МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«**Вятский государственный университет**»

**(«ВятГУ»)**

Факультет автоматики и вычислительной техники

Кафедра электронных вычислительных машин

Отчет по лабораторной работе №3

по дисциплине «Параллельное программирование»

Вариант 10

Выполнил студент группы ИВТ-32 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Щесняк Д. С./

Проверил доцент кафедры ЭВМ\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Чистяков Г.А./

Киров 2017

1. Задание на лабораторную работу

Познакомиться с потоками операционной системы, получить навыки реализации многопоточных приложений.

1. Выделить в полученной в ходе первой лабораторной работы реализации алгоритма фрагменты кода, выполнение которых может быть разнесено на несколько процессорных ядер.
2. Реализовать многопоточную версию алгоритма с помощью языка С++ и потоков операционной системы, используя при этом необходимые примитивы синхронизации.
3. Показать корректность полученной реализации путем осуществления на построенном в ходе первой лабораторной работы наборе тестов.
4. Провести доказательную оценку эффективность многопоточной реализации алгоритма.
5. Первый вариант параллельной программы

Основной цикл проверки состояний распараллелить невозможно, так как есть зависимость по данным между итерациями. Внутри самого цикла распараллелить возможно только проверку наличия состояний в списках. Максимально количество секций – 4, соответственно максимальное количество потоков – 4, но не все потоки будут задействованы, так как фишку невозможно сдвинуть в некоторые позиции. Для поля 4х4 среднее количество потоков будет равно:

Для поля 5х5:

Исходный код параллельного алгоритма представлен на рисунке 1

|  |
| --- |
| void t(State\* min, BinTree\* open, BinTree\* close, int i, State\*\* states) {  State\* s = new State(min->getMap()->shift(i), min);  if ((open->find(s) == NULL) && (close->find(s) == NULL)) {  states[i] = s;  }  }  std::vector<State\*> aThreads(Map\* map) {  BinTree\* open = new BinTree();  BinTree\* close = new BinTree(new State(map, NULL));  State\* min = close->min();  std::vector<std::thread> threads;  for (; min->getCost() != 0; min = open->min(), close->add(min), open->del(min))  {  int zero = min->getMap()->find(0);  State\* states[4] = { 0, 0, 0, 0 };  bool v[4] = { false, false, false, false };  if (zero / map->getCols() != 0) v[0] = true;  if (zero % map->getCols() != map->getCols() - 1) v[1] = true;  if (zero / map->getCols() != map->getLines() - 1) v[2] = true;  if (zero % map->getCols() != 0) v[3] = true;  for (int i = 0; i < 4; i++) {  if (v[i])  threads.emplace\_back(t, min, open, close, i, states);  }  for (auto &thr : threads) {  thr.join();  }  threads.clear();  for (int i = 0; i < 4; i++)  {  if (states[i]) {  open->add(states[i]);  }  }  }  std::vector<State\*> solution;  State\* s = min;  do  {  solution.push\_back(s);  s = s->getParent();  } while (s != NULL);  return solution;  } |

Рисунок 1 – Исходный код параллельного алгоритма

Данные тесты проводились на ЭВМ под управлением 64-х разрядной операционной системы Windows 10, с 4 Гб оперативной памяти, с 4-х ядерным процессором Intel N3530 c частотой 2.2 ГГц.

Тестирование параллельного и линейного алгоритма представлено в таблице 1. Экранные формы параллельной и линейной программы представлены на рисунке 2.

