МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное
образовательное учреждение высшего образования
«Самарский национальный исследовательский университет
имени академика С.П. Королева»
(Самарский университет)

Институт информатики, математики и электроники
Факультет информатики
Кафедра технической кибернетики

Отчет по лабораторной работе №3

Параллельные матричные алгоритмы

Вариант №4

Студенты группы 6407 Анурин А.С.
Гринина В.С.
Проверил Головашкин Д.Л.

САМАРА 2018

СОДЕРЖАНИЕ

ЗАДАНИЕ

Вариант №4

Алгоритмы: параллельное перемножение матриц по блочным столбцам на кольце.

1 Теоретические сведения

Пусть даны матрицы A, B, C размерности $n \times n$, будем искать такую матрицу $D = AB + C$. Вычисления будут производиться на процессорном кольце, при этом матрицы будут храниться блочными столбцами.

Для μ-го процессора нахождение μ-го блочного столбца матрицы D выглядит следующим образом:

$$D_\mu = C_\mu + AB_\mu = C_\mu + \sum_{k=1}^{p} A_k B_{k\mu}.$$

Инициализация:

```
n -- размерность матрицы
p -- количество процессоров
r = n/p -- размер блока
mu -- номер текущего процессора
col = (mu - 1) * r + 1 : mu * r
A_loc = A(:, col)
B_loc = B(:, col)
C_loc = C(:, col)
left, right -- номера соседей
```

Алгоритм:

```
for t = 1:p
  send(A_loc, right);
  recv(A_loc, left);
  tau = mu - t;
  if tau <= 0 then tau = tau + p;
  C_loc = C_loc + A_loc * B_loc((tau - 1) * r + 1 : tau * r, :);
End
```

2 Цель работы

Целью данной работы является сравнение времени работы, анализ и реализация алгоритма параллельного перемножения матриц на процессорном кольце с блочными столбцами.

3 Инструментарий

Для вычислений использовалось следующее оборудование:

- Язык программирования C, компилятор mpigcc for the Intel(R) MPI Library 4.1 for Linux, gcc version 4.1.2 20080704 (Red Hat 4.1.2-55)
- Кластер «Сергей Королев».

Данное оборудование удовлетворят всем необходимым аппаратным и программным требованиям, т.к. оно позволяет выполнять высокопроизводительные вычисления с достаточной степенью точности.

4 Параметры эксперимента

Тестирование проводилось на размерах матриц, экспоненциально меняющихся от 2^2 до 2^{11}. Нижней границей размеров матриц взята размерность 4, потому что при меньших значениях время исполнения приближается к точности измерения времени. В качестве верхней границы была взята размерность 2048 как наибольшая размерность, вычисления на которой не дают заснуть.

Параллельный алгоритм будет запускаться в среде Torque с параметрами NODES=2, PPN=2, то есть на двух узлах по два процессора на каждом. Таким образом, в один момент времени будет параллельно выполняться 4 процесса.

5 Теоретическое ожидание

Ожидается, что для небольших размеров матриц время исполнения последовательной программы будет быстрее параллельной из-за того, что параллельная программа будет тратить время на пересылку данных и коммуникацию между процессами. Однако, на больших размерностях ожидается, что параллельная программа будет работать быстрее по очевидным причинам. Максимальным теоретическим ускорением является значение, равное количеству процессов, и ожидается, что на практике этот предел будет достигнут.

$$S = \frac{T_{\text{seq}}}{T_{\text{par}}} = \frac{T_{\text{par}}}{\frac{T_{\text{par}}}{p} + T_{\text{com}}} = \frac{1}{\frac{1}{p} + \frac{T_{\text{com}}}{T_{\text{par}}}},$$

где S — ускорение, T_{seq} — время выполнения последовательного алгоритма, T_{par} — время выполнения параллельного алгоритма, T_{com} — время, затрачиваемое на коммуникацию между процессами, а p — количество процессов.

Можно заметить, что

$T_{\text{com}} \sim n^2, T_{\text{par}} \sim n^3$, поэтому

$$\lim_{n\to\infty} S = \lim_{n\to\infty} \frac{1}{\frac{1}{p} + \frac{C}{n}} = p$$

6 Результаты

Целью данной работы является сравнение работы параллельного и последовательного алгоритмов перемножения матриц.

Приведем времена исполнения параллельного и последовательного алгоритмов на рисунке 1, используя логарифмическую шкалу.

