**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ**

**УНИВЕРСИТЕТ**

**«МЭИ»**

**Институт Информационных и Вычислительных технологий**

Кафедра Прикладной Математики и искусственного интеллекта

Лабораторная работа №3

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

**Выполнил:** Филиппов Е. И.

**Группа:** А-05м-23

**Преподаватель:** Задорин С.А.

Москва, 2024

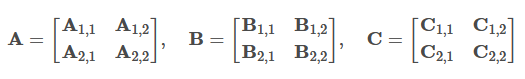
**Постановка задачи**

Реализовать алгоритм умножения матриц методом Штрассена с использованием технологии C++/MPI.

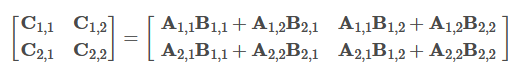
# Алгоритм

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

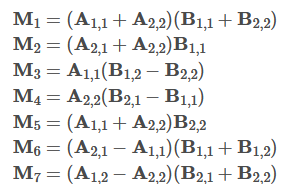
Пусть A и B — две (nxn)-матрицы, причём n — степень числа 2. Тогда можно разбить каждую матрицу A и B на четыре ((n/2)\*(n/2))-матрицы и через них выразить произведение матриц A и B:



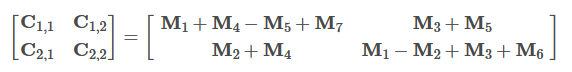
При стандартном подходе матрица C будет иметь такой вид:



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. Затем происходит объединение результатов вычислений в потоке с рангом 0. Каждая из переменных c11, c12, c21, c22 отвечает за соответствующую часть результирующей матрицы.

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

# Описание функций

*1. Strassen(int n, int\*\* matrix1, int\*\* matrix2, int\*\*& result, int rank, int size)*

Эта функция реализует алгоритм умножения матриц Стразена в параллельном режиме с использованием MPI. Она принимает в качестве параметров размер матриц n, две входные матрицы matrix1 и matrix2, ссылку на матрицу результата result, ранг текущего процесса rank и общее количество процессов size. Функция делит матрицы на меньшие подматрицы, вычисляет произведения этих подматриц с помощью рекурсивных вызовов самой себя и объединяет результаты для формирования окончательной матрицы произведения.

*2. StrassenImp(int n, int\*\* matrix1, int\*\* matrix2)*

Эта функция является последовательной реализацией алгоритма умножения матриц Стразена. Она принимает в качестве параметров размер матриц n и две входные матрицы matrix1 и matrix2. Функция делит матрицы на меньшие подматрицы, вычисляет произведения этих подматриц с помощью рекурсивных вызовов самой себя и объединяет результаты для формирования окончательной матрицы произведения.

*3. CreateMatrix(int n)*

Эта функция создает новую матрицу размера n x n и возвращает указатель на нее.

*4. FillMatrix(int n, int value, int\*\*& matrix)*

Эта функция заполняет заданную матрицу размера n x n определенным значением value.

*5. DeleteMatrix(int n, int\*\* matrix)*

Эта функция удаляет заданную матрицу размера n x n.

*6. SliceMatrix(int n, int\*\* matrix, int offseti, int offsetj)*

Эта функция извлекает подматрицу из заданной матрицы размера n x n, начиная с строки offseti и столбца offsetj, и возвращает указатель на подматрицу.

*7. SumMatrix(int n, int\*\* matrix1, int\*\* matrix2)*

Эта функция складывает две матрицы размера n x n элемент за элементом и возвращает указатель на результирующую матрицу.

*8. SubMatrix(int n, int\*\* matrix1, int\*\* matrix2)*

Эта функция вычитает две матрицы размера n x n элемент за элементом и возвращает указатель на результирующую матрицу.

*9. MulMatrix(int n, int\*\* matrix1, int\*\* matrix2)*

Эта функция умножает две матрицы размера n x n с помощью стандартного алгоритма умножения матриц и возвращает указатель на результирующую матрицу.

