**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**"Уфимский государственный авиационный технический университет"**

**Кафедра** Высокопроизводительных вычислительных технологий и систем

**Дисциплина:** Алгоритмы Параллельного Программирования

**Отчет по лабораторной работе № 2**

**Тема:** «Параллельное вычисление произведения двух матриц»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Группа МКН-315 | Фамилия И.О. | Подпись | Дата | Оценка |
| Студент | Гильманов И.И. |  |  |  |
| Принял | Спеле В.В. |  |  |  |

**Уфа 2022**

**Цель:** научиться использовать принцип геометрической декомпозиции в параллельных алгоритмах и создавать параллельные программы для систем с распределенной памятью с использованием коллективных функций MPI на примере вычисления произведения двух матриц

**Теоретический материал.**

***MPI (Message Passing Interface)*** – это стандартизованная библиотека функций, призванная обеспечить совместную работу параллельных процессов путем организации передачи сообщений между ними.

***Интерфейс MPI:***

* Модель программирования MPI.
* Базовые функции MPI.
* Функции парного взаимодействия.
* Функции коллективного взаимодействия.
* Функция определения времени
* Пользовательские типы данных.
* Управление областью взаимодействия и группой процессов.

***Базовые понятия MPI***

* MPI предназначен для написания программ для MIMD архитектур
* Каждая программа представляет собой совокупность одновременно работающих процессов, которые могут обмениваться сообщениями.
* Все процессы объединяются в группы.
* Обмен сообщениями возможен между процессами одной группы, которой поставлена в соответствие своя область связи.
* Идентификатор области связи называется коммуникатором.
* При запуске программы все доступные ей процессы объединяются в начальную группу с общей областью связи, имеющей коммуникатор MPI\_COMM\_WORLD.

***Базовые операции стандарта MPI***

* Операции межпроцессорного взаимодействия типа
* «точка-точка».
* Операции коллективного взаимодействия.
* Операции над группами процессов.
* Операции с областями коммуникации.
* Операции с топологией процессов.
* Функции ввода/вывода (появились в MPI-2.0)

***Особенности реализации MPI для C/C++***

1. Первая строка программы

#include “mpi.h”

1. В MPI принят ANSI C стандарт.
2. Нумерация массивов начинается с 0.
3. Массивы хранятся по строкам.
4. Логические переменные являются переменными типа integer со значением 0 в случае false и любым не нулевым значением, обозначающем true.

***Базовые функции MPI***

1. int ***MPI\_Init***(int \*argc, char \*\*argv[]); – инициализация параллельной части программы.

Возвращает предопределенные константы

MPI\_SUCCESS - возвращается в случае успешного выполнения,

MPI\_ERR\_ARG - ошибка неправильного задания аргумента,

MPI\_ERR\_INTERN - внутренняя ошибка (нехватка памяти),

MPI\_ERR\_UNKNOWN - неизвестная ошибка.

1. int ***MPI\_Finalize***(void); завершение параллельной части программы.
2. int ***MPI\_Comm\_size***(MPI\_Comm comm, int\* size); определение числа процессов size в коммуникационной группе с коммуникатором comm.
3. int ***MPI\_Comm\_rank***(MPI\_comm comm, int\* rank); определение номера rank вызвавшего ее процесса, входящего в коммуникационную группу с коммуникатором comm
4. int ***MPI\_Send***(void\* sbuf, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm) – передача сообщений от одного процесса к другому

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.

1. int ***MPI\_Recv***(void\* rbuf, int count, MPI\_Datatype datatype, int source, int tag, MPI\_comm comm, MPI\_Status \*status) – прием сообщений от одного процесса к другому

*Входные параметры:*

count – количество получаемых элементов;

datatype – тип получаемых элементов;

source – номер процесса-отправителя сообщения;

tag – метка принимаемого сообщения;

comm – коммуникатор.

*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.

