Федеральное государственное бюджетное образовательное учреждение высшего образования
"Уфимский государственный авиационный технический университет"

Кафедра Высокопроизводительных вычислительных технологий и систем

Дисциплина: Основы суперкомпьютерных технологий и параллельное программирование

Отчет по лабораторной работе № 1

Тема: «Параллельное вычисление суммы числового ряда»

Группа ПМ-353	Фамилия И.О.	Подпись	Дата	Оценка
Студент	Шамаев И.Р.			
Принял	Юлдашев А.В.			

Уфа 2022

Цель: научиться программно реализовывать простейшие параллельные вычислительные алгоритмы и проводить анализ их эффективности на многопроцессорных вычислительных системах с распределённой памятью на примере задачи параллельного вычисления суммы числового ряда.

Теоретический материал

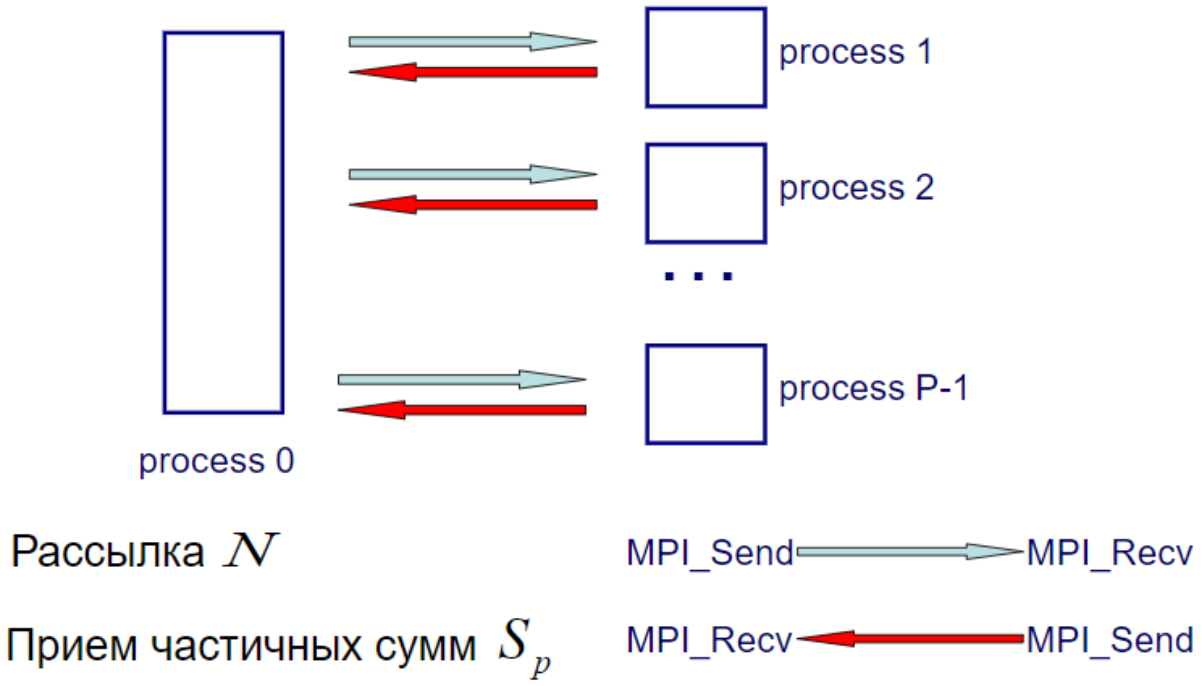

Рисунок 1 Общение между процессорами с помощью функций MPI на примере вычисления частичной суммы рядаI

Функция инициализации имеет следующий вид: int MPI_Init(int *argc, char **argv[]); Она возвращает предопределённые MPI константы:

- MPI_SUCCESS – возвращается в случае успешного выполнения
- MPI_ERR_ARG – ошибка неправильного задания аргумента
- MPI_EE_INTERN – внутренняя ошибка (нехватка памяти)
- MPI_ERR_UNKNOWN – неизвестная ошибка

Функция завершения работы с MPI: int MPI_Finalize(void);