*10. CombineMatrix(int m, int\*\* c11, int\*\* c12, int\*\* c21, int\*\* c22)*

Эта функция объединяет четыре матрицы размера m x m в одну матрицу размера 2m x 2m, где c11, c12, c21 и c22 - это верхняя левая, верхняя правая, нижняя левая и нижняя правая подматрицы соответственно.

*11. PrintMatrix(int n, int\*\* matrix)*

Эта функция печатает заданную матрицу размера n x n в консоль.

# Конфигурация ПК

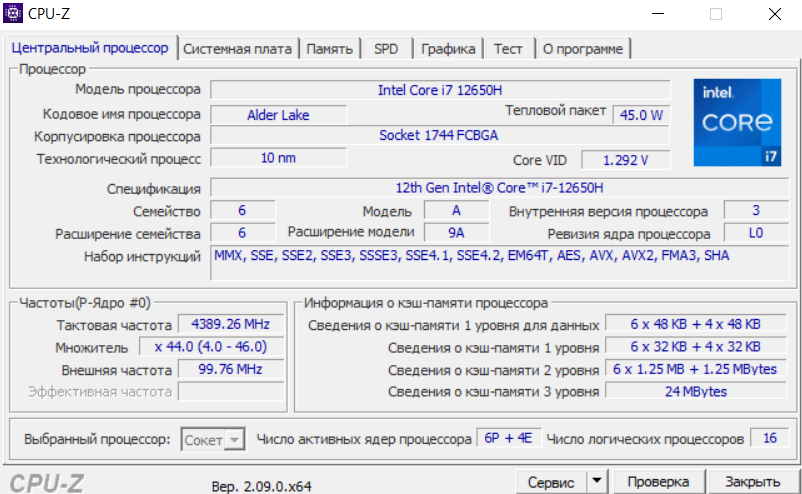


Рис. 1 Информация о процессоре.

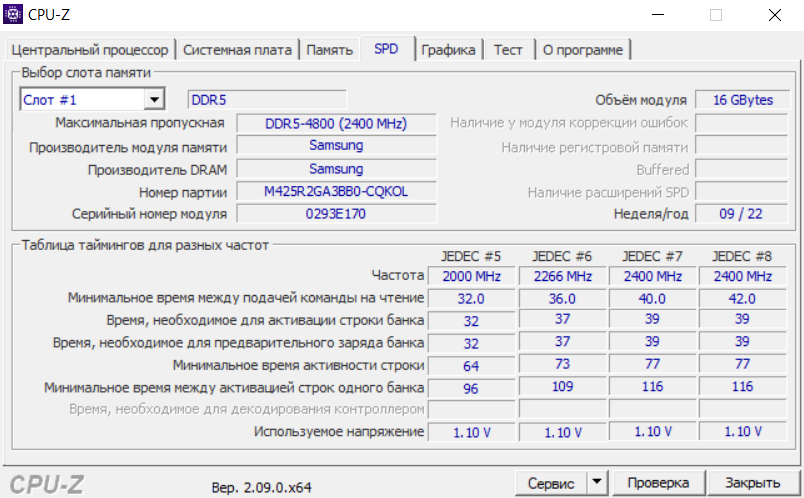
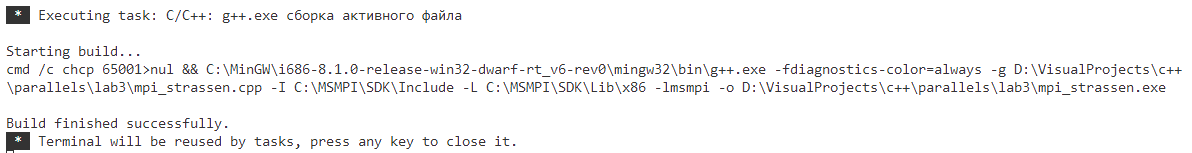


