**Содержание**

[Введение 3](#_Toc118968846)

[1 Теоретические сведения 4](#_Toc118968847)

[2 Реализация алгоритма 8](#_Toc118968849)

[3 Параллельная реализация MPI 14](#_Toc118968851)

[4 Параллельная реализация OpenMP 21](#_Toc118968852)

[5 Тестирование алгоритмов 28](#_Toc118968854)

[Заключение 32](#_Toc118968859)

[Список литературы 33](#_Toc118968860)

# Введение

Параллельное программирование служит для создания программ, эффективно использующих вычислительные ресурсы за счет одновременного исполнения кода на нескольких вычислительных узлах. Для создания параллельных приложений используются параллельные языки программирования и специализированные системы поддержки параллельного программирования, такие как MPI и OpenMP. Параллельное программирование является более сложным по сравнению с последовательным как в написании кода, так и в его отладки.

В отличие от программирования последовательных вычислений, концептуальную основу которого составляет понятие алгоритма, реализуемого по шагам строго последовательно во времени, программа порождает совокупность параллельно протекающих процессов обработки информации, полностью независимых или связанных между собой статическими или динамическими пространственно-временными, или причинно-следственными отношениями.

Объект исследования – Технологии OpenMP и MPI.

Предмет исследования – Алгоритм задачи расстановки на шахматной доске ферзей.

Цель работы (проекта) – Исследование и реализация алгоритма задачи расстановки на шахматной доске ферзей на вычислительных системах с общей (SMP) и распределённой памятью (Cluster).

Задачи курсовой работы:

* Исследование предметной области курсовой работы (описание задачи)
* Анализ исходных данных задания на курсовую работу
* Разработка последовательного алгоритма
* Реализация параллельного алгоритма OpenMP
* Реализация параллельного алгоритма MPI
* Проведение вычислительных экспериментов

# 1 Теоретические сведения

1.1 Стандарт MPI

Система программирования MPI предназначена для ЭВМ с индивидуальной памятью, т.е. для многопроцессорных систем с обменом сообщениями. MPI имеет следующие особенности.

* MPI − библиотека функций, а не язык. Она определяет состав, имена, вызовы функций и результаты их работы. Программы, которые пишутся на языках Fortran, C и C++, компилируются обычными компиляторами и связаны с MPIбиблиотекой.
* MPI − описание функций, а не реализация. Все поставщики параллельных компьютерных систем предлагают реализации MPI для своих машин бесплатно, реализации общего назначения также могут быть получены из Интернета. Правильная MPI-программа должна выполняться на всех реализациях без изменения.

В модели передачи сообщений процессы, выполняющиеся параллельно, имеют раздельные адресные пространства. Обмен происходит, когда часть адресного пространства одного процесса скопирована в адресное пространство другого процесса. Эта операция совместная и возможна только, когда первый процесс выполняет операцию передачи сообщения, а второй − операцию его получения.

Процессы в MPI принадлежат группам. Если группа содержит n процессов, то процессы нумеруются внутри группы номерами, которые являются целыми числами от 0 до n-l. Имеется начальная группа, которой принадлежат все процессы в реализации MPI.

Контекст есть свойство коммуникаторов, которое позволяет разделять пространство обмена. Сообщение, посланное в одном контексте, не может быть получено в другом контексте. Контексты не являются явными объектами MPI, они проявляются только как часть реализации коммуникаторов.

Понятия контекста и группы объединены в едином объекте, называемом коммуникатором. Таким образом, отправитель или получатель, определенные в операции посылки или получения, всегда обращается к номеру процесса в группе, идентифицированной данным коммуникатором. [2]

1.2 Библиотека программирования OpenMP

OpenMP (Open Multi - processing) - открытый стандарт для распараллеливания программ на языках Си, Си++, Фортран. Дает описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью.

OpenMP реализует параллельные вычисления с помощью [многопоточности](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D0%BF%D0%BE%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C), в которой «главный» (master) поток создает набор подчиненных (slave) потоков и задача распределяется между ними. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами (количество процессоров не обязательно должно быть больше или равно количеству потоков).

Задачи, выполняемые потоками параллельно, также как и данные, требуемые для выполнения этих задач, описываются с помощью специальных директив препроцессора соответствующего языка — прагм.

Количество создаваемых потоков может регулироваться как самой программой при помощи вызова библиотечных процедур, так и извне, при помощи переменных окружения.

OpenMP прост в использовании и включает лишь два базовых типа конструкций: директивы pragma и функции исполняющей среды OpenMP. Директивы pragma, как правило, указывают компилятору реализовать параллельное выполнение блоков кода. Все эти директивы начинаются с #pragma omp. Как и любые другие директивы pragma, они игнорируются компилятором, не поддерживающим конкретную технологию — в данном случае OpenMP.

Функции OpenMP служат в основном для изменения и получения параметров среды. Кроме того, OpenMP включает API-функции для поддержки некоторых типов синхронизации. Чтобы задействовать эти функции библиотеки OpenMP периода выполнения (исполняющей среды), в программу нужно включить заголовочный файл omp.h. Если вы используете в приложении только OpenMP-директивы pragma, включать этот файл не требуется.[3].

1.3 Задача «О восьми ферзях»

В более «математическом» виде задача может быть сформулирована несколькими способами, например, так: «Заполнить матрицу размером 8×8 нулями и единицами таким образом, чтобы сумма всех элементов матрицы была равна 8, при этом сумма элементов ни в одном столбце, строке или диагональном ряде матрицы не превышала единицы».

