**Программное обеспечение, использованное при выполнении:** Visual Studio 2017, IPS 2019, Windows 10 – 64x

**Процессор** – четырехъядерный Intel Core i5 8250U с частотой 1.6 ГГц

1. Разберите пример программы нахождения максимального элемента массива и его индекса task\_for\_lecture2.cpp. Предварительный просмотр документа. Запустите программу и убедитесь в корректности ее работы.

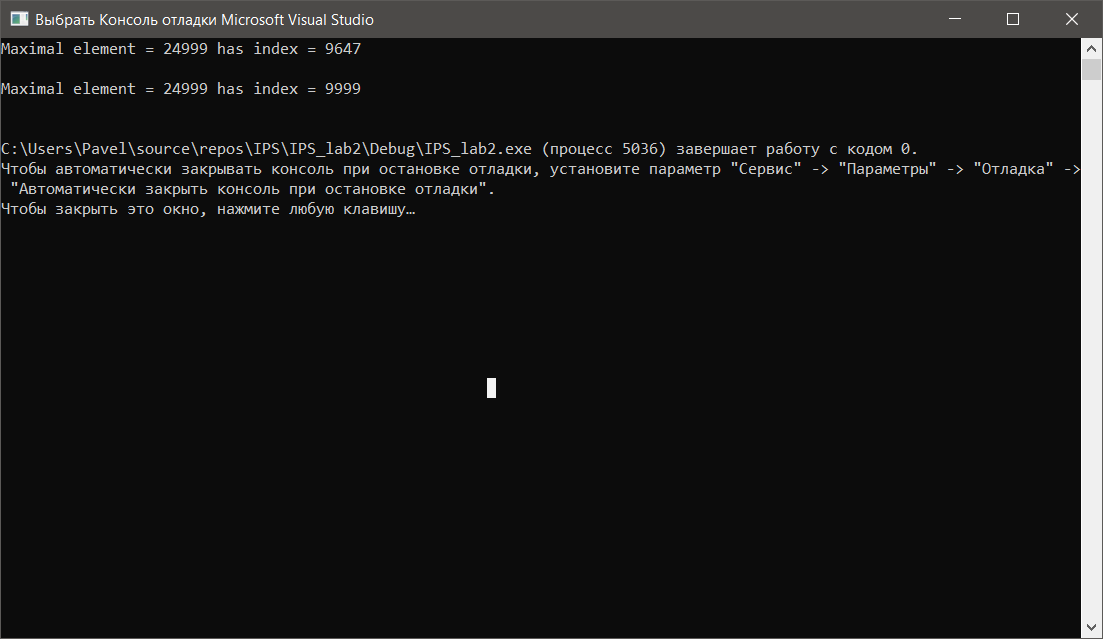


Рисунок 1 – пример работы программы

2. По аналогии с функцией ReducerMaxTest(…), реализуйте функцию ReducerMinTest(…) для нахождения минимального элемента массива и его индекса. Вызовите функцию ReducerMinTest(…) до сортировки исходного массива mass и после сортировки. Убедитесь в правильности работы функции ParallelSort(...): индекс минимального элемента после сортировки должен быть равен 0, индекс максимального элемента (mass\_size - 1).

