Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра автоматизированных систем управления (АСУ)

ОСНОВНЫЕ ФУНКЦИИ MPI

Лабораторная работа №1

по дисциплине

«Параллельное программирование»

Студент гр. 430-2

_____А.А. Лузинсан

«____»_____2023 г.

Руководитель

Доцент кафедры АСУ

_____ С.М. Алфёров

«____»_____2023 г.

Томск 2023

Оглавление

Введение

Цель работы: освоить применение основных функций MPI на примере параллельной программы численного интегрирования.

Индивидуальное задание по варианту №6-n:

$$\int_{-0.2}^{1.0} \frac{dx}{1+(c*x)^2} = \frac{1}{c} arctg(c*x); \quad c = 0.9$$

1 Ход работы

1. Создать в домашнем каталоге папку для лабораторной работы №1. Скопировать в него программу integn.c. Проверить работоспособность программы в режиме последовательного и параллельного выполнения на разном числе процессов. Какой метод численного интегрирования используется в программе? Разобраться в MPI функциях, используемых в программе. Определить назначение и типы параметров функций. Какие дополнительные функции, кроме шести основных используются в программе?

Посредством подключения через Visual Code Remote было произведено удалённое подключение к кластеру cluster.asu.tusur.ru. Далее файл integn.cpp был скопирован в каталог лабораторной работы и скомпилирован вызовом в терминале команды: *mpicxx -o lab1 integn.cpp*. Программа вычисляет интеграл от функции cos(x) в интервале от -0.5 до 0.8 методом численного интегрирования, а именно методом прямоугольника. Последовательный вызов программы, выполняемый командой *mpirun -np <n> lab1* на разном числе процессов (1, 2, 4, 8) и количестве интервалов (10000, 100000, 1000000, 10000000) привел к результату, который приведён в таблице 1.1.

Таблица 1.1 — Результаты работы программы integn.cpp

Количество процессов	Количество интервалов | Time estimating			
	1.00E+04	1.00E+05	1.00E+06	1.00E+07
Последовательный	6.050E-04	5.389E-03	4.787E-02	4.598E-01
2 процесса	3.530E-04	2.298E-03	2.270E-02	2.267E-01
4 процесса	3.300E-04	1.205E-03	1.143E-02	1.155E-01
8 процессов	1.738E-03	1.414E-02	1.885E-02	5.815E-02
AVERAGE INTEGRAL	1.197E+00	1.197E+00	1.197E+00	1.197E+00
AVERAGE ERROR	8.427E-10	8.423E-12	7.928E-14	6.850E-15

Как видно из таблицы, с ростом числа разбиений увеличивается время вычисления и уменьшается погрешность. Однако стоит заметить, что при количестве разбиений равное 1000000, увеличение количества потоков уменьшает общее время на вычисление интегральной суммы.

В программе используется несколько основных функций. Помимо них, присутствует вызов следующих функций:

• MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm) — функция передачи сообщения от процесса отправителя, где buf — адрес буфера памяти отправляемого сообщения, count — количество элементов данных в сообщении, type — тип элементов данных сообщения, dest — ранг процесса-получателя, tag — значение-тег для идентификации сообщения, comm — коммуникатор, в рамках которого выполняется передача данных.

• MPI_Recv(void *buf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Status *status) — buf, count, type, comm — аналогично MPI_Send, только для приёма сообщения, source — ранг процесса-получателя, tag — тег сообщения, которое должно быть принято для процесса, status — указатель на структуру данных с информацией о результате выполнения операции приёма данных.

2. Скопировать программу в новый файл. Произвести замену подынтегральной и первообразной функции в соответствие с вариантом. В программе задать значения переменных для пределов интегрирования и параметра математической функции. Выполнить программу на различном числе процессов. Результаты программы направлять переназначением стандартного вывода в файлы результатов.

Для реализации данного пункта задачи постребовалось заменить уже существующие функции f() и fi() на соответствующие варианту подынтегральную и первообразную функции.

Таким образом подынтегральная функция: f(a) = 1/(1 + pow(0.9 * a, 2)), а соответствующая ей первообразная: fi(a) = (10/(double)9)*atan(0.9 * a).