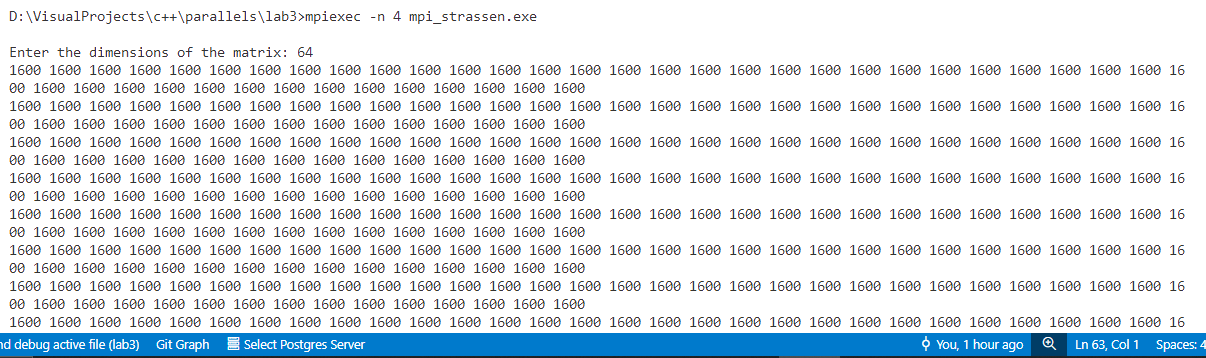
Рис. 2 Информация об оперативной памяти.

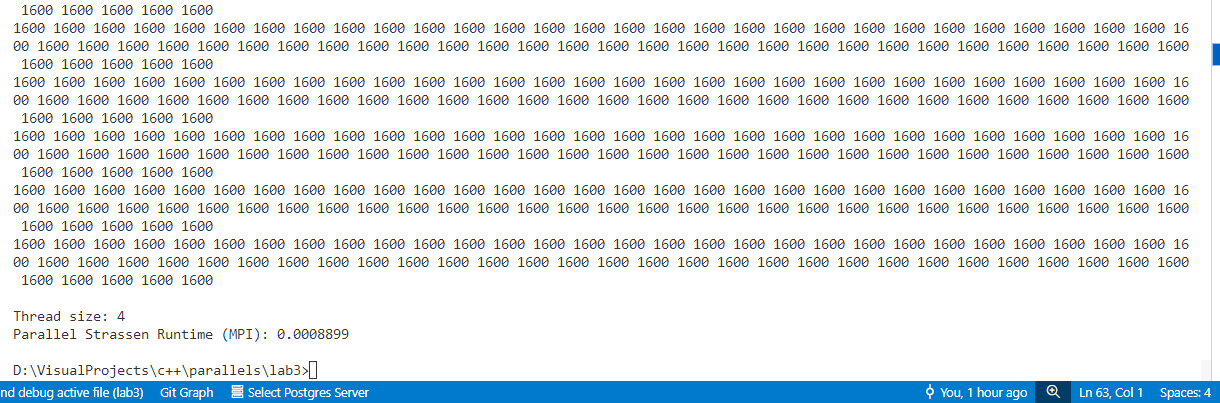
# Пример работы программы

*Сборка программы*



*Запуск 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 | 512х512 | 1024х1024 | 2048х2048 |
| 1 | 0.211 | 1.498 | 10.528 |
| 2 | 0.212 | 1.501 | 10.521 |
| 3 | 0.129 | 0.873 | 6.079 |
| 4 | 0.107 | 0.684 | 4.802 |
| 8 | 0.066 | 0.326 | 2.156 |
| 10 | 0.056 | 0.308 | 2.15 |
| 12 | 0.054 | 0.307 | 2.17 |
| 16 | 0.057 | 0.327 | 2.14 |