***Функция совмещенного приема/передачи.***

int ***MPI\_Sendrecv***(void\* sbuf, int scount, MPI\_Datatype sdatatype, int dest, int stag, void\* rbuf, int rcount, MPI\_Datatype rdatatype, int source, int rtag, MPI\_comm comm, MPI\_Status \*status);

*Входные параметры:*

sbuf – адрес в памяти, начиная с которого размещаются передаваемые данные;

scount – количество передаваемых элементов;

sdatatype – тип отправляемых элементов;

dest – номер процесса-получателя сообщения;

stag – метка отправляемого сообщения;

rcount – количество получаемых элементов;

rdatatype – тип получаемых элементов;

source – номер процесса-отправителя сообщения;

rtag – метка принимаемого сообщения;

comm – коммуникатор.

*Выходные параметры:*

rbuf – адрес в памяти, начиная с которого размещаются принимаемые данные;

status – структура, содержащая информацию о принятом сообщении.

***Функции коллективного взаимодействия***

1. int ***MPI\_Bcast***(void \*buf, int count, MPI\_Datatype datatype, int root, MPI\_Comm comm);Функция предназначена для рассылки данных, хранящихся на одном процессе, всем остальным процессам группы.

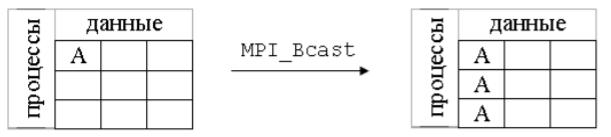


Рисунок 1. MPI\_Bcast

*Входные параметры:*

buf – адрес в памяти, начиная с которого размещаются передаваемые данные;

count – количество рассылаемых элементов;

datatype – тип отправляемых данных;

root – номер процесса-отправителя сообщения;

comm – коммуникатор.

1. int ***MPI\_Gather***(void \*sbuf, int scount, MPI\_Datatype stype, void \*rbuf, int rcount, MPI\_Datatype rtype, int root, MPI\_Comm comm);

Функция предназначена для сбора данных со всех процессов на одном (так называемый, "совок“).



Рисунок 2. MPI\_Gather

*Входные параметры*:

sbuf – адрес в памяти на каждом процессе, начиная с которого размещаются отправляемые данные;

scount – количество элементов, отправляемых с каждого процесса;

stype – тип отправляемых данных;

rcount – количество принимаемых элементов от каждого процесса;

rtype – тип принимаемых данных;

root – номер процесса, на котором осуществляется сборка сообщений;

comm – коммуникатор.Выходные параметры:

rbuf – адрес в памяти на процессе с номером root, начиная с которого размещаются принимаемые сообщения.

1. Вариант функции с варьируемым кол-вом собираемых элементов:

int ***MPI\_Gatherv***(void \*sbuf, int scount, MPI\_Datatype stype, void \*rbuf, int \*rcounts, int \*displs, MPI\_Datatype rtype, int root, MPI\_Comm comm);

*Характерные отличия:*

rcounts – массив длин принимаемых от процессов сообщений;

displs – массив позиций в приемном буфере, по которым размещаются принимаемые сообщения.

1. int ***MPI\_Scatter***(void \*sbuf, int scount, MPI\_Datatype stype, void \*rbuf, int rcount, MPI\_Datatype rtype, int root, MPI\_Comm comm);

Функция предназначена для рассылки данных с одного процесса всем остальным процессам (так называемый, “разбрызгиватель”).

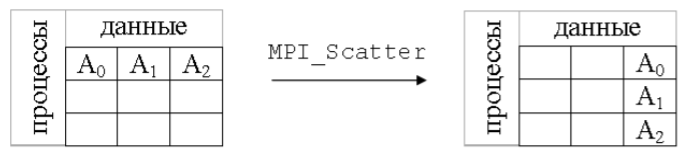


Рисунок 3. MPI\_Scatter

*Входные параметры:*

sbuf – адрес в памяти на процессе-отправителе сообщения, начиная с которого размещаются отправляемые данные;

scount – количество элементов, отправляемых каждому процессу;

stype – тип отправляемых данных;

rcount – количество элементов, принимаемых каждым процессом (длина принимаемого сообщения);

rtype – тип принимаемых данных;

root – номер процесса-отправителя сообщения;

comm – коммуникатор.Выходные параметры:

rbuf – адрес в памяти на каждом процессе, начиная с которого размещаются принимаемые сообщения.