Таблица 1

|  |  |  |  |
| --- | --- | --- | --- |
| № | Тест | Линейное время (сек) | Параллельное время (сек) |
| 1 | |  |  |  |  | | --- | --- | --- | --- | | 13 | 11 | 10 | 9 | | 8 | 6 | 5 | 12 | | 2 | 4 | 3 | 14 | | 7 | 15 | 1 |  | | 7.252 | 12.459 |
| 2 | |  |  |  |  | | --- | --- | --- | --- | | 14 | 6 | 9 | 12 | | 2 | 5 | 11 | 15 | | 1 | 3 | 8 | 10 | | 13 | 7 | 4 |  | | 3.342 | 6.271 |
| 3 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 4 | 13 | 6 | 11 | 8 | | 24 | 3 | 22 | 23 | 7 | | 2 | 10 | 16 | 21 | 18 | | 14 | 15 | 17 | 9 | 1 | | 12 | 20 | 5 | 19 |  | | 476.648 | 303.842 |
| 4 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 16 | 1 | 18 | 11 | 3 | | 12 | 20 | 9 | 22 | 2 | | 19 | 21 | 5 | 17 | 13 | | 7 | 15 | 4 | 14 | 23 | | 6 | 24 | 10 | 8 |  | | 32.537 | 31.857 |
| 5 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 5 | 12 | 11 | 18 | 2 | | 15 | 6 | 4 | 17 | 7 | | 16 | 1 | 22 | 21 | 19 | | 14 | 3 | 23 | 24 | 10 | | 13 | 8 | 20 | 9 |  | | 29.646 | 29.188 |
| 6 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 6 | 18 | 5 | 10 | 2 | | 14 | 8 | 22 | 17 | 7 | | 20 | 11 | 19 | 13 | 4 | | 21 | 3 | 1 | 24 | 15 | | 23 | 9 | 16 | 12 |  | | 17.186 | 20.357 |
| 7 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 17 | 1 | 2 | 6 | 21 | | 13 | 18 | 7 | 4 | 16 | | 15 | 14 | 10 | 20 | 24 | | 8 | 19 | 5 | 22 | 3 | | 12 | 11 | 23 | 9 |  | | 1.290 | 4.035 |
| 8 | |  |  |  |  | | --- | --- | --- | --- | | 12 | 6 | 13 | 11 | | 15 | 9 | 8 | 2 | | 3 | 14 | 7 | 5 | | 4 | 10 | 1 |  | | 0.895 | 3.008 |
| 9 | |  |  |  | | --- | --- | --- | | 3 | 10 | 2 | | 7 | 6 | 4 | | 1 | 5 | 8 | | 11 | 9 |  | | 0.433 | 1.416 |
| 10 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 18 | 15 | 1 | 8 | 9 | | 6 | 19 | 13 | 2 | 11 | | 5 | 16 | 3 | 10 | 12 | | 4 | 17 | 7 | 14 |  | | 2.771 | 6.478 |

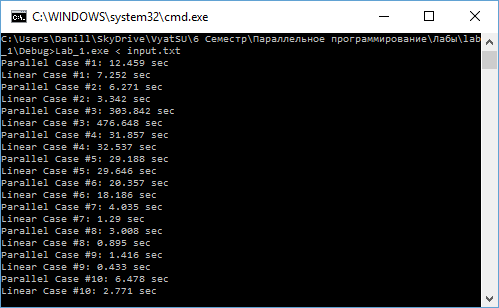


Рисунок 2 – Время прохождения тестов

Как видно из представленного в таблице 2 времени выполнения программы, параллельная реализация алгоритма эффективна при «сложных» комбинациях головоломок, это обуславливается тем, что поиск по дереву осуществляет большое количество операций, которые в параллельном варианте выполняются дольше чем создание потока.

1. Модифицированный вариант параллельного алгоритма

Модификация алгоритма заключается в том, что параллельно теперь вычисляется не одно передвижение фишки, а два. Таким образом каждый поток вычисляет 3 комбинации: один ход в заданном направление, два хода в заданном направлении, ход в заданном направлений и ход в направлении следующего хода по часовой стрелке. Графическая демонстрация ходов из точки 2х2 для матрицы 4х4, которые выполняют потоки представлена на рисунке 3. Цифры от 1 до 4 обозначают номер потока, который будет обсчитывать данный ход.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | 4 | 1 | 1 |  | | 4 | х | 2 | 2 | | 3 | 3 | 2 |  | |  | 3 |  |  | |

Рисунок 3 – Графическая демонстрация ходов.

