МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ И ФИЗИКО-МАТЕМАТИЧЕСКИХ НАУК

КАФЕДРА ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Отчёт по лабораторной работе №3 по дисциплине «Параллельное программирование»

**Многопоточность в языке C++**

|  |  |
| --- | --- |
| Выполнил: студент группы ФИб-4302-51-00 | / Д.А. Савин / |
| Проверил: ст. преподаватель каф. ПМиИ | / В.А. Бызов / |

Киров 2021

Задание 0

Запустить на выполнение пример 1. Заменить join на detach. Прокомментировать результат.

Код:

|  |
| --- |
| #include <iostream>  #include <thread>  using namespace std;  void foo(void)  {  std::cout << "Thread start..." << std::endl;  for (int i = 0; i < 10; ++i)  {  std::cout << "Thread id = " << std::this\_thread::get\_id()  << std::endl;  std::this\_thread::sleep\_for(std::chrono::seconds(1));  }  std::cout << "Thread finish!" << std::endl;  return;  }  int main()  {  std::thread myTh(foo);  std::cout << "Main thread id = " << std::this\_thread::get\_id()  << std::endl;  myTh.detach();  } |

Вывод: Родительский поток завершился с дочерним.

Задание 1

Разработать программу, выводящую числа от 1 до 100 в двух потоках, в первом – чётные, во втором – нечётные. Запустить несколько раз на выполнение. Вопрос. В каком порядке будут выводиться числа в консоли? Почему?

Код:

|  |
| --- |
| #include <iostream>  #include <thread>  using namespace std;  void foo(void)  {  std::cout << "Thread 1 start..." << std::endl;  for (int i = 0; i < 100; ++i)  {  if (i % 2 == 0)  {  cout << i << " - thread 1" << endl;  }  //std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));  }  std::cout << "Thread 1 finish!" << std::endl;  return;  }  void foo2(void)  {  std::cout << "Thread 2 start..." << std::endl;  for (int j = 0; j < 100; ++j)  {  if (j % 2 != 0)  {  cout << j << " - thread 2" << endl;  }  //std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));  }  std::cout << "Thread 2 finish!" << std::endl;  return;  }  int main()  {  std::thread myTh(foo), myTh2(foo2);  std::cout << "Main thread id = " << std::this\_thread::get\_id() << std::endl;  myTh.join();  myTh2.join();  } |

Вывод: Числа выводятся в хаотичном порядке. В момент завершения первого потока может начать выполняться второй поток.

Задание 2

Разработать программу, генерирующую P потоков (число P < 10 задаётся при запуске), каждый из которых сначала выводит сообщение о старте, а потом выводит свой номер 100 раз. Количество потоков P задает пользователь с клавиатуры. Не забудьте имитировать сложные вычисления в потоке. Длительность сложных вычислений – случайное число миллисекунд из диапазона [1000, 2000].

Код:

|  |
| --- |
| #include <iostream>  #include <thread>  using namespace std;  void foo(int k)  {  std::cout << " ---- Thread " << k << " start..." << std::endl;  for (int i = 0; i < 100; ++i)  {  std::cout << "Thread id = " << std::this\_thread::get\_id()  << "\t";  //std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));  }  std::cout << "\n---- Thread " << k << " finish!" << std::endl;  return;  }  int main()  {  std::cout << "Main thread id = " << std::this\_thread::get\_id() << std::endl;  int P = 10;    cout << "Input P: " << endl;  cin >> P;  thread\* threads = new thread[P];  for (int i = 0; i < P; i++)  {  threads[i] = thread(foo, i);  }  for (int i = 0; i < P; i++)  {  threads[i].join();  }  } |

Задание 3

Создать P потоков (число P < 10 задаётся при запуске), конкурирующих за общий ресурс – целочисленную переменную. Каждый поток увеличивает значение переменной на свой индекс. Потоки должны завершиться после того, как значение общей переменной превысило 100 (начальное значение – 0). Организовать корректное взаимодействие потоков с использованием объекта std::mutex.

