МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра «Вычислительная техника»

Дисциплина «Высокопроизводительные вычисления»

**Лабораторная работа №3**

Вариант №3

Выполнил:

студент 3 курса, гр. ИВТВМбд-31

Захарычев Н.А.

Проверил:

д. т. н., профессор кафедры ВТ

Негода В. Н.

г. Ульяновск, 2017

**Цель работы:**

Изучение методов распараллеливания реализации вычисления определенного интеграла.

**Задание по варианту:**

* Функция: *cos (x);*
* Метод численного интегрирования: метод средних прямоугольников;
* Исследование многопоточных реализаций выбранного метода численного интегрирования в среде одно-, двух-, трех машинных кластеров;

**Ход работы:**

В данной лабораторной работе использовалась библиотека *mpich* версии *3.2* и дополнительные утилиты для запуска.

**Основные функции библиотеки:**

Основные функции возвращают целое значение, которое либо равно константе **MPI\_SUCCESS**, либо содержит код произошедшей ошибки.

Перед тем, как программа сможет работать с функциями библиотеки необходимо инициализировать MPI с помощью функции:

**MPI\_Init** — обязательна для вызова, так как выполняет инициализацию библиотеки MPI.   
**MPI\_COMM\_WORLD** — идентификатор глобального коммуникатора, содержащего все процессы.   
**MPI\_Comm\_rank** — возвращает идентификатор (номер, ранг) процесса в рамках заданного коммуникатора.

**int MPI\_Finalize()**

Для передачи сообщений в MPI используется ряд функций, которые работают синхронно и асинхронно. В первом случае функция не возвращает управления, пока не завершит свою работу, во втором – сразу возвращает управление, инициировав соответствующую операцию, затем с помощью специальных вызовов есть возможность проконтролировать ход выполнения асинхронной посылки или приема данных.

Рассмотрим эти функции подробнее.

**int MPI\_Send(void\* buf, int count, MPI\_Datatype datatype, int dest,   
int msgtag, MPI\_Comm comm)**

Функция выполняет синхронную (с блокировкой) посылку сообщения с идентификатором msgtag (идентификатор выбирается самостоятельно программистом!), состоящего из count элементов типа datatype, процессу с номером dest и коммуникатором comm.

Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы. В MPI имеется ряд специальных функций посылки сообщения, устраняющих подобную неопределенность, которые подробно описаны в литературе.

**int MPI\_Recv(void\* buf, int count, MPI\_Datatype datatype, int source,   
int msgtag, MPI\_Comm comm, MPI\_Status \*status)**

Прием сообщения с идентификатором msgtag от процесса source с блокировкой. Число элементов в принимаемом сообщении не должно превосходить значения count. Если число принятых элементов меньше значения count, то гарантируется, что в буфере buf изменятся только элементы, соответствующие элементам принятого сообщения.

Блокировка гарантирует, что после возврата из подпрограммы все элементы сообщения приняты и расположены в буфере buf.

В качестве номера процесса-отправителя можно указать предопределенную константу MPI\_ANY\_SOURCE - признак того, что подходит сообщение от любого процесса. В качестве идентификатора принимаемого сообщения можно указать константу MPI\_ANY\_TAG - признак того, что подходит сообщение с любым идентификатором.

Параметр status содержит служебную информацию о ходе приема, которая может быть использована в программе. status является структурой и содержит поля: MPI\_SOURCE, MPI\_TAG и MPI\_ERROR (источник сообщения, идентификатор сообщения и возникшая ошибка, соответственно).

**int MPI\_Barrier(MPI\_Comm comm)**

MPI\_Barrier блокирует работу процессов, вызвавших данную процедуру, до тех пор, пока все оставшиеся процессы группы comm также не выполнят эту процедуру.

**Описание кластера:**

Кластер состоит из 5 компьютеров:

**Консоль управления:**

1. beowulf-309.rc.ustu

**Рабочие узлы:**

1. beowulf-309.rc.ustu
2. beowulf-309.rc.ustu
3. beowulf-309.rc.ustu
4. beowulf-309.rc.ustu

Распределение потоков по ЭВМ осуществлялось последовательно.