Исходный код параллельного алгоритма представлен на рисунке 4

|  |
| --- |
| bool breakCond(Map\* map, int direction, int zero = -1) {  if (zero == -1) {  zero = map->find(0);  }  switch (direction)  {  case DIRECTION\_UP: if (zero / map->getCols() != 0) return true;  break;  case DIRECTION\_RIGHT: if (zero % map->getCols() != map->getCols() - 1) return true;  break;  case DIRECTION\_DOWN: if (zero / map->getCols() != map->getLines() - 1) return true;  break;  case DIRECTION\_LEFT: if (zero % map->getCols() != 0) return true;  break;  default:  return false;  }  return false;  }  void t(State\* min, BinTree\* open, BinTree\* close, int direction, State\*\* states) {  int k = direction \* 3;  int zero;  State\* s = new State(min->getMap()->shift(direction, &zero), min);  if ((open->find(s) == NULL) && (close->find(s) == NULL)) {  states[k] = s;  }  Map\* map = s->getMap();  bool v = breakCond(map, direction, zero);  if (v) {  State\* s2 = new State(s->getMap()->shift(direction), s);  if ((open->find(s2) == NULL) && (close->find(s2) == NULL)) {  states[k + 1] = s2;  }  }  v = breakCond(map, (direction + 1) % 4, zero);  if (v) {  State\* s3 = new State(s->getMap()->shift((direction + 1) % 4), s);  if ((open->find(s3) == NULL) && (close->find(s3) == NULL)) {  states[k + 2] = s3;  }  }  }  std::vector<State\*> aThreads(Map\* map) {  BinTree\* open = new BinTree();  BinTree\* close = new BinTree(new State(map, NULL));  State\* min = close->min();  std::vector<std::thread> threads;  for (; min->getCost() != 0; min = open->min(), close->add(min), open->del(min))  {  int zero = min->getMap()->find(0);  State\* states[12] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };  bool v[4] = { false, false, false, false };  if (zero / map->getCols() != 0) v[0] = true;  if (zero % map->getCols() != map->getCols() - 1) v[1] = true;  if (zero / map->getCols() != map->getLines() - 1) v[2] = true;  if (zero % map->getCols() != 0) v[3] = true;  for (int i = 0; i < 4; i++) {  if (v[i])  threads.emplace\_back(t, min, open, close, i, states);  }  for (auto &thr : threads) {  thr.join();  }  threads.clear();  for (int i = 0; i < 12; i++)  {  if (states[i]) {  open->add(states[i]);  }  }  }  std::vector<State\*> solution;  State\* s = min;  do  {  solution.push\_back(s);  s = s->getParent();  } while (s != NULL);  return solution;  } |

Рисунок 4 – Исходный код параллельного алгоритма

Данные тесты проводились на ЭВМ под управлением 64-х разрядной операционной системы Windows 10, с 4 Гб оперативной памяти, с 4-х ядерным процессором Intel N3530 c частотой 2.2 ГГц.

Тестирование параллельного и линейного алгоритма представлено в таблице 2. Экранные формы параллельной и линейной программы представлены на рисунке 5.