Код:

|  |
| --- |
| #include <iostream>  #include <thread>  #include <vector>  #include <mutex>  using namespace std;  int counT = 0;  void foo(int cnt, std::mutex& mx)  {  std::cout << "Thread start, index= " << cnt << " counT= " << counT << std::endl;  //std::this\_thread::sleep\_for(std::chrono::milliseconds(rand() % 1000));  std::lock\_guard<std::mutex> lgmx(mx);  if (counT < 100)  {  counT += cnt;  }  else  {  cout << "counT > 100 | counT= " << counT << endl;  }  std::cout << "Thread finish! counT= "<< counT << std::endl;  return;  }  int main()  {  int P = 0;  while (P < 1 || P > 10)  {  std::cout << "Input P: ";  std::cin >> P;  }  vector <thread> Thr;  std::mutex mx;  while (counT < 100)  {  for (int i = 0; i < P; i++)  {    Thr.push\_back(thread(foo, i, ref(mx)));  Thr[i].join();  if (counT >= 100)  {  break;  }  }  std::unique\_lock<std::mutex> ulmx(mx);  ulmx.unlock();  Thr.clear();    }  } |

Задание 4

Написать программу, которая каждый элемент массива размера N заменяет на его наибольший простой делитель. Число N задается пользователем. Элементы массива – случайные натуральные числа из диапазона [105, 106]. Распараллелить с использованием std::thread. Замерить время работы программы для N = 2·107, 5·107 и 108 на 1, 2, 4 и 8 потоках. Для проверки результатов реализовать данную задачу последовательно, сравнить результаты. Замечание. Размер тестируемых массивов выбирать максимально возможным. Размеры приведены для компьютеров университета. Для более мощных компьютеров размеры тестируемых массивов нужно увеличить!

На каждом примере запустить не менее 3 раз. В таблицу занести среднее время выполнения на одном примере в секундах.

Код:

|  |
| --- |
| #include <iostream>  #include <thread>  #include <mutex>  using namespace std;  #define TEST\_COUNT 3  #define SIZE 5\*pow(10,7)  #define MIN\_VALUE (int)pow(10,5)  #define MAX\_VALUE (int)pow(10,6)  #define THREAD\_COUNT 4  std::mutex mx;  int\* array\_ = new int[SIZE];  bool isPrime(int number)  {  for (int i = 2; i < number / 2 + 1; i++)  {  if (number % i == 0)  {  return false;  }  }  return true;  }  int maxPrimeDivider(int number)  {  if (isPrime(number))  {  return number;  }  int max = 1;  for (int i = sqrt(number) + 1; i > 1; i--)  {  if (number % i == 0)  {    if (isPrime(i) && max < i) max = i;  if (i != (number / i) && number % (number / i) == 0)  {    if (isPrime(  number % (number / i)  ) && max <  number % (number / i)  ) max =  number % (number / i);  }  }  }  return max;  }  void function\_(int number)  {  std::cout << "Thread " << std::this\_thread::get\_id() << " start..." << std::endl;    for (int i = number; i < SIZE; i += THREAD\_COUNT)  {  array\_[i] = maxPrimeDivider(array\_[i]);  }  std::cout << "Thread " << std::this\_thread::get\_id() << " finish!" << std::endl;    return;  }  int main()  {  double averageTime = 0;  for (int i = 0; i < TEST\_COUNT; i++)  {  for (int i = 0; i < SIZE; i++)  {  array\_[i] = (rand() % (MIN\_VALUE + rand() % MAX\_VALUE));  }  std::chrono::time\_point<std::chrono::high\_resolution\_clock> start, end;  start = std::chrono::high\_resolution\_clock::now();  thread\* threads = new thread[THREAD\_COUNT];  for (int i = 0; i < THREAD\_COUNT; i++)  {  threads[i] = thread(function\_, i);    }    for (int i = 0; i < THREAD\_COUNT; i++)  {    threads[i].join();  }  end = std::chrono::high\_resolution\_clock::now();  std::chrono::duration<double> diff = end - start;  averageTime += diff.count();  cout << diff.count() << endl;  }  cout << "average time = " << (averageTime / TEST\_COUNT) << endl;  cin.get();  } |

Таблица 1 – Время обработки массивов в секундах

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размерность | Число потоков | | | | |
| 1 | 2 | 4 | 8 | 12 |
| 2·107 | 67,2358 | 34,3606 | 18,8213 | 12,5953 | 11,2178 |
| 5·107 | 168,16 | 85,8833 | 46,7 | 31,3562 | 28,4645 |
| 108 | 336,35 | 171,068 | 93,828 | 63,2507 | 56,3262 |

Замечание. Если N не делится на число потоков, то нужно очень аккуратно раздавать нагрузку! Проверяйте, что у Вас два потока одновременно не делают одно и то же, также необходимо проверить, что нет «отдыхающих» потоков.