Конечная цель, поставленная перед решающим задачу, может формулироваться в нескольких вариантах:

* Построить одно, любое решение задачи.
* Аналитически доказать, что решение существует.
* Определить количество решений.
* Построить все возможные решения.
* Одна из типовых задач по программированию алгоритмов перебора: создать компьютерную программу, находящую все возможные решения задачи.

Общее число возможных расположений 8 ферзей на 64-клеточной доске равно 4 426 165 368 (= 64!/(8!(64-8)!)). Общее число возможных расположений, удовлетворяющих условию задачи, равно 92. Современные компьютеры уже позволяют произвести решение задачи (нахождение любого или всех решений) путём прямого перебора всех возможных вариантов расстановки, но обычно такое решение считается некорректным, и от решающего задачу требуется найти алгоритм, который позволял бы существенно сократить объём перебора. Например, очевидно, что на одной горизонтали или вертикали доски не может находиться больше одного ферзя, поэтому алгоритм решения изначально не должен включать в перебор позиции, где два ферзя стоят на одной горизонтали или вертикали. Даже такое простое правило способно существенно уменьшить число возможных расположений: 16 777 216 (то есть 88) вместо 4 426 165 368. Генерируя перестановки, которые являются решениями задачи о восьми ладьях и затем проверяя атаки по диагоналям, можно сократить число возможных расположений всего до 40 320 (то есть 8!). Однако, если условие нападения по диагонали учитывать при генерации позиций, скорость счёта возрастает на порядок.[5].

## **1.4 Вычисление ускорения и эффективности**

Ускорением параллельного алгоритма называется отношение

, где - время выполнения задачи на *p*процессорах, а - время выполнения на одном процессоре.

C ускорением связана эффективность параллельного алгоритма.

Эффективность определяется следующим образом:

, где - ускорение алгоритма, *p* - кол-во процессоров.

По определению .

Для любого числа p выполняются следующие условия:

и .

Если алгоритм достигает максимального ускорения (), то . На практике эффективность убывает при увеличении числа процессоров.

Иногда возможны случаи, когда (суперлинейное ускорение).

# 2 Реализация алгоритма

2.1 Последовательная реализация алгоритма.