Функция определения количества процессов size в коммуникационной группе с коммуникатором comm: int MPI_Comm_size(MPI_Commcomm, int*size); Функция, возвращающая номер rank вызвавшего ее процесса, входящего в коммуникационную группу с коммуникатором comm: intMPI_Comm_rank(MPI_commcomm, int* rank);

Блокирующая функция передачи данных: int MPI_Send(void* sbuf, intcount,MPI_Datatypedatatype, intdest, inttag, MPI_Commcomm)

Входные параметры:

- sbuf – адрес в памяти, начиная с которого размещаются передаваемые данные;
- count – количество передаваемых элементов;
- datatype – тип передаваемых элементов;

- dest – номер процесса-получателя сообщения;
- tag – метка передаваемого сообщения;
- comm – коммуникатор.

Блокирующая функция приема данных: intMPI_Recv(void* rbuf, intcount, MPI_Datatypedatatype, intsource, inttag, MPI_commcomm, MPI_Status*status);

Входные параметры:

- count – количество получаемых элементов;
- datatype – тип получаемых элементов;source–номер процесса-отправителя сообщения;
- tag – метка принимаемого сообщения;comm–коммуникатор.Входные параметры:Выходные параметры:
- rbuf – адрес в памяти, начиная с которого размещаютсяпринимаемые данные;
- status – структура, содержащая информацию опринятомсообщении.

Выходные параметры:

- rbuf – адрес в памяти, начиная с которого размещаются принимаемые данные;
- status – структура, содержащая информацию о принятом сообщении.

Структура status имеет три поля.

- Status.MPI_SOURCE - номер процесса-отправителя;
- Status.MPI_TAG - метка принимаемого сообщения;
- Status.MPI_ERROR - код завершения приема сообщения.

Тип данных datatype может быть одной из предопределенных констант.

- MPI_CHAR
- MPI_UNSIGNED_CHAR
- MPI_SHORT MPI_UNSIGNED_SHORT
- MPI_INT MPI_UNSIGNED_INT
- MPI_LONG MPI_UNSIGNED_LONG
- MPI_FLOAT
- MPI_DOUBLE MPI_LONG_DOUBLE
- MPI_BYTEMPI_PACKED

Определение времени выполнения параллельной программы: double MPI_Wtime();

Практическая часть

Задание

1. Создать параллельную версию написанной ранее программы вычисления суммы ряда с использованием базовых функций MPI
2. В программе предусмотреть следующее
    a) Ввод-вывод данных должен осуществляться только через нулевой процесс, который рассылает остальным процессам число членов ряда, введённое через аргумент командной строки. Перед рассылкой фиксируется время начала выполнения параллельного участка программы.
    b) Каждый процесс определяет количество членов, которое он суммирует. Для этого вычисляется две величины – число членов на один процессор и остаток.
    c) Если остаток равен нулю, то каждый процесс суммирует одинаковое количество членов ряда. Если количество членов ряда не кратно числу процессов, то остаток от деления равномерно распределяется между всеми процессами.
    d) Каждый процесс считает сумму отведённое ему части ряда и после окончания расчёта пересылает её нулевому процессу.
    e) Нулевой процесс находит искомую сумму, как сумму своей частичной суммы и всех присланных. Фиксируется время окончания параллельного участка программы и результаты выводятся на экран (сумма ряда и затраченное время).
3. Запустить программы на кластере при числе процессов p = 1, 2, 4, 8, 16, 32, 64 и 96. Размерность подобрать так, чтобы время выполнения параллельной программы при p = 1 составляло около 150 и 300 с.
4. Вычислить ускорение и эффективность, построить их графики в зависимости от числа процессов. Составить отчет.

Ход работы

Числовой ряд имеет следующий вид:

$$\sum_{n=2}^{N} \frac{(-1)^{n-1}}{n^2 - n}$$

В ходе выполнения программы на кластере было замерено время расчётов. Результаты приведены в следующей таблице:

p\N	1350000000	2700000000
1	150,3	300,3
2	75,1	149,7
4	37,5	74,8
8	19,0	37,5
16	10,5	19,8
32	4,7	10,7
64	2,5	4,9
96	1,8	3,4

Таблица 1. Время работы программы на различном числе ядер.