Вычислить ускорения для каждого значения N (заполнить таблицу 2) и построить диаграмму зависимости ускорения от числа потоков (три графика на одной диаграмме).

Замечание. Ускорение вычисляется как отношение времени работы последовательной программы к времени работы параллельной программы.

Таблица 2 – Ускорение параллельного суммирования массивов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размерность | Число потоков | | | |
| 2 | 4 | 8 | 12 |
| 2·107 | 1,956 | 3,572 | 5,338 | 5,993 |
| 5·107 | 1,958 | 3,6 | 5,362 | 5,907 |
| 108 | 1,966 | 3,584 | 5,317 | 5,971 |

Задание 5

Разработать программу, включающую в себя последовательный и параллельный алгоритм вычисления произведения квадратной матрицы на вектор. Каждый алгоритм реализуется в отдельном методе. Матрица и вектор генерируются некоторым способом по заданной размерности N. Параллельный алгоритм должен учитывать доступное ему количество процессоров. Программа запрашивает у пользователя размерность N, после чего выводит время вычислений для последовательного и параллельного алгоритмов. Сами матрицы выводить не нужно.

Провести тестирование программ на матрицах размерности N = 5000, 10000 и 20000. На каждом примере запустить не менее 3 раз. В таблицы занести среднее время выполнения на одном примере в секундах и ускорение. Построить диаграмму зависимости ускорения от числа потоков (три графика на одной диаграмме).

Код:

|  |
| --- |
| #include <iostream>  #include <thread>  #include <mutex>  #include <random>  using namespace std;  const int DIMM = 5000;  int\*\* A = new int\* [DIMM];  int\* B = new int [DIMM];  int\* res = new int[DIMM];  #define TEST\_COUNT 3  int thread\_count = std::thread::hardware\_concurrency();  void clear()  {  //cout << "Start clear.." << endl;  for (int i = 0; i < DIMM; i++)  {  B[i] = 0;  res[i] = 0;  // A2[i] = 0;  for (int j = 0; j < DIMM; j++)  {  A[i][j] = 0;  }  }  }  void display(int\*\* A)  {  cout << endl;  for (int i = 0; i < DIMM; i++)  {  for (int j = 0; j < DIMM; j++)  {  cout << A[i][j] << " ";  }  cout << endl;  }  }  void display(int\* A)  {  cout << endl;  for (int i = 0; i < DIMM; i++)  {    cout << A[i] << " ";    }  }  void start\_()  {  srand(time(0));  for (int i = 0; i < DIMM; i++)  {  A[i] = new int[DIMM];  }  }  void init()  {  srand(time(0));  for (int i = 0; i < DIMM; i++)  {  B[i] = rand() % 25;  res[i] = 0;  for (int j = 0; j < DIMM; j++)  {  A[i][j] = rand() % 25;  }  }  }  void multiplication(int\*\* A, int\* B)  {  cout << "Start mult.." << endl;  for (int i = 0; i < DIMM; ++i)  {  for (int j = 0; j < DIMM; ++j)  {  res[i] += A[i][j] \* B[j];  }  }  }  void multiplication2(int number)  {  cout << "Start mult2.." << endl;  int i, j;  for (i = number; i < DIMM;i+= thread\_count) {    for ( j = 0; j < DIMM; ++j)  {  res[i] += A[i][j] \* B[j];  }  }  }  int main()  {  std::cout << "The number of concurrent threads is " << thread\_count << "\n";  std::chrono::time\_point<std::chrono::high\_resolution\_clock> start, end;  double averageTime = 0;  start\_();  for (int i = 0; i < TEST\_COUNT; i++)  {  init();  start = std::chrono::high\_resolution\_clock::now();  multiplication(A, B);  end = std::chrono::high\_resolution\_clock::now();  std::chrono::duration<double> diff = end - start;  averageTime += diff.count();  cout << diff.count() << endl;  clear();  }  cout << "average time = " << (averageTime / TEST\_COUNT) << endl;  averageTime = 0;  thread \*threads = new thread[thread\_count];  for (int i = 0; i < TEST\_COUNT; i++)  {  init();  start = std::chrono::high\_resolution\_clock::now();  for (int i = 0; i < thread\_count; i++)  {  threads[i] = thread(multiplication2, i);  }  for (int i = 0; i < thread\_count; i++)  {  threads[i].join();  }  end = std::chrono::high\_resolution\_clock::now();  std::chrono::duration<double> diff = end - start;  averageTime += diff.count();  cout << diff.count() << endl;  clear();  }  cout << "average time = " << (averageTime / TEST\_COUNT) << endl;  for (int i = 0; i < DIMM; ++i) {  delete[] A[i];  }  delete[] A;  delete[] B;  delete[] res;  cin.get();  } |

