МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ   
РОССИЙСКОЙ ФЕДЕРАЦИИ  
  
Федеральное государственное автономное образовательное учреждение   
высшего образования   
«Самарский национальный исследовательский университет   
имени академика С.П. Королёва» (Самарский университет)  
  
Факультет информатики  
Кафедра программных систем  
  
Дисциплина  
**Параллельное программирование**

**ОТЧЕТ**

по лабораторной работе № 1

**Параллельный алгоритм матричного умножения**

**– реализация на OpenMP и MPI**

Студент: Андреева А.А.

Группа: 6413-020302D   
  
Преподаватель: Оплачко Д.С.  
Подпись: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Самара 2021

Содержание

[Постановка задачи 3](#_Toc84727884)

[Описание вычислительной системы, на которой производились эксперименты 5](#_Toc84727885)

[1 Разработка последовательного алгоритма 6](#_Toc84727886)

[2 Разработка параллельного алгоритма с применением OpenMP 6](#_Toc84727887)

[3 Разработка параллельного алгоритма с применением MPI 7](#_Toc84727888)

[Результаты вычислительных экспериментов 9](#_Toc84727889)

[Выводы 14](#_Toc84727890)

[Список использованных источников 15](#_Toc84727891)

[Приложение а](#_Toc84727892)  [Последовательный алгоритм умножения матриц 16](#_Toc84727893)

[Приложение б](#_Toc84727894)  [Параллельный алгоритм умножения матриц с использованием OpenMP 18](#_Toc84727895)

[Приложение В](#_Toc84727896) [Параллельный алгоритм умножения матриц с использованием MPI 20](#_Toc84727897)

# Постановка задачи

Цель работы: Изучение OpenMP –технологии параллельного программирования для вычислительных систем с общей памятью. Изучение MPI – широко распространенной технологии параллельного программирования для распределенных вычислительных систем.

1. На основе последовательного алгоритма матричного умножения разработать параллельный алгоритм решения той же задачи. Алгоритм должен допускать параллельное выполнение произвольного количества фрагментов алгоритма p <= n, где n – количество строк и столбцов матрицы.

2. Разработать программу на языке C с использованием технологии OpenMP, реализующую указанный алгоритм.

3. Измерить время работы программы для различных размеров матриц. Примерный набор значений размеров матриц (n):10, 50, 100, 200, 400, 800, 1000, 1200, 1500, 2000.

Количество потоков параллельной программы p = {2, 4, 8}. Тип элементов матриц – double.

4. Результаты измерений записать в таблицы.

5. Используя максимальное количество потоков (p=8), измерить длительность работы параллельных программ, использующих различные значения параметра schedule() директивы #pragma omp for.

Дополнить таблицу для случая p=8 результатами измерений.

6. На основе последовательного алгоритма матричного умножения, разработанного в лабораторной работе №1, разработать параллельный алгоритм решения той же задачи. Алгоритм должен допускать параллельное выполнение произвольного количества фрагментов алгоритма N <= n, где n – количество строк и столбцов матрицы.

7. Разработать программу на языке C с использованием библиотеки MPI, реализующую указанный алгоритм.

8. Измерить время работы программы для различных размеров матриц, совпадающих с теми, которые использовались в экспериментах лабораторной работы №1. Примерный набор значений размеров матриц (n):10, 50, 100, 200, 400, 800, 1000, 1200, 1500, 2000.

Количество процессов параллельной программы p = {2, 4, 8}.

Тип элементов матриц – double.

9. Результаты измерений записать в таблицы.

Последовательный вариант программы в данном случае должен быть программой на MPI с числом процессов p=1.

10. Составить отчет по результатам работы.

Отчет должен содержать графики, построенные по данным из таблиц.

Допустимо строить только графики ускорения.

# Описание вычислительной системы, на которой производились эксперименты

Модель процессора: Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz 2.30 GHz

Количество физических ядер: 2

Поддержка hyperthreading: да

1. Разработка последовательного алгоритма

Последовательный алгоритм умножения матриц представляется тремя вложенными циклами: первый отвечает за номер строки матрицы А, второй – за номер столбца матрицы В, третий – перебирает все элементы в выбранных строке матрицы А и столбце матрицы В, суммируя произведения соответствующих элементов. После завершения третьего цикла сумма записывается в ячейку матрицы С. Этот алгоритм является итеративным и ориентирован на последовательное вычисление строк матрицы С. При умножении квадратных матриц размера NxN количество выполненных операций имеет порядок O(n3).