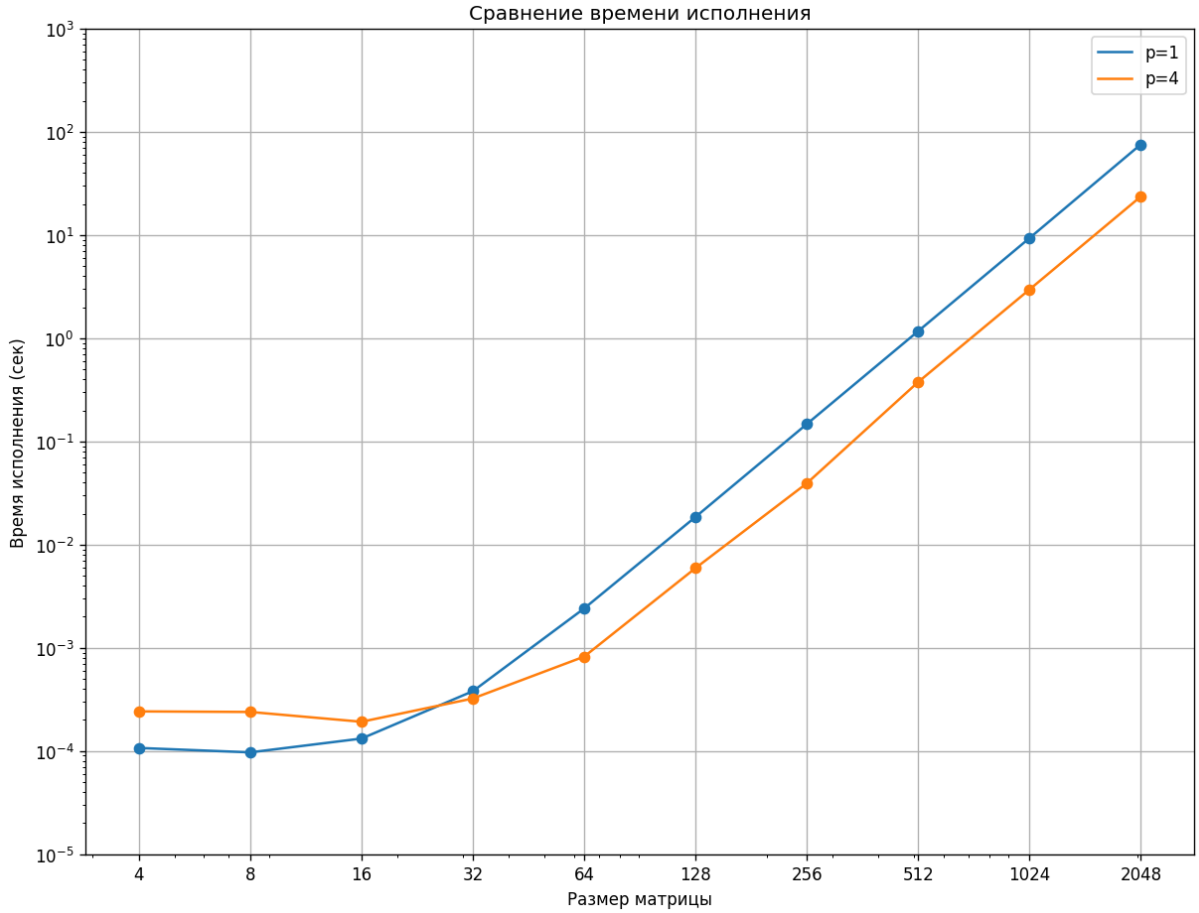

Рисунок 1 — Время исполнения алгоритмов

Можем наблюдать, что для маленьких размеров матриц последовательный алгоритм работает быстрее. Однако с ростом размеров матриц соотношение меняется: параллельный начинает работать быстрее на некоторый константный множитель. Проиллюстрируем непосредственно ускорение на рисунке 2.