Таблица 3 – Время выполнения алгоритма умножения матрицы на вектор

|  |  |  |
| --- | --- | --- |
| Размерность | Алгоритм | |
| последовательный | параллельный |
| 5000 | 0,0139851 | 0,0078917 |
| 10000 | 0,047971 | 0,024981 |
| 20000 | 0,191498 | 0,0877634 |

Таблица 4 – Ускорение работы программы от количества потоков

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размерность | Число потоков | | | |
| 2 | 4 | 8 | 12 |
| 5000 | 1,2336 | 1,7153 | 1,7622 | 1,7721 |
| 10000 | 1,167024284 | 1,98471672 | 1,999908282 | 1,95339159 |
| 20000 | 1,553118841 | 2,1656741 | 2,156421587 | 2,181980188 |

График 1 – ускорения от количества потоков

Задание 6

Написать программу, вычисляющую сумму чисел от 1 до N. Число N вводится как параметр командной строки. Замерить время работы программы. Результат вывести на консоль.

Распараллелить вычисление суммы с использованием std::thread. Результат, полученный каждым потоком, записать в разделяемую переменную. Для синхронизации потоков использовать std::mutex.

Замерить время работы программы для N = 107, 108 и 109 на 1, 2, 4 и 8 потоках. На каждом примере запустить не менее 3 раз. В таблицы занести среднее время выполнения на одном примере в секундах и ускорение. Построить диаграмму зависимости ускорения от числа потоков (три графика на одной диаграмме).

Код:

|  |
| --- |
| #include <iostream>  #include <thread>  #include <mutex>  using namespace std;  #define SIZE pow(10, 7)  #define TEST\_COUNT 3  #define THREAD\_COUNT 4  int\* array\_ = new int[SIZE];  std::mutex mx; int itterator = 0;  double sum = 0;  void function\_(int number)  {  int sum2 = 0;  for (int i = number; i <= SIZE; i += THREAD\_COUNT)  {  sum2 += i;  }  mx.lock();  sum+=sum2;  mx.unlock();  }  int main()  {  std::chrono::time\_point<std::chrono::high\_resolution\_clock> start, end;  double averageTime = 0;  thread\* threads = new thread[THREAD\_COUNT];  for (int i = 0; i < TEST\_COUNT; i++)  {  sum = 0;  start = std::chrono::high\_resolution\_clock::now();  for (int i = 0; i < THREAD\_COUNT; i++)  {  threads[i] = thread(function\_, i);  }  for (int i = 0; i < THREAD\_COUNT; i++)  {  threads[i].join();  }  end = std::chrono::high\_resolution\_clock::now();  std::chrono::duration<double> diff = end - start;  averageTime += diff.count();  cout << diff.count() << endl;  }  cout << "average time = " << (averageTime / TEST\_COUNT) << endl;  cout << "SUM = " << sum << endl;  } |

Таблица 5 – Время параллельного суммирования чисел в секундах

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размерность | Число потоков | | | | |
| 1 | 2 | 4 | 8 | 12 |
| 107 | 0,00719433 | 0,00591053 | 0,00519273 | 0,00554463 | 0,00762517 |
| 108 | 0,0686234 | 0,035157 | 0,0177543 | 0,00980687 | 0,00826943 |
| 109 | 0,683933 | 0,351497 | 0,174635 | 0,0915043 | 0,0909826 |

Таблица 6 – Ускорение параллельного суммирования чисел в секундах

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размерность | Число потоков | | | |
| 2 | 4 | 8 | 12 |
| 107 | 1,217 | 1,385 | 1,297 | 0,943 |
| 108 | 1,951 | 3,865 | 6,997 | 8,29 |
| 109 | 1,945 | 3,916 | 7,474 | 7,517 |

График 2 – ускорение от количества потоков