Текст программы с реализацией алгоритма представлен в Приложении А.

1. Разработка параллельного алгоритма с применением OpenMP

Одним из наиболее популярных средств программирования для компьютеров с общей памятью, базирующихся на традиционных языках программирования и использовании специальных комментариев, в настоящее время является технология OpenMP. За основу берётся последовательная программа, а для создания её параллельной версии пользователю предоставляется набор директив, функций и переменных окружения.

Значительная часть функциональности OpenMP реализуется при помощи директив компилятору. Директивы OpenMP в программах на языке Си оформляются указаниями препроцессору, начинающимися с #pragma omp.

С помощью опции shared данной директивы указывается, что матрицы A, B и С являются общими. Опция private задает список переменных, для которых порождается локальная копия в каждой нити. Опция schedule указывает, каким образом итерации цикла распределяются между потоками, и может принимать следующие значения:

* static – распределение итераций цикла; размер блока – chunk. Первый блок из chunk итераций выполняет нулевая нить, второй блок – следующая и т.д. до последней нити, затем распределение снова начинается с нулевой нити. Если значение chunk не указано, то всё множество итераций делится на непрерывные куски примерно одинакового размера (конкретный способ зависит от реализации), и полученные порции итераций распределяются между нитями.
* dynamic – динамическое распределение итераций с фиксированным размером блока: сначала каждая нить получает chunk итераций (по умолчанию chunk=1), та нить, которая заканчивает выполнение своей порции итераций, получает первую свободную порцию из chunk итераций. Освободившиеся нити получают новые порции итераций до тех пор, пока все порции не будут исчерпаны. Последняя порция может содержать меньше итераций, чем все остальные.
* guided – динамическое распределение итераций, при котором размер порции уменьшается с некоторого начального значения до величины chunk (по умолчанию chunk=1) пропорционально количеству ещё не распределённых итераций, делённому на количество нитей, выполняющих цикл. Размер первоначально выделяемого блока зависит от реализации. В ряде случаев такое распределение позволяет аккуратнее разделить работу и сбалансировать загрузку нитей. Количество итераций в последней порции может оказаться меньше значения chunk.

Директива num\_threads указывается количество используемых потоков.

Текст программы с реализацией алгоритма представлен в Приложении Б.

1. Разработка параллельного алгоритма с применением MPI

Наиболее распространенной технологией программирования параллельных компьютеров с распределенной памятью является технология MPI. Основным способом взаимодействия параллельных процессов в таких системах является передача сообщений друг другу. Это и отражено в названии технологии – Message Passing Interface.

При использовании библиотеки MPI производится умножение матриц происходит следующим образом. Каждый процесс получает несколько строк матрицы A и всю матрицу B. После того, как каждый процесс завершил умножение своей части, он отправляет результат умножения (полосу матрицы C) главному процессу. Главный процесс собирает полученные фрагменты и измеряет общую длительность выполнения программы.

Для рассылки строк матриц использовались функции MPI\_Send и MPI\_Recv. Для измерения длительности работы программы в главном процессе – функция MPI\_Wtime().

Текст программы с реализацией алгоритма представлен в Приложении B.

# Результаты вычислительных экспериментов

В таблице 1 приведены результаты измерения длительности вычислений последовательного и параллельного с использованием OpenMP алгоритмов.

На рисунке 1 приведены графики ускорений в зависимости от размера матрицы, количества процессов и значений параметра schedule при использовании OpenMP.

Рисунок 1 – графики ускорений для OpenMP

В таблице 2 приведены результаты измерения длительности вычислений последовательного и параллельного с использованием с использованием MPI алгоритмов.

На рисунке 2 приведены графики ускорений в зависимости от размера матрицы и количества процессов при использовании MPI.