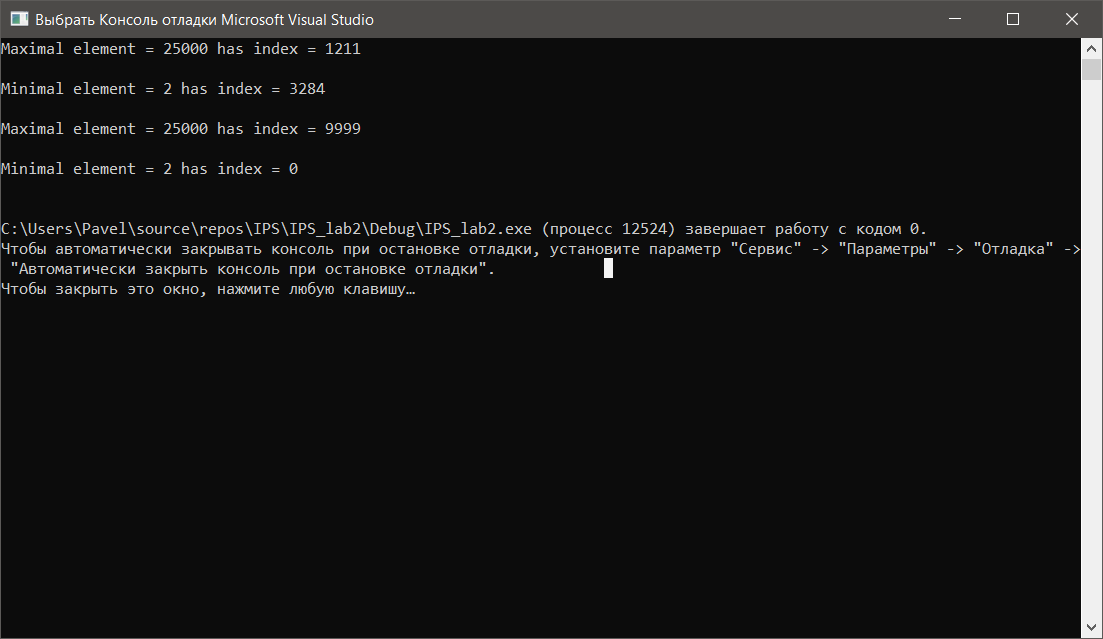


Рисунок 2 – пример работы программы

3. Добавьте в функцию ParallelSort(...) строки кода для измерения времени, необходимого для сортировки исходного массива. Увеличьте количество элементов mass\_size исходного массива mass в 10, 50, 100 раз по сравнению с первоначальным. Выводите в консоль время, затраченное на сортировку массива, для каждого из значений mass\_size. Рекомендуется засекать время с помощью библиотеки chrono.

В таблице, представлена зависимость времени, затраченного на выполнение программы, в зависимости от размера программы

Таблица 1 – зависимость времени выполнения от размера массива

|  |  |
| --- | --- |
| **Размер массива** | **Время, с** |
| 10000 | 0.0106344 |
| 100000 | 0.143007 |
| 500000 | 0.710581 |
| 1000000 | 1.54245 |