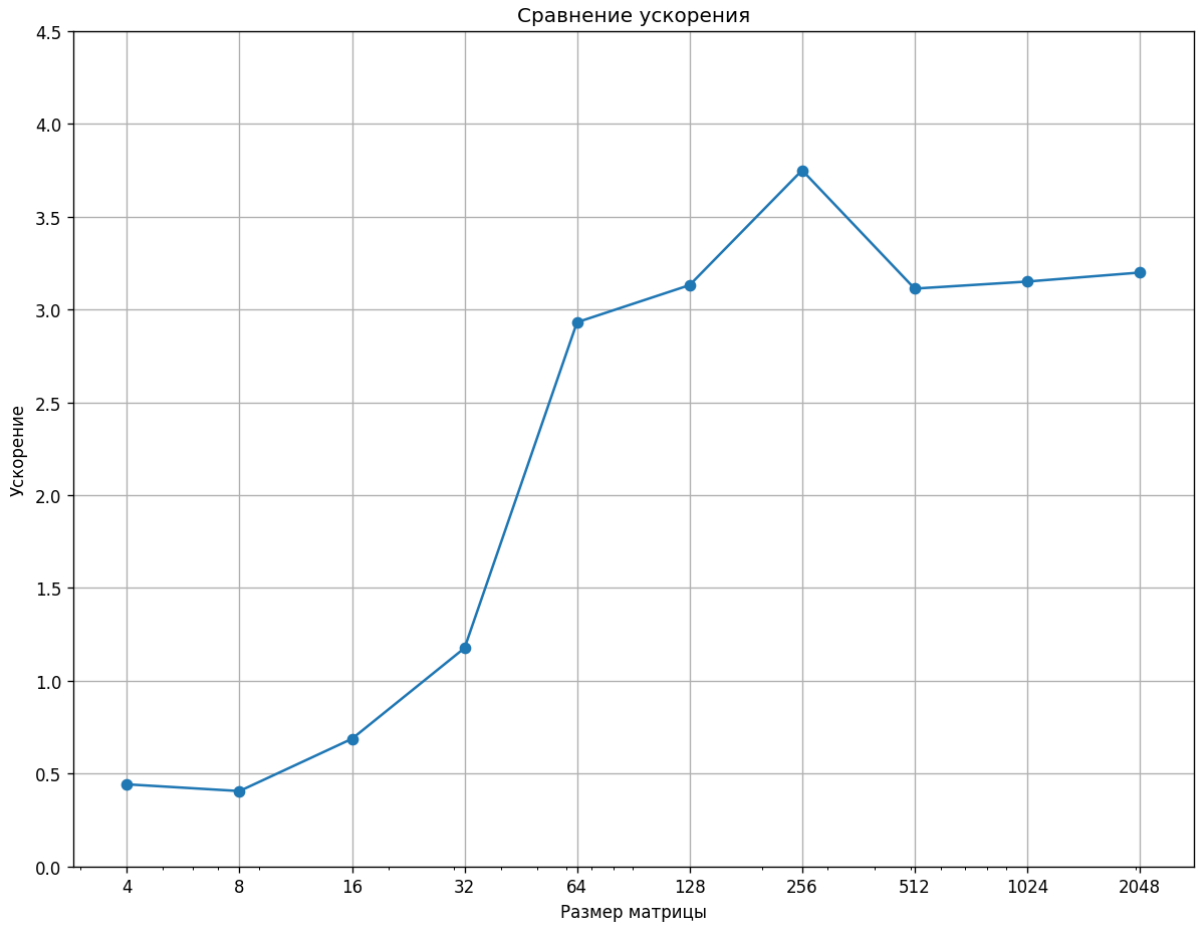

Рисунок 2 — Ускорение параллельного алгоритма

Можно видеть, что ускорение для маленьких матриц меньше единицы. С ростом размера матрицы видно, что ускорение растет и становится чуть больше трех. Чуть правее середины существует пик, на котором достигается максимальное ускорение алгоритма.

При размерах матриц, когда количество вычислений было нетривиальным, ускорение устоялось на отметке в примерно 3.2. Однако максимум ускорения был достигнут на матрицах размера 256 и достигал 3.7.

7 Анализ результатов

Параллельный алгоритм в среднем работал быстрее, чем последовательный. Это связано с тем, что параллельный алгоритм использовал несколько процессоров одновременно, чтобы увеличить производительность.

При небольших размерах матриц последовательный алгоритм был быстрее параллельного, так как не затрачивал ресурсов на коммуникацию процессов и пересылку данных. На больших размерностях параллельный алгоритм показывал значительное ускорение.

Ускорение достигло максимума примерно в середине рассматриваемого диапазона размеров матриц. Это связано с тем, что возник эффект блочного алгоритма в параллельном, то есть при конкретном размере матрицы размер блока наилучшим образом совпал с оптимальным с точки зрения взаимодействия с кэшем. Если рассмотреть чуть меньший размер матрицы, то потенциальный прирост производительности теряется на фоне операций, затрачиваемых на выделение и освобождение памяти и на коммуникацию. Если рассмотреть чуть больший размер матрицы, то блок уже не так хорошо помещается в кэш, что увеличивает количество обменов «кэш — основная память», что сказывается на производительности.

8 Сравнение с теоретическим ожиданием

Как и предполагалось, параллельный алгоритм оказался в целом быстрее последовательного по очевидным причинам. Интересным кажется тот факт, что теоретическое максимальное ускорение не было достигнуто на практике: максимальным практическим ускорением было 3.7 против максимально ожидаемых 4. Однако это можно объяснить несовершенством распараллеливания: некоторая часть времени неизбежно тратится на коммуникацию, инициализацию, синхронизацию и т.д.

