**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ**

**УНИВЕРСИТЕТ**

**«МЭИ»**

**Институт Информационных и Вычислительных технологий**

Кафедра Прикладной Математики и искусственного интеллекта

Лабораторная работа №1

**По курсу** «Параллельное программирование и параллельные системы»

**Выполнил:** Филиппов Е. И.

**Группа:** А-05м-23

**Преподаватель:** Задорин С.А.

Москва, 2024

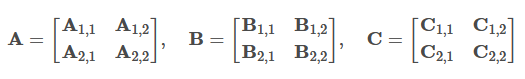
**Постановка задачи**

Реализовать алгоритм умножения матриц методом Штрассена с использованием технологии OpenMP.

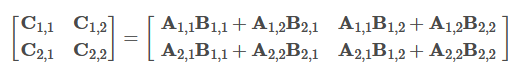
# Алгоритм

Алгоритм умножения матриц Страссена - это первый алгоритм умножения матриц, который асимптотически работает лучше . Данный алгоритм имеет асимптотическую сложность равную .

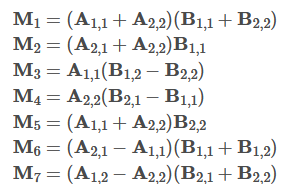
Пусть A и B — две (nxn)-матрицы, причём n — степень числа 2. Тогда можно разбить каждую матрицу A и B на четыре ((n/2)\*(n/2))-матрицы и через них выразить произведение матриц A и B:



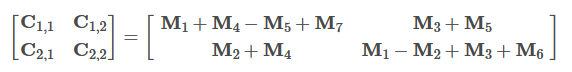
При стандартном подходе матрица C будет иметь такой вид:



Алгоритм Штрассена вводит новые элементы:



Элементы матрицы C будет вычисляться следующим образом:



Таким образом, количество умножений подматриц для приведенной выше формулы становится 7, а количество сложений подматриц для приведенной выше формулы становится 22. Сделано это с расчетом на то, что выполнение операции сложения обходится компьютеру намного дешевле, чем умножение.

Рекурсивный процесс будет продолжатся n раз до тех пор, пока размер матриц не станет достаточно малым, далее используется обычный метод умножения матриц. Это делают из-за того, что алгоритм Штрассена теряет эффективность по сравнению с обычным на малых матрицах в силу большего числа сложений.

# Параллельный алгоритм

При параллельном подходе предлагается вычислять каждый элемент в отдельном потоке. Алгоритм будет состоять из следующих этапов:

1. Задачи m1, m2, m3, m4, m5, m6, m7 выполняются параллельно и независимо друг от друга. Каждая из этих задач выполняет вычисление элементов .
2. Далее происходит ожидание завершения всех созданных задач.
3. Затем происходит объединение результатов вычислений, которые также выполняется параллельно. Каждая из задач c11, c12, c21, c22 отвечает за соответствующую часть результирующей матрицы.
4. После завершения всех задач освобождаются выделенные ресурсы.

Таким образом, параллельное разбиение позволяет ускорить выполнение алгоритма за счет распараллеливания вычислений.

# Конфигурация ПК

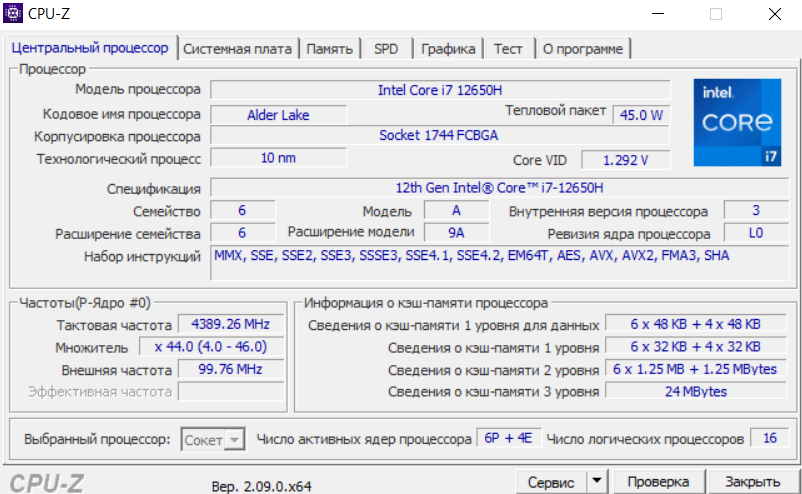


Рис. 1 Информация о процессоре.