Таблица 1. измерения при аргументе от 0 до 1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1000 | 10000 | 100000 | 1000000 | 10000000 | 100000000 | 1000000000 |
| 1 | 0,000186 | 0,000311 | 0,002749 | 0,018975 | 0,204791 | 2,0475 | 20,4696 |
| 2 | 0,000711 | 0,000269 | 0,001655 | 0,010241 | 0,106355 | 1,05875 | 10,5413 |
| 3 | 0,000957 | 0,0003 | 0,000956 | 0,007623 | 0,073087 | 0,728684 | 7,27613 |
| 4 | 0,001411 | 0,000321 | 0,000814 | 0,006594 | 0,05769 | 0,541544 | 5,6781 |
| 5 | 0,002303 | 0,000623 | 0,000351 | 0,005993 | 0,043868 | 0,408818 | 4,28577 |
| 6 | 0,003734 | 0,000659 | 0,000256 | 0,004143 | 0,033727 | 0,278364 | 3,46522 |
| 7 | 0,004047 | 0,000666 | 0,000197 | 0,000729 | 0,005001 | 0,042358 | 3,08787 |
| 8 | 0,004888 | 0,002212 | 0,000144 | 0,000718 | 0,003068 | 0,037237 | 2,76973 |

Графики зависимости времени от числа потоков:

**Вывод:**

Как видно из графиков, при низких значениях гранулярности наблюдается снижение производительности. Это можно объяснить тем, что затраты на распределение задачи между потоками, рассылка данных по сети превосходят получаемую пользу.

Исходя из этого можно сказать, что затраты на коммуникацию между ЭВМ в кластере, оказывают значительное влияние на производительность при малых объемах вычислений.

**Исходный код:**

#include <iostream>

#include <cmath>

#include <mpi.h>

#include <time.h>

using namespace std;

float f(float x) {

return 1 / sin(x);

}

double integral(int n, double a, double b)

{

int i;

double result, h;

result = 0;

h = (b - a) / n; //Шаг сетки

for (i = 0; i < n; i++)

{

result += f(a + h \* (i + 0.5)); //Вычисляем в средней точке и добавляем в сумму

}

result \*= h;

return result;

}

int main(int argc, char \*argv[]){

int n = 1000;

double a = 1, b = 3;

double res = 0;

int process\_id;

int ierr;

int process\_num;

MPI\_Status status;

int master = 0;

MPI\_Init (&argc, &argv);

ierr = MPI\_Comm\_rank (MPI\_COMM\_WORLD, &process\_id); //

/\*

Получение количества процессоров

\*/

ierr = MPI\_Comm\_size (MPI\_COMM\_WORLD, &process\_num);

double proc\_arg[3];

double delta = fabs(a - b) / (process\_num);

double proc\_n = (double) n / (process\_num);

int tag = 1;

clock\_t start = clock();

if(process\_id == master){

cout << "processors count " << process\_num << endl;

for (int process = 1; process < process\_num; process++)

{

proc\_arg[0] = a + delta \* (process);

proc\_arg[1] = proc\_arg[0] + delta;

proc\_arg[2] = proc\_n;

//x1=proc\_arg[0] x2= proc\_arg[1]

ierr = MPI\_Send (proc\_arg, 3, MPI\_DOUBLE, process, tag, MPI\_COMM\_WORLD);

//

}

} else{

ierr = MPI\_Recv(proc\_arg, 3, MPI\_DOUBLE, master, tag, MPI\_COMM\_WORLD, &status);

//

}

ierr = MPI\_Barrier (MPI\_COMM\_WORLD);

if(process\_id != master){

double res\_l = integral(proc\_arg[2], proc\_arg[0], proc\_arg[1]);

int target = master;

tag = 2;

ierr = MPI\_Send (&res\_l, 1, MPI\_DOUBLE, target, tag, MPI\_COMM\_WORLD);

} else {

res = integral(proc\_n, a, a+delta); // master process

for(int i = 0; i < process\_num - 1; i++){

double res\_l = 0;

tag = 2;

ierr = MPI\_Recv (&res\_l, 1, MPI\_DOUBLE, MPI\_ANY\_SOURCE, tag, MPI\_COMM\_WORLD, &status);

res += res\_l;

}

clock\_t end = clock();

float during = ((double)(end - start) / CLOCKS\_PER\_SEC);

cout << "res " << res << "; time " << during << endl;

}

ierr = MPI\_Finalize ();

return 0;

}