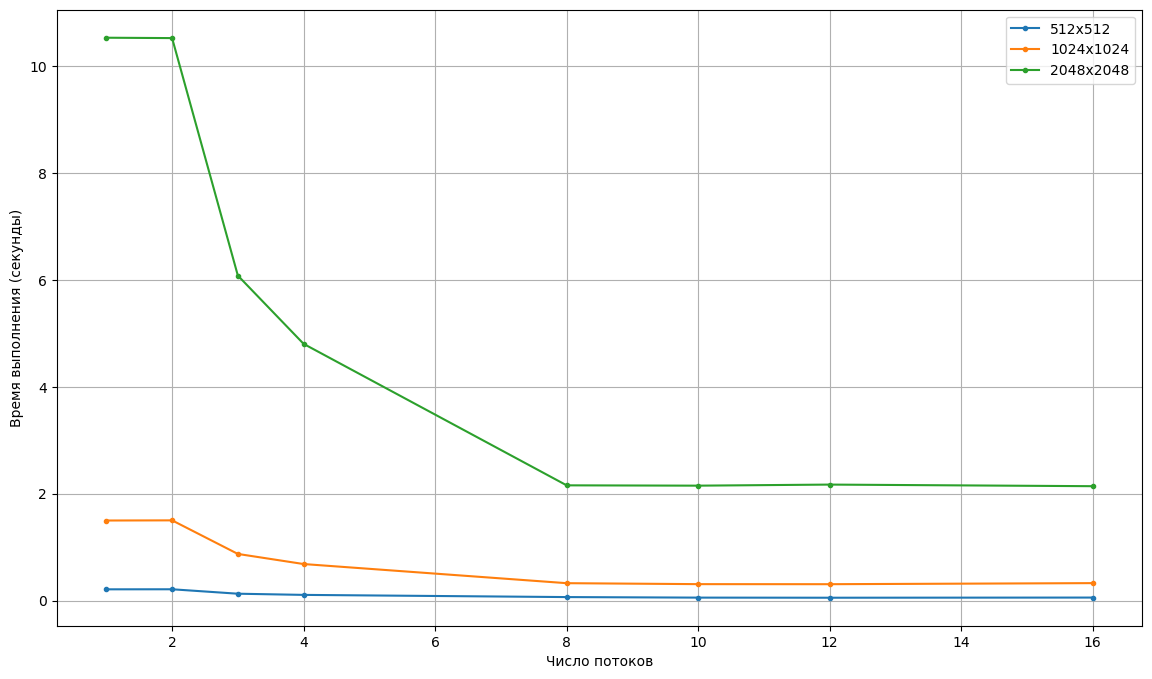


Рис. 3 График зависимости времени выполнения от числа потоков

Таблица 2. Зависимость коэффициента ускорения от числа потоков

|  |  |  |  |
| --- | --- | --- | --- |
| Размерность матрицы  Число потоков | 512х512 | 1024х1024 | 2048х2048 |
| 1 | 1.000 | 1.000 | 1.000 |
| 2 | 0.995 | 0.998 | 1.001 |
| 3 | 1.636 | 1.716 | 1.732 |
| 4 | 1.972 | 2.190 | 2.192 |
| 8 | 3.197 | 4.595 | 4.883 |
| 10 | 3.768 | 4.864 | 4.897 |
| 12 | 3.907 | 4.879 | 4.852 |
| 16 | 3.702 | 4.581 | 4.920 |