|  |  |
| --- | --- |
|  |  |
| #include <iostream>  #include <ctime>  using namespace std;  int BadPos[200][2];  int GoodPos[100][8];  int Count = 0;  void show(int I)  {  if (I == 0)  {  system("cls");  for (int i = 0; i <= 100 && GoodPos[i][0] != 0; i++)  {  for (int j = 0; j < 8; j++)  cout << GoodPos[i][j] << " ";  cout << '\n';  }  }  cout << Count << '\n';  if(I == 1)  {  system("cls");  for (int i = 0; i <= 100 && GoodPos[i][0] != 0; i++)  {  for (int j = 0, X=0; j < 8; j++)  {  X = GoodPos[i][j] % 10;  for (int f = 1; f < X; f++)  cout << "\_|";  cout << "0|";  for (int f = 8; f > X; f--)  cout << "\_|";  cout << '\n';  }  cout << "------------------"<<'\n';  system("pause");  system("cls");  }  }    }  void bad(int \*I)  {  int add = 0;  for (; BadPos[add][0] != 0; add++);  for (int i = I[0], j = I[1]; i <= 8 && j <= 8 && i >= 1 && j >= 1; i += 1, j += 1)  {  BadPos[add][0] = i;  BadPos[add][1] = j;  add++;  }  for (int i = I[0], j = I[1]; i <= 8 && j <= 8 && i >= 1 && j >= 1; i += 1) //Юг  {  BadPos[add][0] = i;  BadPos[add][1] = j;  add++;  }  for (int i = I[0], j = I[1]; i <= 8 && j <= 8 && i >= 1 && j >= 1; i += 1, j -= 1)  {  BadPos[add][0] = i;  BadPos[add][1] = j;  add++;  }  for (int i = I[0], j = I[1]; i <= 8 && j <= 8 && i >= 1 && j >= 1; i -= 1, j += 1)  {  BadPos[add][0] = i;  BadPos[add][1] = j;  add++;  }  for (int i = I[0], j = I[1]; i <= 8 && j <= 8 && i >= 1 && j >= 1; i -= 1) //Север  {  BadPos[add][0] = i;  BadPos[add][1] = j;  add++;  }  for (int i = I[0], j = I[1]; i <= 8 && j <= 8 && i >= 1 && j >= 1; i -= 1, j -= 1)  {  BadPos[add][0] = i;  BadPos[add][1] = j;  add++;  }  BadPos[add][0] = 350169;  }  void good(int \*I1, int \*I2, int \*I3, int \*I4, int \*I5, int \*I6, int \*I7, int \*I8)  {  Count++;  int X1, X2, X3, X4, X5, X6, X7, X8, stop = 0;  X1 = I1[0] \* 10 + I1[1];  X2 = I2[0] \* 10 + I2[1];  X3 = I3[0] \* 10 + I3[1];  X4 = I4[0] \* 10 + I4[1];  X5 = I5[0] \* 10 + I5[1];  X6 = I6[0] \* 10 + I6[1];  X7 = I7[0] \* 10 + I7[1];  X8 = I8[0] \* 10 + I8[1];  for(int i = 0; i < 100 && stop == 0; i++)  for (int j = 0; j < 8 && stop == 0; j++)  {  if (GoodPos[i][j] == 0)  {  GoodPos[i][j] = X1;  GoodPos[i][j + 1] = X2;  GoodPos[i][j + 2] = X3;  GoodPos[i][j + 3] = X4;  GoodPos[i][j + 4] = X5;  GoodPos[i][j + 5] = X6;  GoodPos[i][j + 6] = X7;  GoodPos[i][j + 7] = X8;  stop = 1;  }  }  }  bool ask(int \*I)  {  bool ask = 1;  for (int i = 0; i <= 1000; i++)  {  if(BadPos[i][0] == I[0] && BadPos[i][1] == I[1])  ask = 0;  } | return ask;  }  void unbad()  {  int add = 0;  for (; BadPos[add][0] != 0; add++);  add--;  BadPos[add][0] = 0;  BadPos[add][1] = 0;  for (add--; BadPos[add][0] != 350169; add--)  {  BadPos[add][0] = 0;  BadPos[add][1] = 0;  }  }  void root()  {  int f1[2] = { 1, 1 }, f2[2] = { 2, 1 }, f3[2] = { 3, 1 }, f4[2] = { 4, 1 };  int f5[2] = { 5, 1 }, f6[2] = { 6, 1 }, f7[2] = { 7, 1 }, f8[2] = { 8, 1 }; //все ферзи  int num = 0;  for (int f1j = 1; f1j <= 8; f1j++)  {  num = 0;  for (int i = 0; i < 200; i++)  for (int j = 0; j < 2; j++)  BadPos[i][j] = 0;  f1[1] = f1j;  num++; bad(f1);  for (int f2j = 1; f2j <= 8; f2j++)  {  for (; num != 1; num--)  unbad();  f2[1] = f2j;  if (ask(f2))  {  num++; bad(f2);  for (int f3j = 1; f3j <= 8; f3j++)  {  for (; num != 2; num--)  unbad();  f3[1] = f3j;  if (ask(f3))  {  num++; bad(f3);  for (int f4j = 1; f4j <= 8; f4j++)  {  for (; num != 3; num--)  unbad();  f4[1] = f4j;  if (ask(f4))  {  num++; bad(f4);  for (int f5j = 1; f5j <= 8; f5j++)  {  for (; num != 4; num--)  unbad();  f5[1] = f5j;  if (ask(f5))  {  num++; bad(f5);  for (int f6j = 1; f6j <= 8; f6j++)  {  for (; num != 5; num--)  unbad();  f6[1] = f6j;  if (ask(f6))  {  num++; bad(f6);  for (int f7j = 1; f7j <= 8; f7j++)  {  for (; num != 6; num--)  unbad();  f7[1] = f7j;  if (ask(f7))  {  num++; bad(f7);  for (int f8j = 1; f8j <= 8; f8j++)  {  for (; num != 7; num--)  unbad();  f8[1] = f8j;  if (ask(f8))  {  good(f1, f2, f3, f4, f5, f6, f7, f8);  num++; bad(f8);  }  }  }  }  }  }  }  }  }  }  }  }  }  }  }  show(0);  }  int main()  {  setlocale(LC\_ALL, "Russian");  unsigned int start\_time = clock();  for (int i = 0; i < 100; i++)  for (int j = 0; j < 8; j++)  GoodPos[i][j] = 0;  root();  unsigned int end\_time = clock();  unsigned int search\_time = end\_time - start\_time;  cout << "Время работы (318 было): " << search\_time << '\n';  system("pause");  } |

## **2.1.1 Результат выполнения алгоритма**

На рисунке 1 показан результат работы последовательного алгоритма.