Рисунок 2 – графики ускорений для MPI

По результатам вычислений видно, что на малых размерах матриц последовательный алгоритм работает быстрее, чем параллельные, так как данные полностью умещаются в память и доступ к ним осуществляется быстро, поэтому на организацию параллельных вычислений затрачивается больше времени. Но начиная с некоторых размеров матриц, параллельные алгоритмы начинают давать явное преимущество по времени.

Таблица 1 – Сравнение результатов матричного умножения последовательного алгоритма и параллельного с технологией OpenMP

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Размерность матриц, n** | **10** | **50** | **100** | **200** | **400** | **800** | **1000** | **1200** | **1500** | **2000** |
| **Время работы последовательного алгоритма t,c** | 0,00001 | 0,00125 | 0,011956 | 0,072307 | 1,109266 | 8,955925 | 21,49836 | 32,70873 | 80,34717 | 227,4071 |
| **Время работы алгоритма с помощью OpenMP p=2 t,c** | 0,002128 | 0,003566 | 0,011289 | 0,064118 | 0,633921 | 6,23956 | 12,46801 | 22,6086 | 58,12313 | 157,3193 |
| **Ускорение S** | **0,004699** | **0,350533** | **1,059084** | **1,127718** | **1,749849** | **1,435346** | **1,724281** | **1,446739** | **1,382361** | **1,445513** |
| **Время работы алгоритма с помощью OpenMP p=4 t,c** | 0,009521 | 0,006228 | 0,015084 | 0,048125 | 0,365606 | 4,769156 | 10,02894 | 17,65711 | 42,80941 | 125,574 |
| **Ускорение S** | **0,00105** | **0,200706** | **0,792628** | **1,502483** | **3,034048** | **1,877885** | **2,143632** | **1,85244** | **1,876858** | **1,810941** |
| **Время работы алгоритма с помощью OpenMP p=8 t,c** | 0,011975 | 0,013439 | 0,014376 | 0,050825 | 0,372061 | 4,174846 | 10,27558 | 15,93827 | 41,03097 | 120,0797 |
| **Ускорение S** | **0,000835** | **0,093013** | **0,831664** | **1,422666** | **2,981409** | **2,145211** | **2,092179** | **2,052214** | **1,958208** | **1,893802** |

Таблица 2 – Длительность работы параллельных программ с различными значениями параметра schedule() p=8

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **shedule** | **Размерность матриц, n** | **10** | **50** | **100** | **200** | **400** | **800** | **1000** | **1200** | **1500** | **2000** |
| **static** | **Время работы алгоритма t,c** | 0,015968 | 0,011905 | 0,019606 | 0,064797 | 0,355072 | 4,285052 | 9,415284 | 16,1781 | 41,77262 | 122,4965 |
| **Ускорение S** | **0,000626** | **0,104998** | **0,609813** | **1,1159** | **3,124059** | **2,090039** | **2,283347** | **2,021791** | **1,923441** | **1,856438** |
| **dynamic** | **Время работы алгоритма t,c** | 0,026187 | 0,013558 | 0,021166 | 0,049868 | 0,331109 | 3,921292 | 10,19064 | 17,32843 | 36,52267 | 117,3775 |
| **Ускорение S** | **0,000382** | **0,092196** | **0,564868** | **1,449968** | **3,350154** | **2,283922** | **2,109618** | **1,887576** | **2,199926** | **1,9374** |
| **guided** | **Время работы алгоритма t,c** | 0,011975 | 0,013439 | 0,014376 | 0,050825 | 0,372061 | 4,174846 | 10,27558 | 15,93827 | 41,03097 | 120,0797 |
| **Ускорение S** | **0,000835** | **0,093013** | **0,831664** | **1,422666** | **2,981409** | **2,145211** | **2,092179** | **2,052214** | **1,958208** | **1,893802** |

Таблица 3 – Сравнение результатов матричного умножения последовательного алгоритма и параллельного с технологией MPI