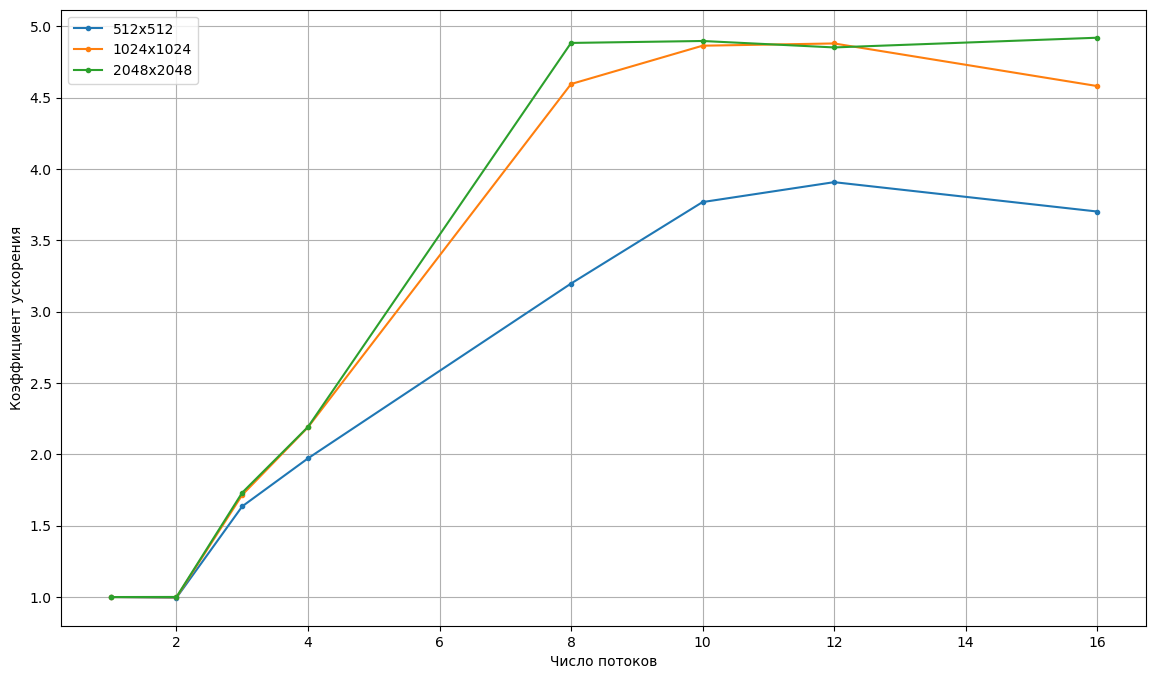


Рис. 4 График зависимости коэффициента ускорения от числа потоков

# Вывод

Грамотный подбор числа потоков позволяет нам достигать наилучшего соотношения выделяемых ресурсов к скорости вычисления. Прежде всего отметим, что увеличение числа потоков с 1 до 2 не принесло прироста производительности. Это связано с тем, что в данной реализации нулевой поток используется только для аккумулирования всех результатов от других потоков, а остальные используются непосредственно для расчета параметров. Также из графиков видно, что на 8 потоках, как и в реализации на Open MP, заканчивается рост ускорения, после чего на графике наблюдается плато. Это связано с особенностью реализации алгоритма. Дело в том, что алгоритм может вычислять независимо в отдельном потоке только 7 параметров, поэтому если число потоков составляет больше 8(1 поток - для аккумуляции расчетов и 7 потоков - для расчета каждого параметра), то оставшиеся потоки фактически будут простаивать. Таким образом, на личном ПК для решения данной задачи оптимальным числом потоков является 8. При параллельном решении задачи удалось ускорить выполнение алгоритма примерно в 4.5 раза. Показатели ускорения превысили результаты предыдущих работ, что делает подход использования технологии MPI на данный момент наиболее предпочтительным.

# Ссылки

https://github.com/FoxJefisto/lab3 - репозиторий с реализацией алгоритма

# Программный код

C++/MPI:

|  |
| --- |
| #include <mpi.h>  #include <iostream>  #include <ctime>  #include <cstdlib>  #include <iomanip>  #include <fstream>  using namespace std;  int\*\* Strassen(int n, int\*\* matrix1, int\*\* matrix2, int\*\*& result, int rank, int size);  int\*\* StrassenImp(int n, int\*\* matrix1, int\*\* matrix2);  int\*\* CreateMatrix(int n);  void FillMatrix(int n, int value, int\*\*& matrix);  void DeleteMatrix(int n, int\*\* matrix);  int\*\* SliceMatrix(int n, int\*\* matrix, int offseti, int offsetj);  int\*\* SumMatrix(int n, int\*\* matrix1, int\*\* matrix2);  int\*\* SubMatrix(int n, int\*\* matrix1, int\*\* matrix2);  int\*\* MulMatrix(int n, int\*\* matrix1, int\*\* matrix2);  int\*\* CombineMatrix(int m, int\*\* c11, int\*\* c12, int\*\* c21, int\*\* c22);  void PrintMatrix(int n, int\*\* matrix);  int main(int argc, char\* argv[])  {      int rank;      int size;      MPI\_Init(&argc, &argv);      MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);      MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);      int n;      if (rank == 0)      {          cout << endl;          cout << "Enter the dimensions of the matrix: ";          cin >> n;      }      MPI\_Barrier(MPI\_COMM\_WORLD);      MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);      int\*\* matrix1 = CreateMatrix(n);      int\*\* matrix2 = CreateMatrix(n);      if (rank == 0)      {          FillMatrix(n, 5, matrix1);          FillMatrix(n, 2, matrix2);      }      MPI\_Bcast(&(matrix1[0][0]), n \* n, MPI\_INT, 0, MPI\_COMM\_WORLD);      MPI\_Bcast(&(matrix2[0][0]), n \* n, MPI\_INT, 0, MPI\_COMM\_WORLD);      double startTime = MPI\_Wtime();      int\*\* result;      if(size == 1)          result = StrassenImp(n, matrix1, matrix2);      else          result = Strassen(n, matrix1, matrix2, result, rank, size);      double endTime = MPI\_Wtime();      if (rank == 0)      {          // PrintMatrix(n, result);          cout << "Thread size: " << size << "\nParallel Strassen Runtime (MPI): "          << setprecision(5) << endTime - startTime << endl;      }      MPI\_Finalize();      return 0;  }  int\*\* Strassen(int n, int\*\* matrix1, int\*\* matrix2, int\*\*& result, int rank, int size)  {      if (n == 1)      {          result = CreateMatrix(1);          result[0][0] = matrix1[0][0] \* matrix2[0][0];      }      int m = n / 2;      int\*\* a = SliceMatrix(n, matrix1, 0, 0);      int\*\* b = SliceMatrix(n, matrix1, 0, m);      int\*\* c = SliceMatrix(n, matrix1, m, 0);      int\*\* d = SliceMatrix(n, matrix1, m, m);      int\*\* e = SliceMatrix(n, matrix2, 0, 0);      int\*\* f = SliceMatrix(n, matrix2, 0, m);      int\*\* g = SliceMatrix(n, matrix2, m, 0);      int\*\* h = SliceMatrix(n, matrix2, m, m);      int\*\* s1 = CreateMatrix(m);      int\*\* s2 = CreateMatrix(m);      int\*\* s3 = CreateMatrix(m);      int\*\* s4 = CreateMatrix(m);      int\*\* s5 = CreateMatrix(m);      int\*\* s6 = CreateMatrix(m);      int\*\* s7 = CreateMatrix(m);      int size1 = size - 1;      if (rank == 0)      {          int i = 0;          int j = 1;          for(int\*\* s : {s1, s2, s3, s4, s5, s6, s7})          {              i = i % size1 + 1;              MPI\_Recv(&(s[0][0]), m \* m, MPI\_INT, i, j, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);              j++;          }      }      if (rank == (0 % size1 + 1))      {          int\*\* bds = SubMatrix(m, b, d);          int\*\* gha = SumMatrix(m, g, h);          s1 = StrassenImp(m, bds, gha);          DeleteMatrix(m, bds);          DeleteMatrix(m, gha);          MPI\_Send(&(s1[0][0]), m \* m, MPI\_INT, 0, 1, MPI\_COMM\_WORLD);      }      if (rank == (1 % size1 + 