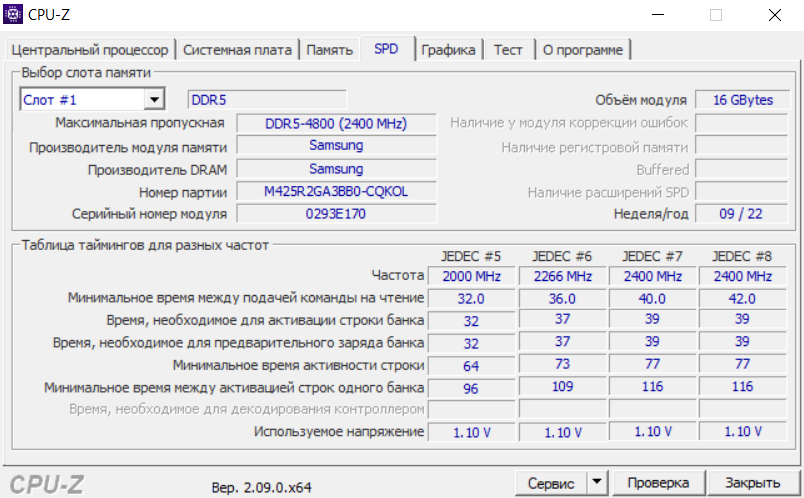


Рис. 2 Информация об оперативной памяти.

# Пример работы программы



# Тесты

При тестировании будем перемножать матрицы размером 512x512, 1024x1024, 2048x2048.

Таблица 1. Зависимость времени работы (в секундах) от числа потоков

|  |  |  |  |
| --- | --- | --- | --- |
| Размерность матрицы  Число потоков | 512x512 | 1024x1024 | 2048x2048 |
| 1 | 0.312 s | 2.153 s | 14.837 s |
| 2 | 0.155 s | 0.997 s | 7.073 s |
| 4 | 0.103 s | 0.682 s | 4.342 s |
| 8 | 0.103 s | 0.644 s | 4.242 s |
| 16 | 0.138 s | 0.881 s | 6.182 s |
| 32 | 0.163 s | 1.048 s | 7.669 s |
| 64 | 0.186 s | 1.148 s | 8.666 s |
| 128 | 0.223 s | 1.237 s | 10.527 s |
| 256 | 0.311 s | 1.824 s | 14.747 s |