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Размерность матриц, n** | **10** | **50** | **100** | **200** | **400** | **800** | **1000** | **1200** | **1500** | **2000** |
| **Время работы последовательного алгоритма t,c** | 0,00001 | 0,00125 | 0,011956 | 0,072307 | 1,109266 | 8,955925 | 21,49836 | 32,70873 | 80,34717 | 227,4071 |
| **Время работы алгоритма с помощью MPI p=2 t,c** | 0,001001 | 0,002552 | 0,006072 | 0,034951 | 0,406406 | 4,125599 | 7,136897 | 15,58645 | 37,28511 | 90,92154 |
| **Ускорение S** | **0,00999** | **0,489812** | **1,969038** | **2,068811** | **2,729453** | **2,170818** | **3,012284** | **2,098536** | **2,15494** | **2,501136** |
| **Время работы алгоритма с помощью MPI p=4 t,c** | 0,001719 | 0,003615 | 0,005529 | 0,038062 | 0,364232 | 3,548663 | 6,71335 | 13,96976 | 31,52699 | 78,56376 |
| **Ускорение S** | **0,005817** | **0,345781** | **2,162416** | **1,899716** | **3,045493** | **2,523746** | **3,202329** | **2,341396** | **2,54852** | **2,894555** |
| **Время работы алгоритма с помощью MPI p=8 t,c** | 0,003077 | 0,003624 | 0,010932 | 0,03675 | 0,337364 | 3,501951 | 7,06838 | 12,99747 | 29,64885 | 80,04777 |
| **Ускорение S** | **0,00325** | **0,344923** | **1,09367** | **1,967537** | **3,288039** | **2,55741** | **3,041483** | **2,516547** | **2,709959** | **2,840893** |

Выводы

Из приведенных в таблице 1 результатов видно, что параллельные алгоритмы с OpenMP на малых размерах матрицы 10-50 не дают выигрыша по времени исполнения, однако на больших размерах матрицы параллельные версии программ дают значительное ускорение в ~ 2 раза.

Приведенные в таблице 2 результаты позволяют сделать вывод, что наибольшее ускорение S = 3.35 выдал параллельный алгоритм со значением dynamic параметра schedule при n = 400, при n = 2000 это значение также позволило добиться наибольшего ускорения S = 1.94.

Из приведенных в таблице 3 результатов видно, что параллельные алгоритмы с MPI начинают давать ускорение только при значениях матрицы более 100. На больших размерах матрицы параллельные версии программ дают значительное ускорение в ~ 2.9 раза, что схоже с результатом алгоритмов с OpenMP.

Список использованных источников

1. Жидченко В.В. Параллельный алгоритм матричного умножения – реализация на OpenMP и MPI: Метод. указания к лабораторным занятиям/ Самар. ун-т - сост. В.В.Жидченко - Самара, 2019 - 8 с.
2. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления - СПб.: БХВ-Петербург, 2002. - 608 с. (дата обращения: 03.10.2021).
3. Р.В. Жалнин, Е.Н. Панюшкина, Е. Е. Пескова, П.А. Шаманаев. Основы параллельного программирования с использованием технологий MPI И OPENMP - Саранск: Изд-во СВМО, 2013. – 78 с. (дата обращения: 04.10.2021).
4. Учебник по OpenMP [Электронный ресурс]. URL: https://pro-prof.com/archives/4335#page\_1 (дата обращения: 04.10.2021).

приложение а

Последовательный алгоритм умножения матриц

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <malloc.h>

#define N 2000

//последовательный алгоритм

int main() {

double t1, t2;

double\*\* matrA = (double\*\*)malloc(N \* sizeof(double\*));

double\*\* matrB = (double\*\*)malloc(N \* sizeof(double\*));

double\*\* matrC = (double\*\*)malloc(N \* sizeof(double\*));

for (int i = 0; i < N; i++) {

matrA[i] = (double\*)malloc(N \* sizeof(double));

matrB[i] = (double\*)malloc(N \* sizeof(double));

matrC[i] = (double\*)malloc(N \* sizeof(double));

}

srand(1);

for (int i = 0; i < N; i++) {

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

{

matrA[i][j] = rand();

matrB[i][j] = rand();;

}

}

t1 = omp\_get\_wtime();//время начала работы алгоритма

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

matrC[i][j] = 0;

for (int k = 0; k < N; k++) {

matrC[i][j] += matrA[i][k] \* matrB[k][j];

}

}

}

t2 = omp\_get\_wtime();//время окончания работы алгоритма

printf("t = %f\n", t2 - t1);