1))      {          int\*\* ada = SumMatrix(m, a, d);          int\*\* eha = SumMatrix(m, e, h);          s2 = StrassenImp(m, ada, eha);          DeleteMatrix(m, ada);          DeleteMatrix(m, eha);          MPI\_Send(&(s2[0][0]), m \* m, MPI\_INT, 0, 2, MPI\_COMM\_WORLD);      }      if (rank == (2 % size1 + 1))      {          int\*\* acs = SubMatrix(m, a, c);          int\*\* efa = SumMatrix(m, e, f);          s3 = StrassenImp(m, acs, efa);          DeleteMatrix(m, acs);          DeleteMatrix(m, efa);          MPI\_Send(&(s3[0][0]), m \* m, MPI\_INT, 0, 3, MPI\_COMM\_WORLD);      }      if (rank == (3 % size1 + 1))      {          int\*\* aba = SumMatrix(m, a, b);          s4 = StrassenImp(m, aba, h);          DeleteMatrix(m, aba);          MPI\_Send(&(s4[0][0]), m \* m, MPI\_INT, 0, 4, MPI\_COMM\_WORLD);      }      DeleteMatrix(m, b);      if (rank == (4 % size1 + 1))      {          int\*\* fhs = SubMatrix(m, f, h);          s5 = StrassenImp(m, a, fhs);          DeleteMatrix(m, fhs);          MPI\_Send(&(s5[0][0]), m \* m, MPI\_INT, 0, 5, MPI\_COMM\_WORLD);      }      DeleteMatrix(m, a);      DeleteMatrix(m, f);      DeleteMatrix(m, h);      if (rank == (5 % size1 + 1))      {          int\*\* ges = SubMatrix(m, g, e);          s6 = StrassenImp(m, d, ges);          DeleteMatrix(m, ges);          MPI\_Send(&(s6[0][0]), m \* m, MPI\_INT, 0, 6, MPI\_COMM\_WORLD);      }      DeleteMatrix(m, g);      if (rank == (6 % size1 + 1))      {          int\*\* cda = SumMatrix(m, c, d);          s7 = StrassenImp(m, cda, e);          DeleteMatrix(m, cda);          MPI\_Send(&(s7[0][0]), m \* m, MPI\_INT, 0, 7, MPI\_COMM\_WORLD);      }      DeleteMatrix(m, c);      DeleteMatrix(m, d);      DeleteMatrix(m, e);      MPI\_Barrier(MPI\_COMM\_WORLD);      if (rank == 0)      {          int\*\* s1s2a = SumMatrix(m, s1, s2);          int\*\* s6s4s = SubMatrix(m, s6, s4);          int\*\* c11 = SumMatrix(m, s1s2a, s6s4s);          DeleteMatrix(m, s1s2a);          DeleteMatrix(m, s6s4s);          int\*\* c12 = SumMatrix(m, s4, s5);          int\*\* c21 = SumMatrix(m, s6, s7);          int\*\* s2s3s = SubMatrix(m, s2, s3);          int\*\* s5s7s = SubMatrix(m, s5, s7);          int\*\* c22 = SumMatrix(m, s2s3s, s5s7s);          DeleteMatrix(m, s2s3s);          DeleteMatrix(m, s5s7s);          result = CombineMatrix(m, c11, c12, c21, c22);          DeleteMatrix(m, c11);          DeleteMatrix(m, c12);          DeleteMatrix(m, c21);          DeleteMatrix(m, c22);          return result;      }      DeleteMatrix(m, s1);      DeleteMatrix(m, s2);      DeleteMatrix(m, s3);      DeleteMatrix(m, s4);      DeleteMatrix(m, s5);      DeleteMatrix(m, s6);      DeleteMatrix(m, s7);      return nullptr;  }  int\*\* StrassenImp(int n, int\*\* matrix1, int\*\* matrix2)  {      if (n <= 32)      {          return MulMatrix(n, matrix1, matrix2);      }      int m = n / 2;      int\*\* a = SliceMatrix(n, matrix1, 0, 0);      int\*\* b = SliceMatrix(n, matrix1, 0, m);      int\*\* c = SliceMatrix(n, matrix1, m, 0);      int\*\* d = SliceMatrix(n, matrix1, m, m);      int\*\* e = SliceMatrix(n, matrix2, 0, 0);      int\*\* f = SliceMatrix(n, matrix2, 0, m);      int\*\* g = SliceMatrix(n, matrix2, m, 0);      int\*\* h = SliceMatrix(n, matrix2, m, m);      int\*\* bds = SubMatrix(m, b, d);      int\*\* gha = SumMatrix(m, g, h);      int\*\* s1 = StrassenImp(m, bds, gha);      DeleteMatrix(m, bds);      DeleteMatrix(m, gha);      int\*\* ada = SumMatrix(m, a, d);      int\*\* eha = SumMatrix(m, e, h);      int\*\* s2 = StrassenImp(m, ada, eha);      DeleteMatrix(m, ada);      DeleteMatrix(m, eha);      int\*\* acs = SubMatrix(m, a, c);      