1. Вариант функции с варьируемым кол-вом рассылаемых элементов:

int ***MPI\_Scatterv***(void \*sbuf, int \*scounts, int \*displs, MPI\_Datatype stype, void \*rbuf, int rcount, MPI\_Datatype rtype, int root, MPI\_Comm comm);

*Характерные отличия:*

scounts – массив, содержащий количество элементов в каждой части, на которые разбивается сообщение;

displs – массив позиций, определяющий начальные положения каждой части сообщения.

1. int ***MPI\_Reduce***(void \*sbuf, void \*rbuf, int count, MPI\_Datatype datatype, MPI\_Op op, int root, MPI\_Comm comm);

Функция проделывает операцию op над данными, хранящимися в sbuf в каждом процессе группы, результат которой записывается в rbuf в процесс с номером root.

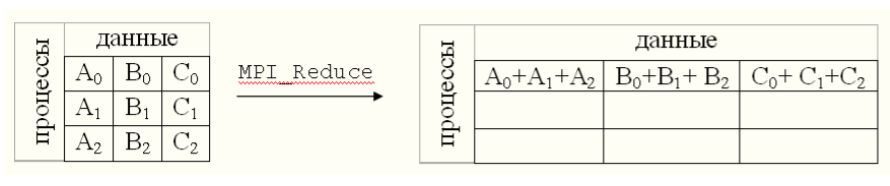


Рисунок 4. MPI\_Reduce

*Входные параметры:*

sbuf – адрес в памяти на каждом процессе, по которому хранятся исходные данные для распределенной операции;

count – количество элементов в sbuf;

datatype – тип данных, над которыми производится распределенная операция;

root – номер процесса, на котором осуществляется размещение результата выполнения операции;

op – название распределенной операции;

comm – коммуникатор.

*Выходные параметры:*

rbuf – адрес в памяти, по которому размещаются результаты выполнения операции.

***12 предопределенных операций:***

MPI\_MAX – поиск поэлементного максимума;

MPI\_MIN – поиск поэлементного минимума;

MPI\_SUM – вычисление суммы векторов;

MPI\_PROD – вычисление поэлементного произведения векторов;

MPI\_LAND – логическое “И”;

MPI\_LOR – логическое “ИЛИ”;

MPI\_LXOR – логическое исключающее “ИЛИ”;

MPI\_BAND – бинарное “И”;

MPI\_BOR – бинарное “ИЛИ”;

MPI\_BXOR – бинарное исключающее ИЛИ;

MPI\_MAXLOC – поиск индексированного максимума;

MPI\_MINLOC – поиск индексированного минимума.

**Практическая часть**

**Задание**

1. Написать программу с использованием коллективных функций MPI.
2. В программе предусмотреть следующее:

а) размерность L вводится пользователем (аргумент командной строки), размерность N=10L;

б) матрицы заполняются случайными элементами вещественного типа;

в) замер времени выполнения параллельной программы (вычисления + коммуникации).

1. 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 60 и 600 с.
2. Вычислить ускорение и эффективность, построить их графики в зависимости от числа процессов. Составить отчет с оценкой полученных результатов.

**Ход работы**

Таблица 1. Результаты замеров времени

|  |  |  |
| --- | --- | --- |
| p\N | 1050 | 2500 |
| 1 | 64,3231 | 665,257 |
| 2 | 37,5568 | 391,559 |
| 4 | 23,4362 | 254,934 |
| 8 | 15,9321 | 198,578 |
| 16 | 8,5346 | 125,191 |
| 32 | 4,461 | 64,235 |
| 64 | 2,3342 | 34,751 |
| 96 | 1,7513 | 21,966 |

Вычислим ускорение и эффективность по формулам , :

Таблица 2. Вычисление ускорения для разного числа процессоров

|  |  |  |
| --- | --- | --- |
| p\N | 1050 | 2500 |
| 1 | 1 | 1 |
| 2 | 1,712688514 | 1,69899555 |
| 4 | 2,744604501 | 2,60952639 |
| 8 | 4,037327157 | 3,35010424 |
| 16 | 7,536744546 | 5,31393631 |
| 32 | 14,41898677 | 10,3566124 |
| 64 | 27,55680747 | 19,1435354 |
| 96 | 36,72877291 | 30,2857598 |