Результаты выполнения программы на различном числе процессов представлены в таблице 1.2.

Таблица 1.2 — Результаты работы программы в соответствие с вариантом

Количество процессов	Количество интервалов | Time estimating			
	1.00E+04	1.00E+05	1.00E+06	1.00E+07
Последовательный	1.125E-03	1.071E-02	9.425E-02	9.164E-01
2 процесса	5.700E-04	4.626E-03	4.580E-02	4.575E-01
4 процесса	4.960E-04	2.755E-03	2.318E-02	2.314E-01
8 процессов	5.800E-04	1.232E-03	1.242E-02	1.189E-01
AVERAGE INTEGRAL	1.012E+00	1.012E+00	1.012E+00	1.012E+00
AVERAGE ERROR	4.791E-10	1.215E-05	1.215E-06	1.215E-07

В данном случае явно прослеживается польза параллельного выполнения программы, т. к. вне зависимости от используемого количества разбиений при увеличении количества процессов уменьшается общее время вычисления интеграла.

3. Скопировать программу с индивидуальной функцией в новый файл. Заменить индивидуальные функции передачи и приема на коллективные функции MPI_Bcast и MPI_Reduce. Заменить в программе назначения пределов интегрирования и параметра функции на ввод их с терминала. Ввод провести в нулевом процессе. Упаковать введенные с терминала пределы интегрирования и параметр функции в буфер, разослать буфер всем процессам и распаковать (функции MPI_Pack и MPI_Unpack).

В нулевом процессе осуществляется ввод данных с консоли в программу. Далее с помощью метода multiple_pack данные запаковываются в буфер и рассылаются посредством метода broadcast на все процессы. Получив буфер, процессы распаковывают в своих экземплярах класса

необходимые члены класса. Наконец, после вычисления, все процессы посредством метода reduce посылают root процессу свой результат, в свою очередь root процесс принимает данные, применяет оператор MPI_SUM и выводит результат вычислений. Результаты выполнения параллельной программы на различном числе процессов представлены в таблице 1.3.

Таблица 1.3 — Результаты работы программы с использованием запаковки данных

Количество процессов	Количество интервалов | Time estimating			
	1.00E+04	1.00E+05	1.00E+06	1.00E+07
Последовательный	1.208E-03	1.068E-02	9.358E-02	9.184E-01
2 процесса	7.153E-04	4.616E-03	4.596E-02	4.586E-01
4 процесса	7.148E-04	2.785E-03	2.346E-02	2.315E-01
8 процессов	6.501E-04	1.255E-03	1.190E-02	1.174E-01
AVERAGE INTEGRAL	1.012E+00	1.012E+00	1.012E+00	1.012E+00
AVERAGE ERROR	4.791E-10	1.215E-05	1.215E-06	1.215E-07

Как видно из таблицы, с ростом количества разбиений программа работает медленнее, однако, в перспективе конкретного количества разбиений, увеличение количества процессов в результате приводит к меньшим потерям по времени. Листинг программы представлен приложении А.1.

4. Провести анализ времен выполнения всех трех программ. Какие выводы и рекомендации можно сделать из этого анализа?

В результате анализа всех трёх программ можно сделать вывод, что с увеличением количества процессов повышается эффективность выполнения вычисления, что заметнее всего на большем количестве разбиений интервала определённого интеграла. Также стоит заметить, что при использовании запаковки и распаковки программа в некоторых случаях работает медленнее, так как для данных операций нужны временные ресурсы.

Заключение

В результате выполнения лабораторной работы я освоила применение основных функций MPI на примере параллельной программы численного интегрирования.