int\*\* efa = SumMatrix(m, e, f);      int\*\* s3 = StrassenImp(m, acs, efa);      DeleteMatrix(m, acs);      DeleteMatrix(m, efa);      int\*\* aba = SumMatrix(m, a, b);      int\*\* s4 = StrassenImp(m, aba, h);      DeleteMatrix(m, aba);      DeleteMatrix(m, b);      int\*\* fhs = SubMatrix(m, f, h);      int\*\* s5 = StrassenImp(m, a, fhs);      DeleteMatrix(m, fhs);      DeleteMatrix(m, a);      DeleteMatrix(m, f);      DeleteMatrix(m, h);      int\*\* ges = SubMatrix(m, g, e);      int\*\* s6 = StrassenImp(m, d, ges);      DeleteMatrix(m, ges);      DeleteMatrix(m, g);      int\*\* cda = SumMatrix(m, c, d);      int\*\* s7 = StrassenImp(m, cda, e);      DeleteMatrix(m, cda);      DeleteMatrix(m, c);      DeleteMatrix(m, d);      DeleteMatrix(m, e);      int\*\* s1s2a = SumMatrix(m, s1, s2);      int\*\* s6s4s = SubMatrix(m, s6, s4);      int\*\* c11 = SumMatrix(m, s1s2a, s6s4s);      DeleteMatrix(m, s1s2a);      DeleteMatrix(m, s6s4s);      DeleteMatrix(m, s1);      int\*\* c12 = SumMatrix(m, s4, s5);      DeleteMatrix(m, s4);      int\*\* c21 = SumMatrix(m, s6, s7);      DeleteMatrix(m, s6);      int\*\* s2s3s = SubMatrix(m, s2, s3);      int\*\* s5s7s = SubMatrix(m, s5, s7);      int\*\* c22 = SumMatrix(m, s2s3s, s5s7s);      DeleteMatrix(m, s2s3s);      DeleteMatrix(m, s5s7s);      DeleteMatrix(m, s2);      DeleteMatrix(m, s3);      DeleteMatrix(m, s5);      DeleteMatrix(m, s7);      int\*\* result = CombineMatrix(m, c11, c12, c21, c22);      DeleteMatrix(m, c11);      DeleteMatrix(m, c12);      DeleteMatrix(m, c21);      DeleteMatrix(m, c22);      return result;  }  int\*\* CreateMatrix(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 value, int\*\*& matrix)  {      for (int i = 0; i < n; i++)      {          for (int j = 0; j < n; j++)          {              matrix[i][j] = 5;          }      }  }  void DeleteMatrix(int n, int\*\* matrix)  {      free(matrix[0]);      free(matrix);  }  int\*\* SliceMatrix(int n, int\*\* matrix, int offseti, int offsetj)  {      int m = n / 2;      int\*\* slice = CreateMatrix(m);      for (int i = 0; i < m; i++)      {          for (int j = 0; j < m; j++)          {              slice[i][j] = matrix[offseti + i][offsetj + j];          }      }      return slice;  }  int\*\* SumMatrix(int n, int\*\* matrix1, int\*\* matrix2)  {      int\*\* result = CreateMatrix(n);      for (int i = 0; i < n; i++)      {          for (int j = 0; j < n; j++)          {              result[i][j] = matrix1[i][j] + matrix2[i][j];          }      }      return result;  }  int\*\* SubMatrix(int n, int\*\* matrix1, int\*\* matrix2)  {      int\*\* result = CreateMatrix(n);      for (int i = 0; i < n; i++)      {          for (int j = 0; j < n; j++)          {              result[i][j] = matrix1[i][j] - matrix2[i][j];          }      }      return result;  }  int\*\* MulMatrix(int n, int\*\* matrix1, int\*\* matrix2)  {      int\*\* result = CreateMatrix(n);      for (int i = 0; i < n; i++)      {          for (int j = 0; j < n; j++)          {              result[i][j] = 0;              for (int k = 0; k < n; k++)              {                  result[i][j] += matrix1[i][k] \* matrix2[k][j];              }          }      }      return result;  }  int\*\* CombineMatrix(int m, int\*\* c11, int\*\* c12, int\*\* c21, int\*\* c22)  {      int n = 2 \* m;      int\*\* result = CreateMatrix(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;  }  void PrintMatrix(int n, int\*\* matrix)  {      for (int i = 0; i < n; i++)      {          for (int j = 0; j < n; j++)          {              cout << matrix[i][j] << " ";          }          cout << endl;      }      cout << endl;  } |