Таблица 3. Вычисление эффективности для разного числа процессоров

|  |  |  |
| --- | --- | --- |
| p\N | 1050 | 2500 |
| 1 | 1 | 1 |
| 2 | 0,856344257 | 0,84949778 |
| 4 | 0,686151125 | 0,6523816 |
| 8 | 0,504665895 | 0,41876303 |
| 16 | 0,471046534 | 0,33212102 |
| 32 | 0,450593337 | 0,32364414 |
| 64 | 0,430575117 | 0,29911774 |
| 96 | 0,382591384 | 0,31547666 |

Графики ускорения и эффективности:

Рисунок 1. График зависимости ускорения от числа процессоров

Рисунок 2. График зависимости эффективности от числа процессоров

**Вывод**: в ходе лабораторной работы на примере задачи параллельного умножения матриц научились реализовывать простейшие параллельные вычислительные алгоритмы и проводить анализ их эффективности.

**Код программы.**

#include "mpi.h"

#include <iostream>

#include <math.h>

#include <stdlib.h>

#include <stdio.h>

using namespace std;

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

{

srand((unsigned int)1);

int MyID, NumProc, ierror;

ierror = MPI\_Init(&argc, &argv);

if (ierror != MPI\_SUCCESS)

cout << "MPI Init error" << endl;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &NumProc);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &MyID);

if (argc != 2)

{

cout << "[!] 'N' needed" << endl;

return -1;

}

long long L;

L = atoi(argv[1]);

long long N = 10 \* L, p = N / NumProc, o = N % NumProc;

double \*A = (double\*)calloc(N\*L, sizeof(double)),

\*B = (double\*)calloc(L\*L, sizeof(double)),

\*C = (double\*)calloc(N\*L, sizeof(double)),

\*Ap = (double\*)calloc((p + 1)\*L, sizeof(double)),

\*Cp = (double\*)calloc((p + 1)\*L, sizeof(double)),

\*normid = (double\*)calloc(1, sizeof(double)),

\*Norma = (double\*)calloc(1, sizeof(double));

int \*count = (int\*)calloc(NumProc, sizeof(int)), \*displs = (int\*)calloc(NumProc, sizeof(int)),

\*end=(int\*)calloc(NumProc, sizeof(int)),

ccount;

double tstart, tfinish;

if (MyID == 0)

{

for (int i = 0; i < N\*L; i++) A[i] = -0.5 + (double)rand() / RAND\_MAX;;

for (int i = 0; i < L\*L; i++) B[i] = -0.5 + (double)rand() / RAND\_MAX;;

tstart = MPI\_Wtime();

}

MPI\_Barrier(MPI\_COMM\_WORLD);

if (MyID == 0)

{

for (int i = 0; i < NumProc; i++)

{

if (o > i)

{

count[i] = (p + 1)\*L;

displs[i] = i \* (p + 1)\*L;

}

else

{

count[i] = p \* L;

displs[i] = o \* (p + 1)\*L + (i - o)\*p\*L;

}

}

}

if (MyID < o)

{

ccount = (p + 1)\*L;

end[MyID] = p + 1;

}

else

{

end[MyID] = p;

ccount = (p)\*L;

}

MPI\_Scatterv(A, count, displs, MPI\_DOUBLE, Ap, ccount, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(B, L\*L, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Barrier(MPI\_COMM\_WORLD);

for (int i = 0; i < end[MyID]; i++)

{

for (int j = 0; j < L; j++)

{

Cp[L \* (i)+j] = .0;

for (int k = 0; k < L; k++)

{

Cp[L \* (i)+j] += Ap[L \* (i)+k] \* B[L \* (k)+j];

}

normid[0] += Cp[L \* (i)+j] \* Cp[L \* (i)+j];

}

}

MPI\_Gatherv(Cp, ccount, MPI\_DOUBLE, C, count, displs, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Reduce(normid, Norma, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (MyID == 0)

{

tfinish = MPI\_Wtime() - tstart;

cout << "> Norm = " << Norma[0] << endl;

cout << "> Time = " << tfinish << endl;

}

free(A); free(C); free(Ap); free(B); free(Cp); free(normid); free(Norma); free(count); free(displs); free(end);

MPI\_Finalize();

}