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

{

free(matrA[i]);

free(matrB[i]);

free(matrC[i]);

}

free(matrA);

free(matrB);

free(matrC);

system("pause");

}

# приложение б

# Параллельный алгоритм умножения матриц с использованием OpenMP

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <malloc.h>

#include <omp.h>

#include "mpi.h"

#define N 2000

//параллельный алгоритм с OpenMP

int i, j, k;

int main() {

omp\_set\_num\_threads(8);//количество процессов

double t1, t2;

double\*\* matrA = (double\*\*)malloc(N \* sizeof(double\*));

double\*\* matrB = (double\*\*)malloc(N \* sizeof(double\*));

double\*\* matrC = (double\*\*)malloc(N \* sizeof(double\*));

for (int i = 0; i < N; i++) {

matrA[i] = (double\*)malloc(N \* sizeof(double));

matrB[i] = (double\*)malloc(N \* sizeof(double));

matrC[i] = (double\*)malloc(N \* sizeof(double));

}

srand(1);

for (int i = 0; i < N; i++) {

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

{

matrA[i][j] = rand();

matrB[i][j] = rand();

}

}

t1 = omp\_get\_wtime();//время начала работы алгоритма

#pragma omp parallel for private(i,j,k) shared(matrA,matrB,matrC) schedule(dynamic) //распараллеленный алгоритм умножения матриц

for (i = 0; i < N; i++) {

for (j = 0; j < N; j++) {

matrC[i][j] = 0;

for (k = 0; k < N; k++) {

matrC[i][j] = matrC[i][j] + matrA[i][k] \* matrB[k][j];

}

}

}

t2 = omp\_get\_wtime();//время окончания работы алгоритма

printf("t = %f\n", t2 - t1);

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

{

free(matrA[i]);

free(matrB[i]);

free(matrC[i]);

}

free(matrA);

free(matrB);

free(matrC);

system("pause");

}

Приложение В

Параллельный алгоритм умножения матриц с использованием MPI

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <malloc.h>

#include <omp.h>

#include "mpi.h"

//#define N 2000

////параллельный алгоритм с MPI

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

{

const int N = 1200;

double t1, t2;

int size, rank;

MPI\_Status status;

MPI\_Init(argc, &argv);

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

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

int chunk = N / size;

double\* matrA = (double\*)malloc(sizeof(double) \* N \* N);

if (rank == 0) {

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

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

matrA[j + i \* N] = 1.0;

}

double\* matrB = (double\*)malloc(sizeof(double) \* N \* N);

if (rank == 0) {

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

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

matrB[j + i \* N] = 1.0;

}

double\* PartRes = (double\*)malloc(sizeof(double) \* chunk \* N);

double\* matrC = (double\*)malloc(sizeof(double) \* N \* N);

t1 = MPI\_Wtime();//время начала работы алгоритма

if (rank == 0) {

for (int i = 1; i < size; i++) {

//передача сообщений между процессами

MPI\_Send(matrA + i \* N \* chunk, N \* chunk, MPI\_DOUBLE, i, 0, MPI\_COMM\_WORLD);

}

}

else

{

//прием сообщения процессом получателем

MPI\_Recv(matrA, N \* chunk, MPI\_DOUBLE, 0, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &status);

}

//осуществление рассылки всем процессам

MPI\_Bcast(matrB, N \* N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

for (int i = 0; i < chunk; i++) {

for (int j = 0; j < N; j++) {

PartRes[i \* N + j] = 0;

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

PartRes[i \* N + j] += matrA[i \* N + k] \* matrB[k \* N + j];

}

}

if (rank != 0) {

MPI\_Send(PartRes, N \* chunk, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD);

}

if (rank == 0) {

for (int i = 0; i < chunk \* N; i++) {

matrC[i] = PartRes[i];

}

for (int i = 1; i < size; i++) {

MPI\_Recv(matrC + i \* N \* chunk, N \* chunk, MPI\_DOUBLE, i,

MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &status); }

}

t2 = MPI\_Wtime();//время окончания работы алгоритма

if (rank == 0) {

printf("N=%d\n", N);

printf("threads=%d\n", size);

printf("t= %f\n", t2 - t1);

}

MPI\_Finalize();

system("pause");

}