Подтвердилась гипотеза о маленьких матрицах: на них последовательный алгоритм был быстрее из-за своей простоты.

Интересный феномен был обнаружен в том, что максимум ускорения был достигнут примерно в середине рассматриваемого диапазона размеров матриц. Этот эффект не был предсказан теоретически.

ЗАКЛЮЧЕНИЕ

В ходе данной работы были продемонстрированы различия между последовательным и параллельным алгоритмами перемножения матриц с блочными столбцами. Прирост производительности, вызванный распараллеливанием вычислений, был близок к теоретическому максимуму.

В целом, практические результаты эксперимента достаточно точно соответствуют теоретическим предсказаниям.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Вильямс В. Алгоритмы. Введение в разработку и анализ [Текст] / Вильямс В. – М. 2006. – 576 с.

2 Голуб Д. Матричные вычисления [Текст] / Голуб Д., Ван Ч. – М.: Мир, 1999. – 548 с.

x

ПРИЛОЖЕНИЕ А КОД ПРОГРАММЫ

```
#include <math.h>
#include <omp.h>
#include <time.h>
#include <stdlib.h>
#include <locale.h>
#include <stdio.h>
#include "mpi.h"
#define ll long long
void print_matrix(int n, int m, double* A) {
  printf("[\n");
  for (int i = 0; i < n; i++) {
    printf("[");
    for (int j = 0; j < m; j++) {
      printf("%f, ", A[i + j * n]);
    }
    printf("]\n");
  }
  printf("]\n");
}
int main(int argc, char* argv[]) {
  MPI_Status Status;
  MPI_Init(&argc, &argv);
  int proc_count;
  MPI_Comm_size(MPI_COMM_WORLD, &proc_count);
  int my_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Request request1;
  MPI_Request request2;
  if (argc < 2) {
    printf("too few arguments\n");
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
    exit(1);
  }
  int n = atoi(argv[1]);
  int p = proc_count;
  int r = n / p;
  int mu = my_rank;
  double* A;
  double* B;
  double* C;
  int block_size = n * r;
  double* A_local = malloc(block_size * sizeof(double));
  double* B_local = malloc(block_size * sizeof(double));
  double* C_local = malloc(block_size * sizeof(double));
  if (my_rank == 0) {
    A = malloc(n * n * sizeof(double));
    B = malloc(n * n * sizeof(double));
    C = malloc(n * n * sizeof(double));
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        A[i * n + j] = (i * n + j) % 100;
        B[i * n + j] = (i * n + j) % 87;
        C[i * n + j] = (i * n + j) % 113;
      }
    }
```

```
  }
  double start_time;
  if (my_rank == 0) {
    start_time = MPI_Wtime();
  }
  MPI_Scatter(      A, block_size, MPI_DOUBLE,
              A_local, block_size, MPI_DOUBLE,
                    0, MPI_COMM_WORLD);
  MPI_Scatter(      B, block_size, MPI_DOUBLE,
              B_local, block_size, MPI_DOUBLE,
                    0, MPI_COMM_WORLD);
  MPI_Scatter(      C, block_size, MPI_DOUBLE,
              C_local, block_size, MPI_DOUBLE,
                    0, MPI_COMM_WORLD);
  double* buffer = malloc(10 * block_size * sizeof(double));
  MPI_Buffer_attach(buffer, 10 * block_size * sizeof(double));
  for (int iter = 0; iter < p; iter++) {
    MPI_Ibsend(A_local, block_size, MPI_DOUBLE,
             (my_rank + 1 + p) % p, 0, MPI_COMM_WORLD, &request1);
    MPI_Recv(A_local, block_size, MPI_DOUBLE,
             (my_rank - 1 + p) % p, 0, MPI_COMM_WORLD, &Status);
    int tau = (mu - iter - 1 + p) % p;
    for (int j = 0; j < r; j++) {
      for (int k = 0; k < r; k++) {
        for (int i = 0; i < n; i++) {
          C_local[i + j * n] +=
            A_local[i + k * n] * B_local[(k + tau * r) + j * n];
        }
      }
    }
    MPI_Wait(&request1, &Status);
  }
  MPI_Gather(C_local, block_size, MPI_DOUBLE,
                  C, block_size, MPI_DOUBLE,
                  0, MPI_COMM_WORLD);
  MPI_Barrier(MPI_COMM_WORLD);
  if (my_rank == 0) {
    double end_time = MPI_Wtime();
    printf("%d,%d,%f\n", n, p, end_time - start_time);
  }
  MPI_Finalize();
  return 0;
}
```