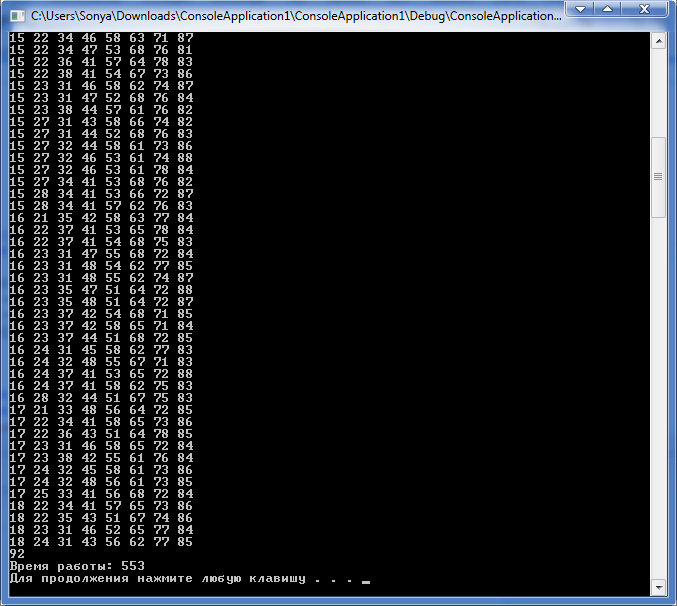


Рисунок 1 – Результат выполнения программы

# 3 Параллельная реализация MPI

|  |  |
| --- | --- |
|  |  |
| #include <iostream>  #include "mpi.h"  #include <ctime>  using namespace std;  int BadPos[200][2]; //плохие ячейки доски  int GoodPos[100][8]; //хорошие ячейки доски  int Count = 0; //количество вариантов  int \_m\_my\_id, \_m\_num\_procs, \_m\_TAG = 0,  \_m\_root\_id = 0;  MPI\_Status \_m\_status;  void show(int I)  {  if (I == 0)  {  system("cls");  for (int i = 0; i <= 100 && GoodPos[i][0] != 0; i++)  {  for (int j = 0; j < 8; j++)  cout << GoodPos[i][j] << " ";  cout << '\n';  }  }  cout << Count << '\n';  if(I == 1)  {  system("cls");  for (int i = 0; i <= 100 && GoodPos[i][0] != 0; i++)  {  for (int j = 0, X=0; j < 8; j++)  {  X = GoodPos[i][j] % 10;  for (int f = 1; f < X; f++)  cout << "\_|";  cout << "0|";  for (int f = 8; f > X; f--)  cout << "\_|";  cout << '\n';  }  cout << "------------------"<<'\n';  system("pause");  system("cls");  }  }  }  void bad(int \*I)  {  int add = 0;//номер ячейки с которой вставляем  for (; BadPos[add][0] != 0; add++);  for (int i = I[0], j = I[1]; i <= 8 && j <= 8 && i >= 1 && j >= 1; i += 1, j += 1) //Юго-Восток  {  BadPos[add][0] = i;  BadPos[add][1] = j;  add++;  }  for (int i = I[0], j = I[1]; i <= 8 && j <= 8 && i >= 1 && j >= 1; i += 1) //Юг  {  BadPos[add][0] = i;  BadPos[add][1] = j;  add++;  }  for (int i = I[0], j = I[1]; i <= 8 && j <= 8 && i >= 1 && j >= 1; i += 1, j -= 1) //Юго-запад  {  BadPos[add][0] = i;  BadPos[add][1] = j;  add++;  }  for (int i = I[0], j = I[1]; i <= 8 && j <= 8 && i >= 1 && j >= 1; i -= 1, j += 1) //Северо-Восток  {  BadPos[add][0] = i;  BadPos[add][1] = j;  add++;  }  for (int i = I[0], j = I[1]; i <= 8 && j <= 8 && i >= 1 && j >= 1; i -= 1) //Север  {  BadPos[add][0] = i;  BadPos[add][1] = j;  add++;  }  for (int i = I[0], j = I[1]; i <= 8 && j <= 8 && i >= 1 && j >= 1; i -= 1, j -= 1) //Северо-Запад  {  BadPos[add][0] = i;  BadPos[add][1] = j;  add++;  }  BadPos[add][0] = 350169;  }  void good(int \*I1, int \*I2, int \*I3, int \*I4, int \*I5, int \*I6, int \*I7, int \*I8)  {  Count++;  int X1, X2, X3, X4, X5, X6, X7, X8, stop = 0;  X1 = I1[0] \* 10 + I1[1];  X2 = I2[0] \* 10 + I2[1];  X3 = I3[0] \* 10 + I3[1];  X4 = I4[0] \* 10 + I4[1];  X5 = I5[0] \* 10 + I5[1];  X6 = I6[0] \* 10 + I6[1];  X7 = I7[0] \* 10 + I7[1];  X8 = I8[0] \* 10 + I8[1];  for(int i = 0; i < 100 && stop == 0; i++)  for (int j = 0; j < 8 && stop == 0; j++)  {  if (GoodPos[i][j] == 0)  {  GoodPos[i][j] = X1;  GoodPos[i][j + 1] = X2;  GoodPos[i][j + 2] = X3;  GoodPos[i][j + 3] = X4;  GoodPos[i][j + 4] = X5;  GoodPos[i][j + 5] = X6;  GoodPos[i][j + 6] = X7;  GoodPos[i][j + 7] = X8;  stop = 1;  }  }  }  bool ask(int \*I)  {  bool ask = 1;  for (int i = 0; i <= 1000; i++)  {  if(BadPos[i][0] == I[0] && BadPos[i][1] == I[1])  ask = 0;  }  return ask;  }  void unbad()  {  int add = 0;  for (; BadPos[add][0] != 0; add++); | add--;  BadPos[add][0] = 0;  BadPos[add][1] = 0;  for (add--; BadPos[add][0] != 350169; add--)  {  BadPos[add][0] = 0;  BadPos[add][1] = 0;  }  }  void root()  {  int f1[2] = { 1, 1 }, f2[2] = { 2, 1 }, f3[2] = { 3, 1 }, f4[2] = { 4, 1 };  int f5[2] = { 5, 1 }, f6[2] = { 6, 1 }, f7[2] = { 7, 1 }, f8[2] = { 8, 1 }; //все ферзи  int num = 0;  for (int f1j = 1; f1j <= 8; f1j++)  {  // каждому четному процессору отправляем четный номер, каждому нечетному нечетный номер  if ((\_m\_my\_id%2 == 0 && f1j%2 == 0) || (\_m\_my\_id%2 1= 0 && f1j%2 != 0))  {  num = 0;  for (int i = 0; i < 200; i++)  for (int j = 0; j < 2; j++)  BadPos[i][j] = 0;  f1[1] = f1j;  num++; bad(f1);  for (int f2j = 1; f2j <= 8; f2j++)  {  for (; num != 1; num--)  unbad();  f2[1] = f2j;  if (ask(f2))  {  num++; bad(f2);  for (int f3j = 1; f3j <= 8; f3j++)  {  for (; num != 2; num--)  unbad();  f3[1] = f3j;  if (ask(f3))  {  num++; bad(f3);  for (int f4j = 1; f4j <= 8; f4j++)  {  for (; num != 3; num--)  unbad();  f4[1] = f4j;  if (ask(f4))  {  num++; bad(f4);  for (int f5j = 1; f5j <= 8; f5j++)  {  for (; num != 4; num--)  unbad();  f5[1] = f5j;  if (ask(f5))  {  num++; bad(f5);  for (int f6j = 1; f6j <= 8; f6j++)  {  for (; num != 5; num--)  unbad();  f6[1] = f6j;  if (ask(f6))  {  num++; bad(f6);  for (int f7j = 1; f7j <= 8; f7j++)  {  for (; num != 6; num--)  unbad();  f7[1] = f7j;  if (ask(f7))  {  num++; bad(f7);  for (int f8j = 1; f8j <= 8; f8j++)  {  for (; num != 7; num--)  unbad();  f8[1] = f8j;  if (ask(f8))  {  good(f1, f2, f3, f4, f5, f6, f7, f8);  num++; bad(f8);  }  }  }  }  }  }  }  }  }  }  }  }  }  }  }  }  show(1);  }  int main()  {  setlocale(LC\_ALL, "Russian");  unsigned int start\_time = clock();  // инициализация mpi  MPI\_Init(&argc, &argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &\_m\_my\_id);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &\_m\_num\_procs);    for (int i = 0; i < 100; i++)  for (int j = 0; j < 8; j++)  GoodPos[i][j] = 0;    // отправляем кол-во  MPI\_Bcast(&Count, 1, MPI\_INT, \_m\_root\_id, MPI\_COMM\_WORLD);  // отправляем массивы  MPI\_Bcast(BadPos, Count, MPI\_INT, \_m\_root\_id, MPI\_COMM\_WORLD);  MPI\_Bcast(GoodPos, Count, MPI\_INT, \_m\_root\_id, MPI\_COMM\_WORLD);  root();  unsigned int end\_time = clock();  unsigned int search\_time = end\_time - start\_time;  cout << "Время работы (318 было): " << search\_time << '\n';  system("pause");    MPI\_Finalize();  } |