Приложение А
(обязательное)
Листинг программы

Листинг А.1

```
#include "mpi.h"
#include <cstdio>
#include <math.h>
#include <iostream>
#include <cassert>
#include <fstream>
#include <sstream>
#include <stdio.h>
#include <stdarg.h>

static double f(double a, double c = 0.9);
static double fi(double a, double c = 0.9);

class Communicator{
protected:
   MPI_Comm comm;
   int numprocs;
   char processor_name[MPI_MAX_PROCESSOR_NAME];
   int namelen;
protected:
   Communicator(int argc, char *argv[],
        MPI_Comm _comm = MPI_COMM_WORLD){
      int error;
      if (error = MPI_Init(&argc, &argv))  throw error;
      comm = _comm;
      MPI_Comm_size(comm, &numprocs);
      MPI_Get_processor_name(processor_name, &namelen);
   }

   virtual void printInfo(std::string accompanying_message = "",
                          std::ostream &out = std::cout){
      out << "\n\tProcessor name: " << processor_name
         << "\n\tNumber of processes: " << numprocs
         << "\n\t" << accompanying_message;
   }
   ~Communicator() { MPI_Finalize(); }
};
```

```
class Process : public Communicator
{
protected:
   int PID;
   enum PIDs  { INIT };
public:
   int getPID() { return PID; }
public:
   Process(int argc, char *argv[], MPI_Comm _comm = MPI_COMM_WORLD)
      : Communicator(argc, argv, _comm)
   {    MPI_Comm_rank(_comm, &PID);     }

   void printInfo(std::string accompanying_message = "",
                  std::ostream &out = std::cout)
   {out << "\n" << PID << ": " << accompanying_message;       fflush(NULL);}

   void send(void *buffer, int to_PID,
          MPI_Datatype type = MPI_INT, int count_data = 1,
          int tag = 1)
   {    MPI_Send(buffer, count_data, type, to_PID, tag, comm); }

   MPI_Status receive(void *buffer, int from_PID,
                 MPI_Datatype type = MPI_INT, int count_data = 1, int tag = 1)
   {
      MPI_Status status;
      MPI_Recv(buffer, count_data, type, from_PID, tag, comm, &status);
      return status;
   }

   void broadcast(void *buffer, MPI_Datatype type = MPI_INT,
              int count_data = 1, int root = Process::INIT)
   {    MPI_Bcast(buffer, count_data, type, root, comm);  }

   void reduce(void *sendbuf, void *recvbuf, MPI_Datatype type = MPI_INT,
            int root = Process::INIT, MPI_Op _operator = MPI_SUM,
            int count_data = 1)
   {MPI_Reduce(sendbuf, recvbuf, count_data, type, _operator, root, comm);}

   void pack(void *data, int count_data, void *buf, int count_buf,
          MPI_Datatype type = MPI_INT, int *bufpos = NULL)
   {    MPI_Pack(data, count_data, type, buf, count_buf, bufpos, comm);       }
   char *multiplePack(int lenBytes, std::string szTypes, ...){
      va_list vl; va_start(vl, szTypes);
```

```
        char *buf = (char *)malloc(lenBytes);
        int buffpos = 0;
        for (int i = 0; szTypes[i] != '\0'; ++i) {
            union Printable_t{ int i; float f; double d; char c; char *s; } Printable;
            switch (szTypes[i])
            {
            case 'i':
                Printable.i = va_arg(vl, int);
                pack(&Printable.i, 1, buf, lenBytes, MPI_INT, &buffpos);
                break;
            case 'f':
                Printable.f = va_arg(vl, double);
                pack(&Printable.f, 1, buf, lenBytes, MPI_FLOAT, &buffpos);
                break;
            case 'd':
                Printable.d = va_arg(vl, double);
                pack(&Printable.d, 1, buf, lenBytes, MPI_DOUBLE, &buffpos);
                break;
            case 'c':
                Printable.c = va_arg(vl, int);
                pack(&Printable.c, 1, buf, lenBytes, MPI_CHAR, &buffpos);
                break;
            case 's':
                Printable.s = va_arg(vl, char *);
                pack(&Printable.s, 1, buf, lenBytes, MPI_CHAR, &buffpos);
                break;
            default:
                break;
            }
        } va_end(vl);
        return buf;
    }
    void unpuck(void *inbuf, int insize, void *outbuf, int *bufpos,
