНИЕ А
КОД ПРОГРАММЫ OPENMP CRITICAL

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define CHUNK 100
#define NMAX 1500000
#include "stdio.h"

int main(int argc, char* argv[]){
     omp_set_num_threads(8);
     int i;
     double *a, sum;
     a = malloc(sizeof(double) * NMAX);
     //заполнение массива
     for (i = 0; i < NMAX; i++) {
          a[i] = (double)rand();
     }
     double st_time, end_time;
     st_time = omp_get_wtime();
     sum = 0;
     #pragma omp parallel for shared(a) private(i)
          for (i = 0; i < NMAX; i++) {
               #pragma omp critical
               sum = sum + a[i];
          }
     end_time = omp_get_wtime ();
     end_time = end_time - st_time;
     printf("\nTotal Sum = %10.2f",sum);
     printf("\nTIME OF WORK IS %f ", end_time);
     free(a);
     return 0;
}
```

ПРИЛОЖЕНИЕ Б
КОД ПРОГРАММЫ OPENMP ATOMIC

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define CHUNK 100
#define NMAX 1500000
#include "stdio.h"

int main(int argc, char* argv[]){
     omp_set_num_threads(16);
     int i;
     double *a, sum;
     a = malloc(sizeof(double) * NMAX);
     //заполнение массива
     for (i = 0; i < NMAX; i++) {
          a[i] = (double)rand();
     }
     double st_time, end_time;
     st_time = omp_get_wtime();
     sum = 0;
     #pragma omp parallel for shared(a) private(i)
          for (i = 0; i < NMAX; i++) {
               #pragma omp atomic
               sum += a[i];
          }
     end_time = omp_get_wtime();
     end_time = end_time - st_time;
     printf("\nTotal Sum = %10.2f",sum);
     printf("\nTIME OF WORK IS %f ", end_time);
     free(a);
     return 0;
}
```

ПРИЛОЖЕНИЕ В
КОД ПРОГРАММЫ OPENMP REDUCTION

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define CHUNK 100
#define NMAX 1500000
#include "stdio.h"

int main(int argc, char* argv[]){
     omp_set_num_threads(16);
     int i;
     double *a, sum;
     a = malloc(sizeof(double) * NMAX);
     //заполнение массива
     for (i = 0; i < NMAX; i++) {
          a[i] = (double)rand();
     }
     double st_time, end_time;
     st_time = omp_get_wtime();
     sum = 0;
     #pragma omp parallel for reduction(+:sum)
          for (i = 0; i < NMAX; i++) {
               sum = sum + a[i];
          }
     end_time = omp_get_wtime ();
     end_time = end_time - st_time;
     printf("\nTotal Sum = %10.2f",sum);
     printf("\nTIME OF WORK IS %f ", end_time);
     free(a);
     return 0;
}
```

ПРИЛОЖЕНИЕ Г
КОД ПРОГРАММЫ СКРИПТА ДЛЯ ЗАПУСКА MPI

```
#!/bin/bash
#PBS -N MPI_mapo_8
#PBS -l walltime=00:01:10
#PBS -l nodes=1:ppn=8
#PBS -j oe
#PBS -A tk

cd $PBS_O_WORKDIR

module load impi/4
export I_MPI_DEVICE=rdma
export I_MPI_DEBUG=0
export I_MPI_FALLBACK_DEVICE=disable
mpirun -r ssh -machinefile $PBS_NODEFILE -np $PBS_NP ./MPI
```

ПРИЛОЖЕНИЕ Д
КОД ПРОГРАММЫ MPI P2P

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char* argv[]) {
     double *x, TotalSum, ProcSum = 0.0;
     int ProcRank, ProcNum, N=1500000,i;
     MPI_Status Status;

     double st_time, end_time;

     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
     MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

     x = malloc(sizeof(double) * N);

     if (ProcRank == 0) {
          //заполнение массива
          for (i = 0; i < N; i++) {
               x[i] = (double)rand();
          }
     }
     st_time = MPI_Wtime();
     // подготовка данных, рассылка 0-ым процессом по всем
остальным
     MPI_Bcast(x, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

     int k = N / ProcNum;
     int i1 = k * ProcRank;
     int i2 = k * (ProcRank + 1);

     if (ProcRank == ProcNum - 1) {
          i2 = N;
     }
     for (i = i1; i < i2; i++) {
          ProcSum = ProcSum + x[i];
     }

     if (ProcRank == 0) {
          TotalSum = ProcSum;
          for (i = 1; i < ProcNum; i++) {
               MPI_Recv(&ProcSum, 1, MPI_DOUBLE, i, 0,
MPI_COMM_WORLD, &Status);
```

```
                TotalSum = TotalSum + ProcSum;
            }
        } else {
            MPI_Send(&ProcSum, 1, MPI_DOUBLE, 0, 0,
MPI_COMM_WORLD);
        }

        MPI_Barrier(MPI_COMM_WORLD);

        end_time = MPI_Wtime();
        end_time = end_time - st_time;

        if (ProcRank == 0) {
            printf("\nTotal Sum = %10.2f", TotalSum);
            printf("\nTIME OF WORK IS %f ", end_time);
        }
        free(x);
        MPI_Finalize();
        return 0;
}
```

ПРИЛОЖЕНИЕ Е
КОД ПРОГРАММЫ MPI REDUCE

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char* argv[]) {
     double *x, TotalSum, ProcSum = 0.0;
     int ProcRank, ProcNum, N=1600000,i;
     MPI_Status Status;

     double st_time, end_time;

     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
     MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

     x = malloc(sizeof(double) * N);
     if (ProcRank == 0) {
          //заполнение массива
          for (i = 0; i < N; i++) {
               x[i] = (double)rand();
          }
     }
     st_time = MPI_Wtime();
     // подготовка данных, рассылка 0-ым процессом по всем
остальным
     MPI_Bcast(x, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

     int k = N / ProcNum;
     int i1 = k * ProcRank;
     int i2 = k * (ProcRank + 1);

     if (ProcRank == ProcNum - 1) {
          i2 = N;
     }
     for (i = i1; i < i2; i++) {
          ProcSum = ProcSum + x[i];
     }

     MPI_Reduce(&ProcSum, &TotalSum, 1, MPI_DOUBLE, MPI_SUM,
0, MPI_COMM_WORLD);
     MPI_Barrier(MPI_COMM_WORLD);

     end_time = MPI_Wtime();
     end_time = end_time - st_time;
```

```
     if (ProcRank == 0) {
          printf("\nTotal Sum = %10.2f", TotalSum);
          printf("\nTIME OF WORK IS %f ", end_time);
     }

     free(x);

     MPI_Finalize();
     return 0;
}
```