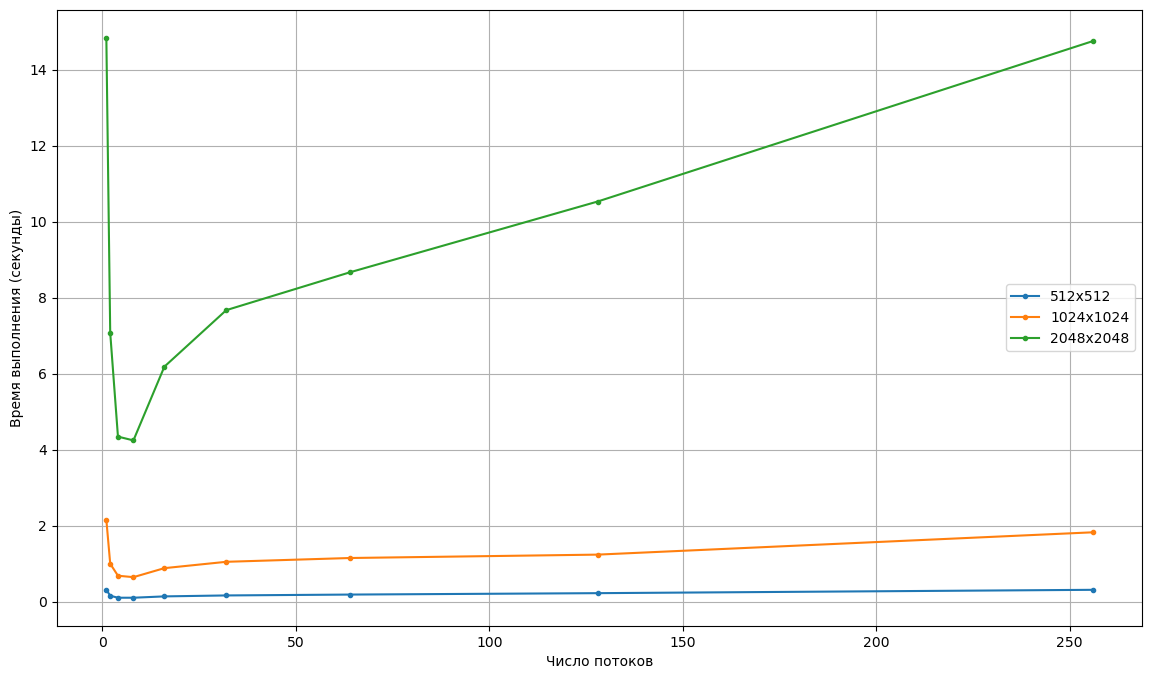


Рис. 3 График зависимости времени выполнения от числа потоков

Таблица 2. Зависимость коэффициента ускорения от числа потоков

|  |  |  |  |
| --- | --- | --- | --- |
| Размерность матрицы  Число потоков | 512x512 | 1024x1024 | 2048x2048 |
| 1 | 1.00 | 1.00 | 1.00 |
| 2 | 2.01 | 2.16 | 2.10 |
| 4 | 3.03 | 3.16 | 3.42 |
| 8 | 3.03 | 3.34 | 3.50 |
| 16 | 2.26 | 2.44 | 2.40 |
| 32 | 1.91 | 2.05 | 1.93 |
| 64 | 1.68 | 1.88 | 1.71 |
| 128 | 1.40 | 1.74 | 1.41 |
| 256 | 1.00 | 1.18 | 1.01 |

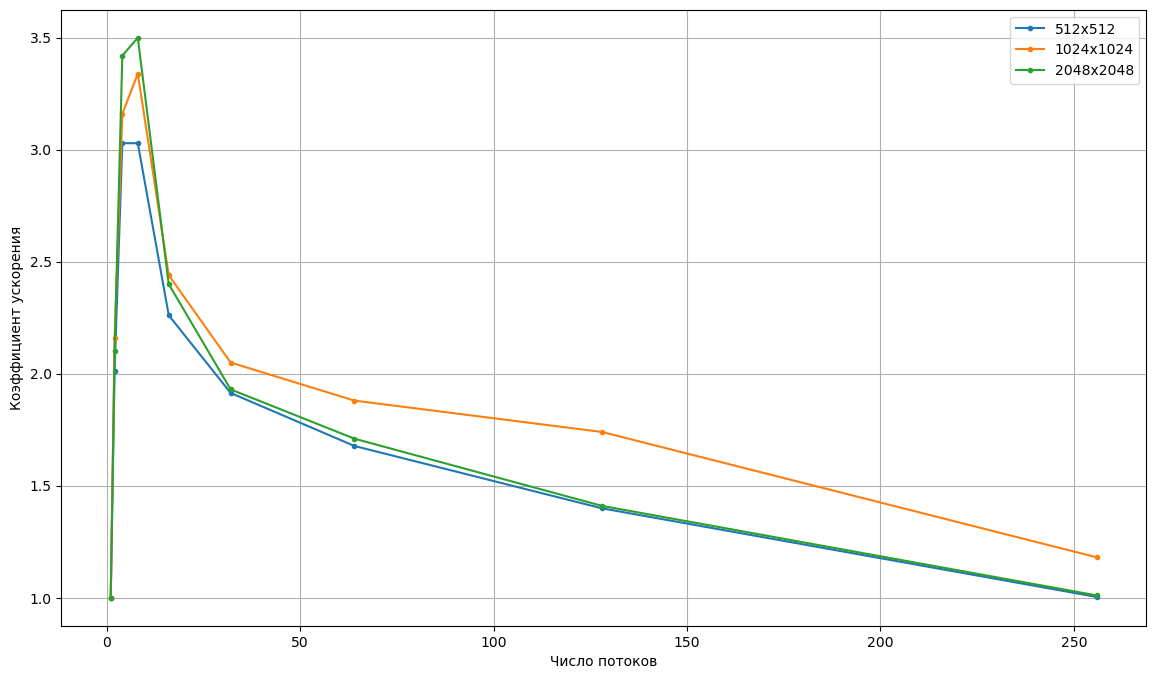


Рис. 4 График зависимости коэффициента ускорения от числа потоков

# Вывод

Грамотный подбор числа потоков позволяет нам достигать наилучшего соотношения выделяемых ресурсов к скорости вычисления. Из графиков видно, что на 8 потоках заканчивается рост ускорения, после чего начинается планомерное его падение. Таким образом, на личном ПК для решения данной задачи оптимальным числом потоков является 8. При параллельном решении задачи удалось ускорить выполнение алгоритма примерно в 3 раза.