При написании кода программы использовались такие функций MPI как:

* MPI\_Init – инициализирует окружение MPI, эта функция вызывается один раз в начале программы, до первого вызова других MPI-функций.
* MPI\_Comm\_size – возвращает число процессов внутри коммуникатора, существует предопределенный коммуникатор MPI\_COMM\_WORLD, который включает все процессы;
* MPI\_Comm\_rank – возвращает ранг (номер) текущего процесса в коммуникаторе;
* MPI\_Send – отправляет сообщение;
* MPI\_Recv – получает сообщение.
* MPI\_Finalize – выход из системы MPI, после вызова этой функции кодпрограммы может продолжить свое выполнение, но выполнять вызов MPI-функций уже нельзя.
* MPI\_Bcast**-**широковещательная рассылка данных. Процесс с номером root рассылает сообщение из своего буфера передачи всем процессам области связи коммуникатора comm.
* MPI\_Scatter -функция разбивает сообщение из буфера посылки процесса root на равные части размером sendcount и посылает i-ю часть в буфер приема процесса с номером i (в том числе и самому себе). Процесс root использует оба буфера (посылки и приема), поэтому в вызываемой им подпрограмме все параметры являются существенными. Остальные процессы группы с коммуникатором comm являются только получателями, поэтому для них параметры, специфицирующие буфер посылки, не существенны.

Компиляция программы производится командой:

mpicc –o sort sort.c,где:

* *mpicc*– это стандартный компилятор для MPI
* sort – исполняемый файл
* sort.c-исходный код программы

Запуск программы производиться командой:

mpirun –np 10 sort, где*:*

*10*– это количество процессов

# 4 Параллельная реализация OpenMP

4.1 Код программы с использованием параллельных алгоритмов

|  |  |
| --- | --- |
| #include <iostream>  #include <omp.h>  #include <ctime>  using namespace std;  int BadPos[200][2];  int GoodPos[100][8];  int Count = 0;  void show(int I)  {  if (I == 0)  {  system("cls");  for (int i = 0; i <= 100 && GoodPos[i][0] != 0; i++)  {  for (int j = 0; j < 8; j++)  cout << GoodPos[i][j] << " ";  cout << '\n';  }  }  cout << Count << '\n';  if(I == 1)  {  system("cls");  for (int i = 0; i <= 100 && GoodPos[i][0] != 0; i++)  {  for (int j = 0, X=0; j < 8; j++)  {  X = GoodPos[i][j] % 10;  for (int f = 1; f < X; f++)  cout << "\_|";  cout << "0|";  for (int f = 8; f > X; f--)  cout << "\_|";  cout << '\n';  }  cout << "------------------"<<'\n';  system("pause");  system("cls");  }  }    }  void bad(int \*I)  {  int add = 0;  for (; BadPos[add][0] != 0; add++);  for (int i = I[0], j = I[1]; i <= 8 && j <= 8 && i >= 1 && j >= 1; i += 1, j += 1)  {  BadPos[add][0] = i;  BadPos[add][1] = j;  add++;  }  for (int i = I[0], j = I[1]; i <= 8 && j <= 8 && i >= 1 && j >= 1; i += 1) //Юг  {  BadPos[add][0] = i;  BadPos[add][1] = j;  add++;  }  for (int i = I[0], j = I[1]; i <= 8 && j <= 8 && i >= 1 && j >= 1; i += 1, j -= 1)  {  BadPos[add][0] = i;  BadPos[add][1] = j;  add++;  }  for (int i = I[0], j = I[1]; i <= 8 && j <= 8 && i >= 1 && j >= 1; i -= 1, j += 1)  {  BadPos[add][0] = i;  BadPos[add][1] = j;  add++;  }  for (int i = I[0], j = I[1]; i <= 8 && j <= 8 && i >= 1 && j >= 1; i -= 1) //Север  {  BadPos[add][0] = i;  BadPos[add][1] = j;  add++;  }  for (int i = I[0], j = I[1]; i <= 8 && j <= 8 && i >= 1 && j >= 1; i -= 1, j -= 1)  {  BadPos[add][0] = i;  BadPos[add][1] = j;  add++;  }  BadPos[add][0] = 350169;  }  void good(int \*I1, int \*I2, int \*I3, int \*I4, int \*I5, int \*I6, int \*I7, int \*I8)  {  Count++;  int X1, X2, X3, X4, X5, X6, X7, X8, stop = 0;  X1 = I1[0] \* 10 + I1[1];  X2 = I2[0] \* 10 + I2[1];  X3 = I3[0] \* 10 + I3[1];  X4 = I4[0] \* 10 + I4[1];  X5 = I5[0] \* 10 + I5[1];  X6 = I6[0] \* 10 + I6[1];  X7 = I7[0] \* 10 + I7[1];  X8 = I8[0] \* 10 + I8[1];  for(int i = 0; i < 100 && stop == 0; i++)  for (int j = 0; j < 8 && stop == 0; j++)  {  if (GoodPos[i][j] == 0)  {  GoodPos[i][j] = X1;  GoodPos[i][j + 1] = X2;  GoodPos[i][j + 2] = X3;  GoodPos[i][j + 3] = X4;  GoodPos[i][j + 4] = X5;  GoodPos[i][j + 5] = X6;  GoodPos[i][j + 6] = X7;  GoodPos[i][j + 7] = X8;  stop = 1;  }  }  }  bool ask(int \*I)  {  bool ask = 1;  for (int i = 0; i <= 1000; i++)  {  if(BadPos[i][0] == I[0] && BadPos[i][1] == I[1])  ask = 0;  } | return ask;  }  void unbad()  {  int add = 0;  for (; BadPos[add][0] != 0; add++);  add--;  BadPos[add][0] = 0;  BadPos[add][1] = 0;  for (add--; BadPos[add][0] != 350169; add--)  {  BadPos[add][0] = 0;  BadPos[add][1] = 0;  }  }  void root()  {  int f1[2] = { 1, 1 }, f2[2] = { 2, 1 }, f3[2] = { 3, 1 }, f4[2] = { 4, 1 };  int f5[2] = { 5, 1 }, f6[2] = { 6, 1 }, f7[2] = { 7, 1 }, f8[2] = { 8, 1 }; //все ферзи  int num = 0;  for (int f1j = 1; f1j <= 8; f1j++)  {  num = 0;  #pragma omp parallel for  for (int i = 0; i < 200; i++)  for (int j = 0; j < 2; j++)  BadPos[i][j] = 0;  f1[1] = f1j;  num++; bad(f1);  for (int f2j = 1; f2j <= 8; f2j++)  {  for (; num != 1; num--)  unbad();  f2[1] = f2j;  if (ask(f2))  {  num++; bad(f2);  for (int f3j = 1; f3j <= 8; f3j++)  {  for (; num != 2; num--)  unbad();  f3[1] = f3j;  if (ask(f3))  {  num++; bad(f3);  for (int f4j = 1; f4j <= 8; f4j++)  {  for (; num != 3; num--)  unbad();  f4[1] = f4j;  if (ask(f4))  {  num++; bad(f4);  for (int f5j = 1; f5j <= 8; f5j++)  {  for (; num != 4; num--)  unbad();  f5[1] = f5j;  if (ask(f5))  {  num++; bad(f5);  for (int f6j = 1; f6j <= 8; f6j++)  {  for (; num != 5; num--)  unbad();  f6[1] = f6j;  if (ask(f6))  {  num++; bad(f6);  for (int f7j = 1; f7j <= 8; f7j++)  {  for (; num != 6; num--)  unbad();  f7[1] = f7j;  if (ask(f7))  {  num++; bad(f7);  for (int f8j = 1; f8j <= 8; f8j++)  {  for (; num != 7; num--)  unbad();  f8[1] = f8j;  if (ask(f8))  {  good(f1, f2, f3, f4, f5, f6, f7, f8);  num++; bad(f8);  }  }  }  }  }  }  }  }  }  }  }  }  }  }  }  show(0);  }  int main()  {  setlocale(LC\_ALL, "Russian");  unsigned int start\_time = clock();  #pragma omp parallel for  for (int i = 0; i < 100; i++)  for (int j = 0; j < 8; j++)  GoodPos[i][j] = 0;  root();  unsigned int end\_time = clock();  unsigned int search\_time = end\_time - start\_time;  cout << "Время работы (318 было): " << search\_time << '\n';  system("pause");  } |

При написании кода программы использовались такие директивы OpenMP как:

* *#pragma omp parallel* - Эта директива сообщает компилятору, что структурированный блок кода должен быть выполнен параллельно, в нескольких потоках. Каждый поток будет выполнять один и тот же поток команд, но не один и тот же набор команд — все зависит от операторов, управляющих логикой программы, таких как *if-else*.
* *#pragma omp for* - Так как циклы являются самыми распространенными конструкциями, где выполнение кода можно распараллелить, OpenMP поддерживает данную директиву и для них.
* *#pragma omp critical* – Данная директива OpenMP указывает на то, что код должен быть выполнен в одном потоке одновременно, в нашем случае, использовалась, в основном, для вывода информации на экран.
* *#pragma omp single* – Неявная барьерная синхронизация, которая выполняется в конце каждого параллельного региона для всех сопоставленных с ним потоков. Механизм барьерной синхронизации таков, что, пока все потоки не достигнут конца параллельного региона, ни один поток не сможет перейти его границу.