Таблица 2

|  |  |  |  |
| --- | --- | --- | --- |
| № | Тест | Линейное время (сек) | Параллельное время (сек) |
| 1 | |  |  |  |  | | --- | --- | --- | --- | | 13 | 11 | 10 | 9 | | 8 | 6 | 5 | 12 | | 2 | 4 | 3 | 14 | | 7 | 15 | 1 |  | | 2.966 | 5.373 |
| 2 | |  |  |  |  | | --- | --- | --- | --- | | 14 | 6 | 9 | 12 | | 2 | 5 | 11 | 15 | | 1 | 3 | 8 | 10 | | 13 | 7 | 4 |  | | 2.902 | 6.084 |
| 3 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 4 | 13 | 6 | 11 | 8 | | 24 | 3 | 22 | 23 | 7 | | 2 | 10 | 16 | 21 | 18 | | 14 | 15 | 17 | 9 | 1 | | 12 | 20 | 5 | 19 |  | | 407.36 | 278.627 |
| 4 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 16 | 1 | 18 | 11 | 3 | | 12 | 20 | 9 | 22 | 2 | | 19 | 21 | 5 | 17 | 13 | | 7 | 15 | 4 | 14 | 23 | | 6 | 24 | 10 | 8 |  | | 172.56 | 116.941 |
| 5 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 5 | 12 | 11 | 18 | 2 | | 15 | 6 | 4 | 17 | 7 | | 16 | 1 | 22 | 21 | 19 | | 14 | 3 | 23 | 24 | 10 | | 13 | 8 | 20 | 9 |  | | 54.615 | 47.819 |
| 6 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 6 | 18 | 5 | 10 | 2 | | 14 | 8 | 22 | 17 | 7 | | 20 | 11 | 19 | 13 | 4 | | 21 | 3 | 1 | 24 | 15 | | 23 | 9 | 16 | 12 |  | | 3.932 | 5.675 |
| 7 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 17 | 1 | 2 | 6 | 21 | | 13 | 18 | 7 | 4 | 16 | | 15 | 14 | 10 | 20 | 24 | | 8 | 19 | 5 | 22 | 3 | | 12 | 11 | 23 | 9 |  | | 52.266 | 42.256 |
| 8 | |  |  |  |  | | --- | --- | --- | --- | | 12 | 6 | 13 | 11 | | 15 | 9 | 8 | 2 | | 3 | 14 | 7 | 5 | | 4 | 10 | 1 |  | | 0.246 | 0.854 |
| 9 | |  |  |  | | --- | --- | --- | | 3 | 10 | 2 | | 7 | 6 | 4 | | 1 | 5 | 8 | | 11 | 9 |  | | 0.45 | 1.447 |
| 10 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 18 | 15 | 1 | 8 | 9 | | 6 | 19 | 13 | 2 | 11 | | 5 | 16 | 3 | 10 | 12 | | 4 | 17 | 7 | 14 |  | | 1.325 | 2.095 |

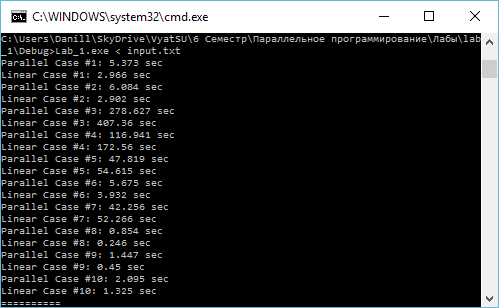


Рисунок 5 – Время выполнения тестов

Данный алгоритм в большинстве случаев работает быстрее чем первоначальный, но так как в данном алгоритме вместо новых 4-х комбинаций в каждом шаге добавляется по 12, то увеличивается список вершин, из-за чего увеличивается количество шагов при поиске элемента в дереве. Параллельная реализация так же эффективна только при «сложных» комбинациях головоломок.

1. Вывод

В ходе выполнения лабораторной работы были изучены стандартные потоки операционной системы. В результате выполнения лабораторной работы были разработаны и реализованный параллельные алгоритмы решения пятнашек.

Модифицированные параллельные и линейные алгоритмы оказались эффективнее чем их аналоги. Линейная версия модифицированного алгоритма при решений простых головоломок работает быстрее, чем ее параллельная реализация. Это связанно с тем, что создание потоков вычислительно сопоставимо с выполнением задачи данного потока, что существенно увеличивает время решения простых головоломок. На сложных комбинациях, для решения которых линейный алгоритм затрачивает более 30 секунд, наибольший выигрыш дает параллельный модифицированный алгоритм.