# Программный код

OpenMP:

|  |
| --- |
| #include <iostream>  #include <omp.h>  #include <ctime>  #include <cstdlib>  #include <iomanip>  #include <fstream>  using namespace std;  void print(int n, int \*\*mat)  {      for (int i = 0; i < n; i++)      {          for (int j = 0; j < n; j++)          {              cout << mat[i][j] << " ";          }          cout << endl;      }      cout << endl;  }  int \*\*allocateMatrix(int n)  {      int \*data = (int \*)malloc(n \* n \* sizeof(int));      int \*\*array = (int \*\*)malloc(n \* sizeof(int \*));      for (int i = 0; i < n; i++)      {          array[i] = &(data[n \* i]);      }      return array;  }  void fillMatrix(int n, int \*\*&mat)  {      for (int i = 0; i < n; i++)      {          for (int j = 0; j < n; j++)          {              mat[i][j] = rand() % 5;          }      }  }  void freeMatrix(int n, int \*\*mat)  {      free(mat[0]);      free(mat);  }  int \*\*naive(int n, int \*\*mat1, int \*\*mat2)  {      int \*\*prod = allocateMatrix(n);      int i, j;  #pragma omp parallel for collapse(2)      for (i = 0; i < n; i++)      {          for (j = 0; j < n; j++)          {              prod[i][j] = 0;              for (int k = 0; k < n; k++)              {                  prod[i][j] += mat1[i][k] \* mat2[k][j];              }          }      }      return prod;  }  int \*\*getSlice(int n, int \*\*mat, int offseti, int offsetj)  {      int m = n / 2;      int \*\*slice = allocateMatrix(m);      for (int i = 0; i < m; i++)      {          for (int j = 0; j < m; j++)          {              slice[i][j] = mat[offseti + i][offsetj + j];          }      }      return slice;  }  int \*\*addMatrices(int n, int \*\*mat1, int \*\*mat2, bool add)  {      int \*\*result = allocateMatrix(n);      for (int i = 0; i < n; i++)      {          for (int j = 0; j < n; j++)          {              if (add)                  result[i][j] = mat1[i][j] + mat2[i][j];              else                  result[i][j] = mat1[i][j] - mat2[i][j];          }      }      return result;  }  int \*\*combineMatrices(int m, int \*\*c11, int \*\*c12, int \*\*c21, int \*\*c22)  {      int n = 2 \* m;      int \*\*result = allocateMatrix(n);      for (int i = 0; i < n; i++)      {          for (int j = 0; j < n; j++)          {              if (i < m && j < m)                  result[i][j] = c11[i][j];              else if (i < m)                  result[i][j] = c12[i][j - m];              else if (j < m)                  result[i][j] = c21[i - m][j];              else                  result[i][j] = c22[i - m][j - m];          }      }      return result;  }  int \*\*strassen(int n, int \*\*mat1, int \*\*mat2)  {      if (n <= 32)      {          return naive(n, mat1, mat2);      }      int m = n / 2;      int \*\*a = getSlice(n, mat1, 0, 0);      int \*\*b = getSlice(n, mat1, 0, m);      int \*\*c = getSlice(n, mat1, m, 0);      int \*\*d = getSlice(n, mat1, m, m);      int \*\*e = getSlice(n, mat2, 0, 0);      int \*\*f = getSlice(n, mat2, 0, m);      int \*\*g = getSlice(n, mat2, m, 0);      int \*\*h = getSlice(n, mat2, m, m);      int \*\*m1;  #pragma omp task shared(m1)      {          int \*\*bds = addMatrices(m, b, d, false);          int \*\*gha = addMatrices(m, g, h, true);          m1 = strassen(m, bds, gha);          freeMatrix(m, bds);          freeMatrix(m, gha);      }      int \*\*m2;  #pragma omp task shared(m2)      {          int \*\*ada = addMatrices(m, a, d, true);          int \*\*eha = addMatrices(m, e, h, true);          m2 = strassen(m, ada, eha);          freeMatrix(m, ada);          freeMatrix(m, eha);      }      int \*\*m3;  #pragma omp task shared(m3)      {          int \*\*acs = addMatrices(m, a, c, false);          int \*\*efa = addMatrices(m, e, f, true);          m3 = strassen(m, acs, efa);          freeMatrix(m, acs);          freeMatrix(m, efa);      }      int \*\*m4;  #pragma omp task shared(m4)      {          int \*\*aba = addMatrices(m, a, b, true);          m4 = strassen(m, aba, h);          freeMatrix(m, aba);      }      int \*\*m5;  #pragma omp task shared(m5)      {          int \*\*fhs = addMatrices(m, f, h, false);          m5 = strassen(m, a, fhs);          freeMatrix(m, fhs);      }      int \*\*m6;  #pragma omp task shared(m6)      {          int \*\*ges = addMatrices(m, g, e, false);          m6 = strassen(m, d, ges);          freeMatrix(m, ges);      }      int \*\*m7;  #pragma omp task shared(m7)      {          int \*\*cda = addMatrices(m, c, d, true);          m7 = strassen(m, cda, e);          freeMatrix(m, cda);      }  #pragma omp taskwait      freeMatrix(m, a);      freeMatrix(m, b);      freeMatrix(m, c);      freeMatrix(m, d);      freeMatrix(m, e);      freeMatrix(m, f);      freeMatrix(m, g);      freeMatrix(m, h);      int \*\*c11;  #pragma omp task shared(c11)      {          int \*\*s1s2a = addMatrices(m, m1, m2, true);          int \*\*s6s4s = addMatrices(m, m6, m4, false);          c11 = addMatrices(m, s1s2a, s6s4s, true);          freeMatrix(m, s1s2a);          freeMatrix(m, s6s4s);      }      int \*\*c12;  #pragma omp task shared(c12)      {          c12 = addMatrices(m, m4, m5, true);      }      int \*\*c21;  #pragma omp task shared(c21)      {          c21 = addMatrices(m, m6, m7, true);      }      int \*\*c22;  #pragma omp task shared(c22)      {          int \*\*s2s3s = addMatrices(m, m2, m3, false);          int \*\*s5s7s = addMatrices(m, m5, m7, false);          c22 = addMatrices(m, s2s3s, s5s7s, true);          freeMatrix(m, s2s3s);          freeMatrix(m, s5s7s);      }  #pragma omp taskwait      freeMatrix(m, m1);      freeMatrix(m, m2);      freeMatrix(m, m3);      freeMatrix(m, m4);      freeMatrix(m, m5);      freeMatrix(m, m6);      freeMatrix(m, m7);      int \*\*prod = combineMatrices(m, c11, c12, c21, c22);      freeMatrix(m, c11);      freeMatrix(m, c12);      freeMatrix(m, c21);      freeMatrix(m, c22);      return prod;  }  bool check(int n, int \*\*prod1, int \*\*prod2)  {      for (int i = 0; i < n; i++)      {          for (int j = 0; j < n; j++)          {              if (prod1[i][j] != prod2[i][j])                  return false;          }      }      return true;  }  void printMatrix(int n, int \*\* mat, ostream& stream)  {      for(int i = 0; i < n; i++)      {          for(int j = 0; j < n; j++)          {              stream << mat[i][j] << " ";          }          stream << endl;      }      stream << endl;  }  void outputToFile(int n, int \*\*a, int \*\*b, int \*\*result)  {      ofstream out;      out.open("result.txt");      if (out.is\_open())      {          out << "Matrix A:" << endl;          printMatrix(n, a, out);          out << "Matrix B:" << endl;          printMatrix(n, a, out);          out << "Multiplication Result:" << endl;          printMatrix(n, a, out);      }      out.close();      std::cout << "File has been written" << std::endl;  }  int main()  {      int n;      cout << "\nEnter matrix Dimension: ";      cin >> n;      int \*\*mat1 = allocateMatrix(n);      fillMatrix(n, mat1);      int \*\*mat2 = allocateMatrix(n);      fillMatrix(n, mat2);      int threadCounts[] = {1, 2, 4, 8, 12, 16, 32, 64, 128, 256};      int \*\*prod;      for (int threads : threadCounts)      {          double startParStrassen = omp\_get\_wtime();          omp\_set\_num\_threads(threads);          #pragma omp parallel          {          #pragma omp single              {                  prod = strassen(n, mat1, mat2);              }          }          double endParStrassen = omp\_get\_wtime();          cout << "\nParallel Strassen (" << threads << " threads) Runtime: ";          cout << setprecision(5) << endParStrassen - startParStrassen << endl;          cout << endl;      }      outputToFile(n, mat1, mat2, prod);      return 0;  } |