Компиляция программы производится командой:

*gcc –fopenmp z1.c –o z*, где:

* *gcc* – это стандартный компилятор Си для Unix систем.
* *-fopenmp* – это специальный ключ, используемый в дистрибутиве PelicanHPC для трансляции OpenMP-программы.

Запуск программы производиться командой:

* *OMP\_NUM\_THREADS=10 ./1z*, где*:*

*OMP\_NUM\_THREADS* – это количество потоков.

# .4.1.1 Результат выполнения параллельного алгоритма

На рисунках 2 и 3 изображены результаты выполнения программы с распараллеливанием вычислений, а также наглядный результат расстановки.

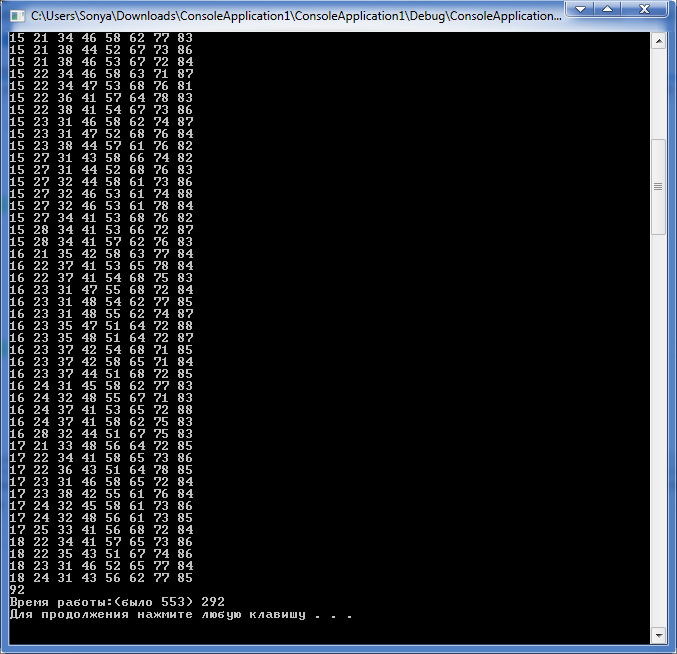


Рисунок 2 - Результат выполнения параллельной программы

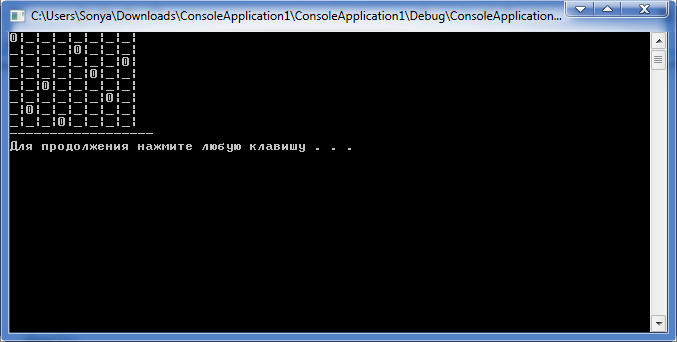


Рисунок 3 - Вариант расстановки ферзей

# 5 Тестирование алгоритмов

## **5.1.1 Вычисление ускорения MPI**

Вычислим ускорение алгоритма как

Таблица 1 — Зависимость ускорения от размера массива

|  |  |  |  |
| --- | --- | --- | --- |
| Количество элементов | Параллельный алгоритм | | |
| 2 процесса | 4 процесса | 8 процессов |
| 10000 | 0,18 | 0,48 | 0,04 |
| 20000 | 0,30 | 0,65 | 0,09 |
| 30000 | 0,37 | 0,75 | 0,11 |
| 40000 | 0,37 | 0,74 | 0,14 |
| 50000 | 0,41 | 0,80 | 0,17 |

По таблице 1 был построен график, который отображает данную зависимость. Результат показан на рисунке 4.

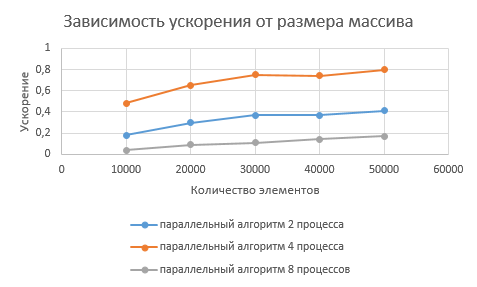


Рисунок 4 — График зависимости ускорения от количества элементов в массиве

## **5.1.2 Вычисление ускорения OpenMP**

Вычислим ускорение алгоритма как

Таблица 2 — Зависимость ускорения от размера массива

|  |  |  |  |
| --- | --- | --- | --- |
| Количество элементов | Параллельный алгоритм | | |
| 2 процесса | 4 процесса | 8 процессов |
| 10000 | 0,17 | 0,47 | 0,048 |
| 20000 | 0,31 | 0,66 | 0,093 |
| 30000 | 0,38 | 0,76 | 0,12 |
| 40000 | 0,36 | 0,73 | 0,13 |
| 50000 | 0,42 | 0,81 | 0,18 |

По таблице 2 был построен график, который отображает данную зависимость. Результат показан на рисунке 5.