Отношение времени выполнения параллельной программы на одном процессоре (ядре) $T1$ ко времени выполнения параллельной программы на p процессорах Tp называется ускорением при использовании p ядер:

$$S_p^* = \frac{T_1^*}{T_p}$$

Отношение ускорения $Sp*$ к количеству ядер p называется эффективностью при использовании p ядер:

$$E_p^* = \frac{S_p^*}{p}$$

По результатам расчётов были построены графики ускорения и эффективности:

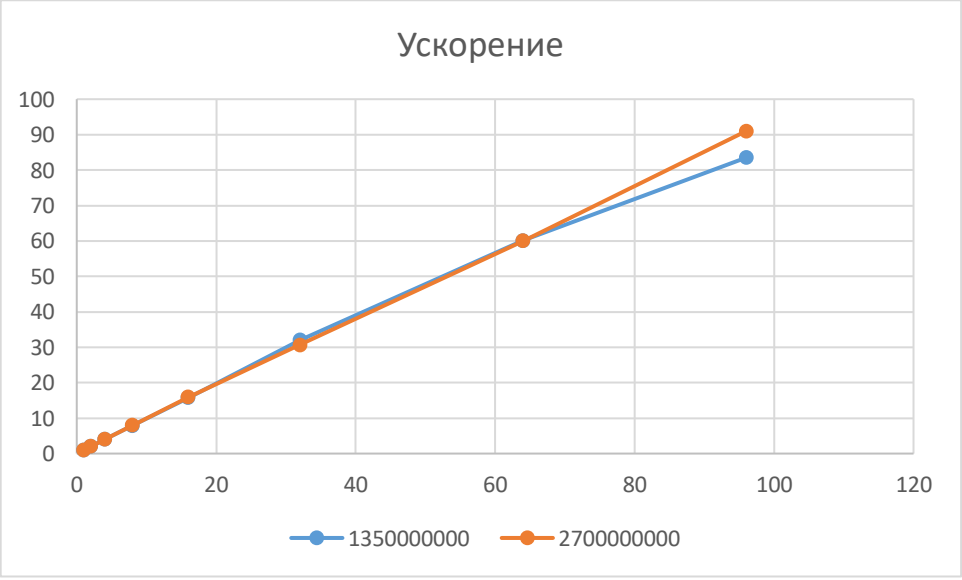

Рисунок 2. Ускорение вычислений программы в зависимости от числа ядер.

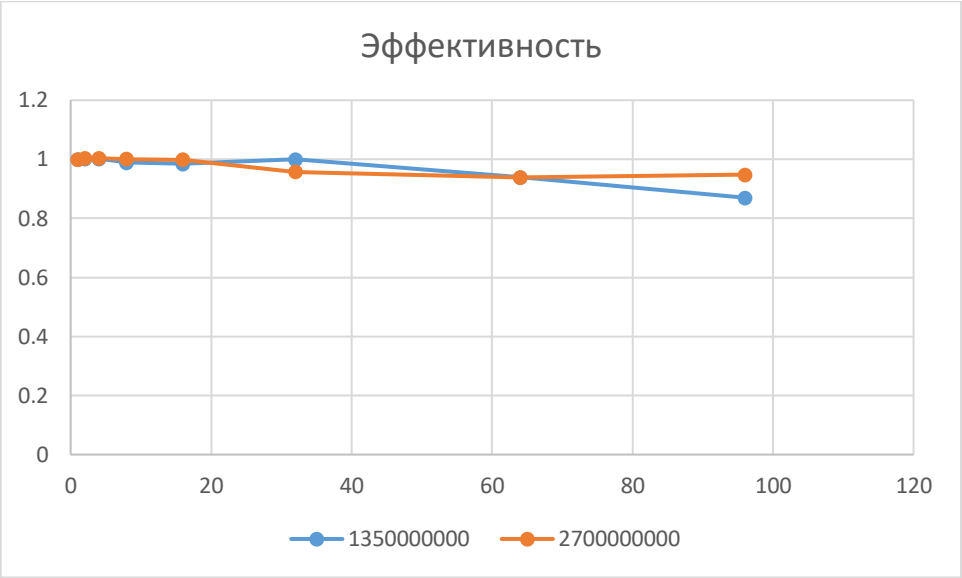

Рисунок 3. Эффективность распараллеливания вычислений в зависимости от числа ядер.

