МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»

Институт информатики, математики и электроники
Факультет информатики
Кафедра технической кибернетики

ЛАБОРАТОРНАЯ РАБОТА №2

Запуск программ для поиска элементов суммы на OpenMP и MPI

по курсу

Параллельное программирование

Студент группы 6407 Исаев М.А .
Преподаватель Козлова Е.С.

Самара 2018

ЗАДАНИЕ

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, представленные ниже, добавить код инициализации входных данных.

На основе технологии OpenMP реализуйте три варианта с использованием:

1) опции reduction;

2) директивы critical;

3) директивы atomic.

На основе технологии MPI 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 необходимо 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 оценить время ее выполнения на различном количестве исполняющих нитей (процессов). Оценить влияние различных функций и директив на скорость работы приложений.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В данной работе мы ознакомились с опцией reduction и директивами critical и atomic стандарта OpenMP и с принципом рассылки данных в стандарте MPI.

ЗАПУСК ПРОГРАММ ДЛЯ ПОИСКА ЭЛЕМЕНТОВ СУММЫ НА OPENMP И MPI

В данной лабораторной работе аналогично, как и в предыдущей будем использовать протоколы SSH и SFTP для доступа к кластеру. Используя логин\пароль выданные преподавателем подключаемся к кластеру.

Наша программа состоит из пяти файлов: reduction, critical, atomic стандарта OpenMP. И два файла стандарта MPI которые отличаются способом передачи данных от процесса к процессу – в одном файле выполняется передача по звезде, в другом используется алгоритм точка-точка. Согласно [1] вставляем код необходимой директивы в исходный код и производим компиляцию и последующий запуск исполняемого файла.

В таблице 1 представлены результаты времени, для каждой из перечисленных технологий, затраченного на выполнения различным количеством потоков для умножения 900000 элементов вектора. В таблице 2 представлено ускорение на разном количестве процессоров для каждой технологии.

Таблица 1 - Время работы программ OpenMP в секундах на разном количестве процессоров для технологий: Reduction, Critical, Atomic

Количество потоков:	*Reduction:*	*Critical:*	*Atomic:*
1	0.006620	0.027734	0.012307
4	0.003858	0.299712	0.05068
8	0.004024	0.270101	0.049990

Для технологии MPI использовался вектор той же длины. Результаты вычислений представлены в таблице 2.

Таблица 2 – Время работы программ MPI на разном количестве процессоров в секундах

Количество потоков:	Коллективная операция	Операция точка-точка
1	0.000378	0.000379
4	0.000191	0.000177
8	0.000178	0.000187

Программа с использованием MPI отработала быстрее, чем программа с использованием OpenMP с любым количеством потоков и с любыми модификаторами.

ЗАКЛЮЧЕНИЕ

В лабораторной работе были изучены отличия директив стандарта OpenMP и то, как они влияют на конечное время выполнения программы. Так же мы изучили, каким образом рассылаются данные в стандарте MPI перед непосредственным исполнением кода. Узнали в чем принципиальные отличия различных подходов и как они в конечном итоге влияют на результат.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Козлова Е.С. Методические указания к лабораторной работе №2 по курсу «Параллельное программирование» Методические указания к лабораторным работам / Сост. Козлова Е.С. – Самара, 2017. 12 с.

ПРИЛОЖЕНИЕ А

Код программ на OpenMP и MPI

Исходный код программы, работающей со стандартом OpenMP, с использованием опции reduction:

```
#include <omp.h>
#define CHUNK 100
#define NMAX 1000000
#include "stdio.h"
int main(int argc, char* argv[]){
        omp_set_num_threads(2);
        int i;
        double a[NMAX], sum;
        for (i=0; i<NMAX; i++) {
          a[i]=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 ms", end_time);
        return 0;
}
```

Исходный код программы, работающей со стандартом OpenMP, с использованием директивы critical:

```
#include <omp.h>
#define CHUNK 100
#define NMAX 1000000
#include "stdio.h"
int main(int argc, char* argv[]){
        omp_set_num_threads(2);
        int i;
        double a[NMAX], sum;
        for (i=0; i<NMAX; i++) {
          a[i]=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 ms", end_time);
        return 0;
}
```

Исходный код программы, работающей со стандартом OpenMP, с использованием директивы atomic:

```
#include <omp.h>
#define CHUNK 100
#define NMAX 1000000
```

```
#include "stdio.h"
int main(int argc, char* argv[]){
	omp_set_num_threads(2);
	int i;
	double a[NMAX], sum;
	for (i=0; i<NMAX; i++) {
	  a[i]=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 = 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 ms", end_time);
	return 0;
}
```

Исходный код программы, работающей со стандартом MPI:

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]) {
```

```
        double TotalSum, ProcSum = 0.0;
        int *smallv, ProcRank, ProcNum, N=8;
        int *bigv;
        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);
        int k = N / ProcNum;
        if( ProcRank == 0 ) {
                bigv = (int *)malloc(N * sizeof(int));
                for(i=0; i < N; i++)
                        bigv[i]=rand();
        }
        smallv = (int *)malloc(k * sizeof(int));
        MPI_Scatter(bigv, k, MPI_INT, smallv, k, MPI_INT, 0,
MPI_COMM_WORLD);
        if( ProcRank == 0 )
        {
                free(bigv);
        }
        st_time = MPI_Wtime();
        for ( int i = 0; i < k; i++ )
                ProcSum = ProcSum + smallv[i];
        if ( ProcRank == 0 )
        {
                TotalSum = ProcSum;
                for ( int 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);
        free(smallv);
        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);
        }
        MPI_Finalize();
        return 0;
    }
```

Исходный код программы, работающей со стандартом MPI с использованием функции MPI_Reduce:

```
    #include "mpi.h"
    #include <math.h>
    #include <stdio.h>
    #include <stdlib.h>
    int main(int argc, char* argv[]) {
        int TotalSum, ProcSum = 0;
        int *smallv, ProcRank, ProcNum, N=8;
        int *bigv;
        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);
int k = N / ProcNum;
if( ProcRank == 0 ) {
        bigv = (int *)malloc(N * sizeof(int));
        for(i=0; i < N; i++)
                bigv[i]=rand();
}
smallv = (int *)malloc(k * sizeof(int));
MPI_Scatter(bigv, k, MPI_INT, smallv, k, MPI_INT, 0,
MPI_COMM_WORLD);
if( ProcRank == 0 ) {
        free(bigv);
}
st_time = MPI_Wtime();
for ( int i = 0; i < k; i++ ) {
        ProcSum = ProcSum + smallv[i];
        printf("\nsmallv[%d] = %d", i, smallv[i]);
}
printf("\nProcSum %d = %d", ProcRank, ProcSum);
MPI_Reduce(&ProcSum, &TotalSum, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);
free(smallv);
MPI_Barrier(MPI_COMM_WORLD);
end_time = MPI_Wtime();
end_time = end_time - st_time;
if ( ProcRank == 0 ) {
        printf("\nTotal Sum = %d",TotalSum);
        printf("\nTIME OF WORK IS %f ", end_time);
```

```
        }
        MPI_Finalize();
        return 0;
}
```