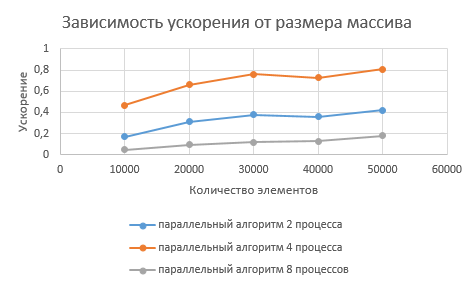


Рисунок 5 — График зависимости ускорения от количества элементов в массиве

## **5.2.1 Вычисление эффективности MPI**

Вычислим эффективности как , основываясь на данных по ускорению.

Таблица 3 — Зависимость эффективности от размера массива

|  |  |  |  |
| --- | --- | --- | --- |
| Количество элементов | Параллельный алгоритм | | |
| 2 процесса | 4 процесса | 8 процессов |
| 10000 | 0,04 | 0,24 | 0,01 |
| 20000 | 0,07 | 0,32 | 0,01 |
| 30000 | 0,09 | 0,37 | 0,01 |
| 40000 | 0,09 | 0,37 | 0,01 |
| 50000 | 0,10 | 0,40 | 0,02 |

По таблице 3 был построен график, который отображает данную зависимость. Результат показан на рисунке 6.

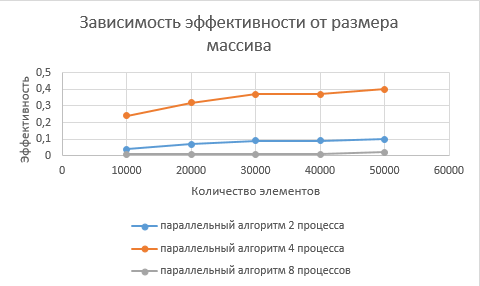


Рисунок 6 — График зависимости эффективности от количества элементов в массиве

## **5.2.2 Вычисление эффективности OpenMP**

Вычислим эффективности как , основываясь на данных по ускорению.

Таблица 4 — Зависимость эффективности от размера массива

|  |  |  |  |
| --- | --- | --- | --- |
| Количество элементов | Параллельный алгоритм | | |
| 2 процесса | 4 процесса | 8 процессов |
| 10000 | 0,04 | 0,25 | 0,006 |
| 20000 | 0,07 | 0,32 | 0,012 |
| 30000 | 0,09 | 0,38 | 0,015 |
| 40000 | 0,09 | 0,36 | 0,017 |
| 50000 | 0,12 | 0,41 | 0,02 |

По таблице 4 был построен график, который отображает данную зависимость. Результат показан на рисунке 7.

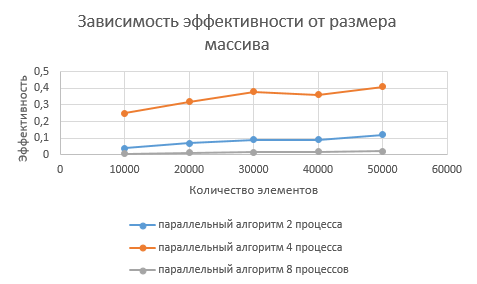


Рисунок 7 — График зависимости эффективности от количества элементов в массиве

# Заключение

В ходе курсовой работы была изучена парадигма параллельного программирования MPI и OpenMP, изучены основные функции и директивы.

Найдены все варианты расстановки восьми ферзей на стандартной шахматной доске, по средством прямого перебора ячеек, так, чтобы ни один из них не находился под боем другого.

Реализован алгоритм поставленной задачи с помощью OpenMP.

Реализован алгоритм поставленной задачи с помощью MPI.

При четырех процессорах, имеющих доступ к общей памяти и 50000 элементов, ускорение не превысит 0.81.

Вычислена эффективность работы алгоритмов, в зависимости от размера массива. При четырех процессорах и 50000 элементов, эффективность не превысит 0.41.

# Список литературы

1. Гусева, А. И. Вычислительные системы, сети и телекоммуникации [Текст] : учебник для студ. высш.проф. образования / А.И.Гусева, В.С.Киреев. - М. : ИЦ Академия, 2014. - 288 с. - (Бакалавриат)
2. Две парадигмы параллельного программирования [Электронный ресурс]. URL:http://www.intuit.ru/EDI/22\_05\_15\_7/1432246798-28527/tutorial/450/objects/1/files/01.pdf.
3. Мелехин, В.Ф. Вычислительные системы и сети [Текст] : учебник для студ. учреждений высш. проф. образования / В.Ф. Мелехин, Е.Г. Павловский. - М. : Академия, 2013. - 208 с. : ил. - (Бакалавриат).
4. Задача о восьми ферзях [Электронный ресурс]. https://ru.wikipedia.org/wiki/ (дата обращения: 10.11.2022).
5. https://studfile.net/preview/7015180/page:45/