По графикам можно сделать вывод о том, что вычисление на большем количестве процессоров даёт около-идеальное ускорение. Небольшую заминку можно объяснить тем, что время расчета начиная с шестого эксперимента довольно мало и как следствие на производительность заметно может повлиять фоновая активность.

Вывод

В ходе выполнения лабораторной работы была посчитана частичная сумма ряда с использованием средств MPI на кластере. По результатам работы были построены графики ускорения и эффективности.

Приложение

```
#include "mpi.h"
#include <stdio.h>
#include <iostream>
#include <cmath>
#include <stdlib.h>

int main(int argc, char* argv[])
{
	int MyID, NumProc;
	int iError = MPI_Init(&argc, &argv); //Функция инициализации
	MPI_Comm_size(MPI_COMM_WORLD, &NumProc); //Функция определения количества процессов в коммуникационной группе с коммуникатором comm
	MPI_Comm_rank(MPI_COMM_WORLD, &MyID); //Функция, возвращающая номер rank вызвавшего ее процесса, входящего в коммуникационную группу с коммуникатором comm
	MPI_Status status;//Статус полученного сообщения
	std::cout.precision(8);
	if (iError != MPI_SUCCESS) //возвращается в случае успешного выполнения
	{
		std::cout << "MPI error!\n";
		exit(1);
	}

	MPI_Barrier(MPI_COMM_WORLD);//Останавливает выполнение вызвавшей ее задачи до тех пор, пока не будет вызвана изо всех остальных задач, подсоединенных к указываемому коммуникатору
	double timerStart = MPI_Wtime();//Определение времени выполнения параллельной программы

	if (MyID == 0)
	{
		long long n = atoll(argv[1]); //Анализирует str, интерпретируя ее содержимое как целое число
		double sum = 0.;
		double localSum = 0.;
		long long amountOfOperations = n / NumProc;//Кол-во операций
//		long long remainderOperations = n % NumProc;//Остатки

		for (int i = 1; i <= NumProc-1; i++)
		{
			MPI_Send(&n, 1, MPI_LONG, i, 1000 + i, MPI_COMM_WORLD); //Блокирующая функция передачи данных
		}


		long long from = n - amountOfOperations + 1 + 1;
		long long to = n + 1;
		for (long long i = from; i <= to; i++)
		{
			sum += pow(-1., i - 1)/(pow(i, 2) - i);
		}
```

```
            //std::cout << "Sum that ID: " << MyID << " calculated. S = " <<
sum << "\n";
            for (int i = 1; i <= NumProc - 1; i++)
            {
                  MPI_Recv(&localSum,     1,     MPI_DOUBLE,     i,     1000,
MPI_COMM_WORLD, &status); //Блокирующая функция приема данных
                  sum += localSum;
            }
            double timerEnd = MPI_Wtime();
            std::cout << "Number of steps: " << n << std::endl;
            std::cout << "Partial sum is: " << sum << std::endl;
            std::cout  <<  "Elapsed  time:  "  <<  timerEnd  -  timerStart  <<
std::endl;
      }
      else if (MyID > 0)
      {
            double localSum = 0.;
            long long n;
            //std::cout << "Processor ID: " << MyID << std::endl;
            MPI_Recv(&n,  1,  MPI_LONG,  0,  1000  +  MyID,  MPI_COMM_WORLD,
&status);
            long long amountOfOperations = n / NumProc;
            long long remainderOperations = n % NumProc;
            long long from;
            long long to;
            if (MyID <= remainderOperations)
            {
                  from = (amountOfOperations+1) * (MyID-1) + 1 + 1;
                  to = from + amountOfOperations + 1;
            }
            else
            {
                  from  =  remainderOperations  *  (amountOfOperations  +  1)  +
(MyID - 1 - remainderOperations) * amountOfOperations + 1;
                  to = from + amountOfOperations-1;
            }
            //std::cout << "from: " << from << " to " << to << "\n";
            for (long long i = from; i <= to; i++)
            {
                  localSum += pow(-1., i - 1)/(pow(i, 2) - i);
            }
            MPI_Send(&localSum, 1, MPI_DOUBLE, 0, 1000, MPI_COMM_WORLD);

      }
      MPI_Finalize();//функция завершения работы с MPI
      return 0;
}
```