                    MPI_Datatype type = MPI_INT, int outcount = 1)
    {    MPI_Unpack(inbuf, insize, bufpos, outbuf, outcount, type, comm);      }
};

class Integral : public Process{
private:
    int intervals = 0;
    double xl = -0.2, xh = 1.0;
    double c = 0.9;
    double step;
```

```
    double sum = 0.0;
    double startwtime, endwtime;
    std::ofstream fout;
public:
    Integral(int argc, char *argv[], MPI_Comm comm = MPI_COMM_WORLD,
            std::string filename = "output.txt"): Process(argc, argv, comm)
    {    fout.open(filename, std::ios::out);       }
    ~Integral() { fout.close(); }

    void printInfo(std::string accompanying_message = "",
                        std::ostream &out = std::cout)
    {
        out << "\nNumber of intervals: " << intervals
            << "\nLow border: " << xl << "\nHight border: " << xh
            << "\nParameter: " << c << "\n"
            << accompanying_message; fflush(NULL);
    }

    void unpuckParams(void *buf)
    {
        int length = sizeof(int) + sizeof(double) * 3 + 100;
        int bufpos = 0;
        unpuck(buf, length, &intervals, &bufpos);
        unpuck(buf, length, &xl, &bufpos, MPI_DOUBLE);
        unpuck(buf, length, &xh, &bufpos, MPI_DOUBLE);
        unpuck(buf, length, &c, &bufpos, MPI_DOUBLE);
    }

    int getIntervals(){
        int len_buf = sizeof(int) + sizeof(double) * 3 + 100;
        char *buf;
        if (PID == Process::INIT){
            std::cout << "\nEnter the number of intervals (0 quit): ";
            int n; std::cin >> n; intervals = n;
            if (n != 0) {
                double temp;
                std::cout << "Enter the low border: ";
                std::cin >> temp; xl = temp;
                std::cout << "Enter the hight border: ";
                std::cin >> temp; xh = temp;
                std::cout << "Enter the parameter of function: ";
                std::cin >> temp; c = temp;
            } else
```

```
            xl, xh, c = 0, 0, 0;
        buf = multiplePack(len_buf, "iddd", intervals, xl, xh, c);
        startwtime = MPI_Wtime();
    }
    else buf = new char[len_buf];
    broadcast(buf, MPI_PACKED, len_buf);
    if (PID != Process::INIT){
        unpuckParams(buf);
        if (intervals == 0) return 0;
    }
    delete[] buf;
    return intervals;
}

double evalIntegral(){
    step = (xh - xl) / static_cast<double>(intervals);
    for (int i = PID + 1; i <= intervals; i += numprocs){
        double x = xl + step * ((double)i - 0.5);
        sum += f(x, c);
    }
    sum *= step;
    std::stringstream str;
    str << std::scientific << "SUMM: " << sum;
    Process::printInfo(str.str()); str.clear(); str.str("");
    return sum;
}

double summarazing(){
    double integral = 0;
    reduce(&sum, &integral, MPI_DOUBLE);
    endwtime = MPI_Wtime();
    if (PID == Process::INIT){
        Communicator::printInfo("", fout);
        printInfo("", fout);
        std::stringstream str;
        str << std::scientific << "Integral is approximately: " << integral;
        Process::printInfo(str.str(), fout); str.clear(); str.str("");
        str << std::scientific << "Error: " << integral - fi(xh, c) + fi(xl, c);
        Process::printInfo(str.str(), fout); str.clear(); str.str("");
        str << std::scientific << "Time of calculation: " << endwtime - startwtime;
        Process::printInfo(str.str(), fout); str.clear(); str.str("");
        return integral;
    }
```

```
        else      return sum;
    }
    static double f(double a, double c)
    {    return 1 / (1 + pow(c * a, 2));    }
    static double fi(double a, double c)
    {    return (1 / c) * atan(c * a);        }
};
void lab1(int argc, char *argv[]){
    Integral proc(argc, argv);
    while (true){
        int success = proc.getIntervals();
        if (!success)   break;
        else{
            proc.evalIntegral();
            proc.summarazing();
}}}

int main(int argc, char *argv[]){
    lab1(argc, argv);
    return 0;
}
```