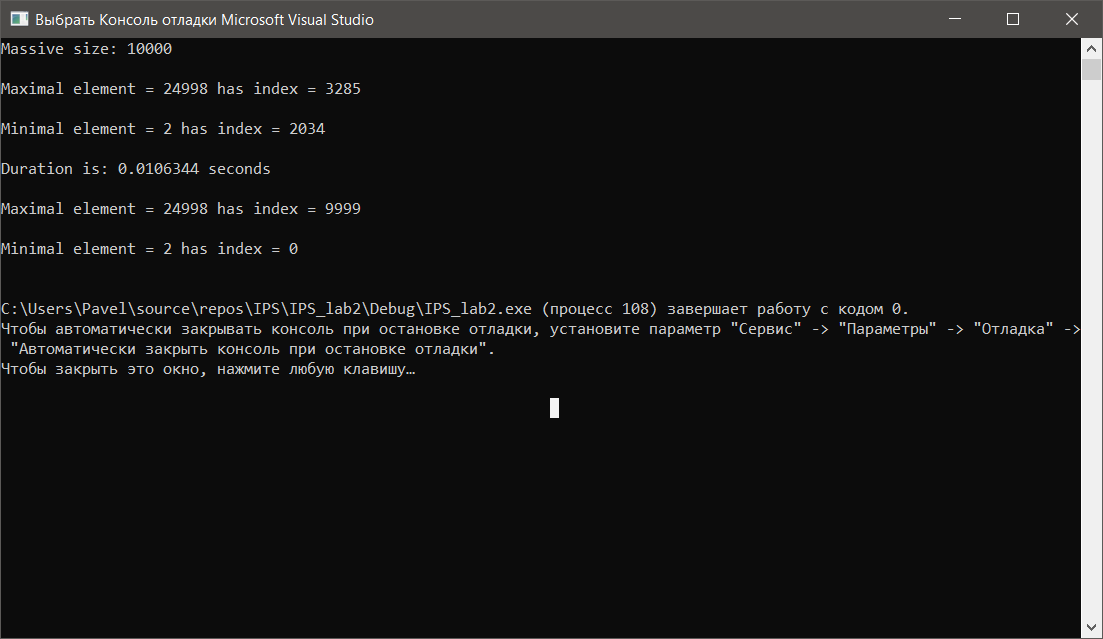


Рисунок 3 – пример работы программы при массиве из 10000 элементов

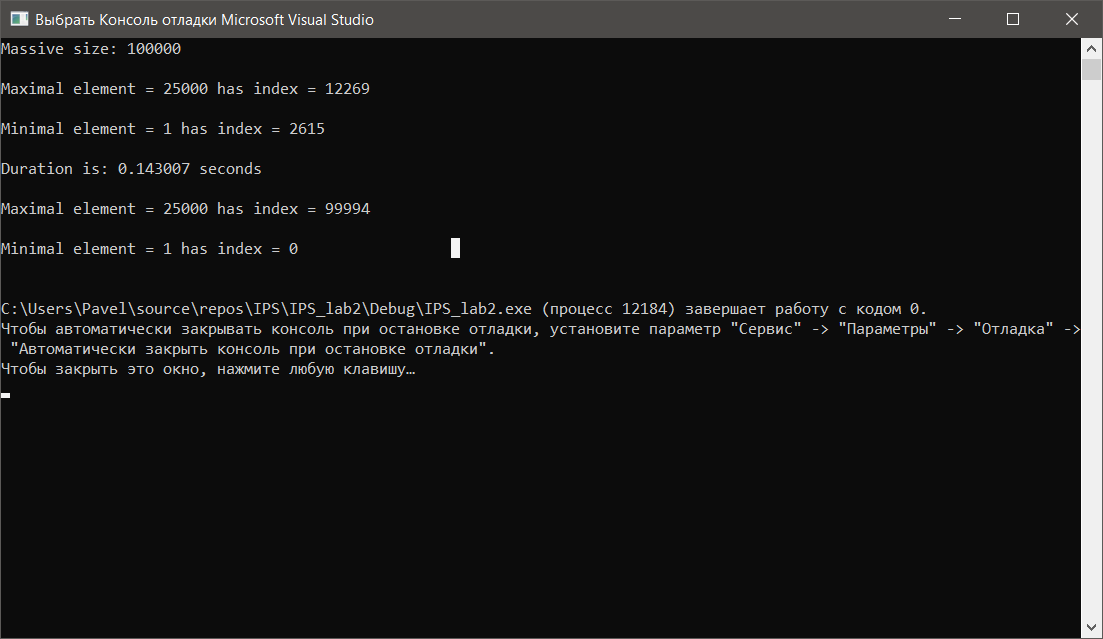


Рисунок 4 – пример работы программы при массиве из 100000 элементов

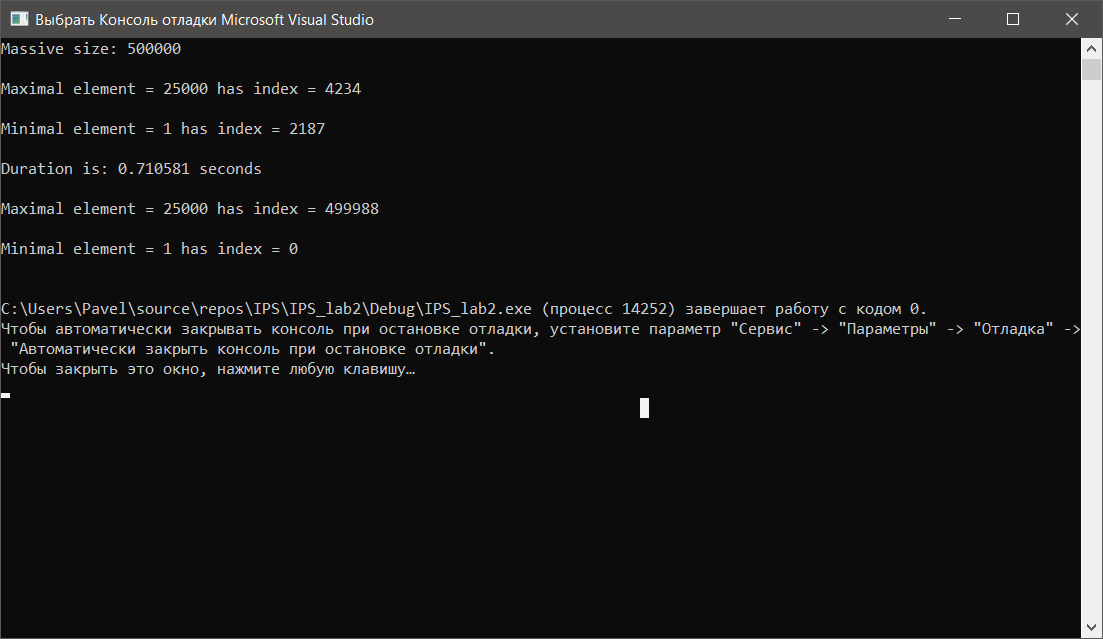


Рисунок 5 – пример работы программы при массиве из 500000 элементов

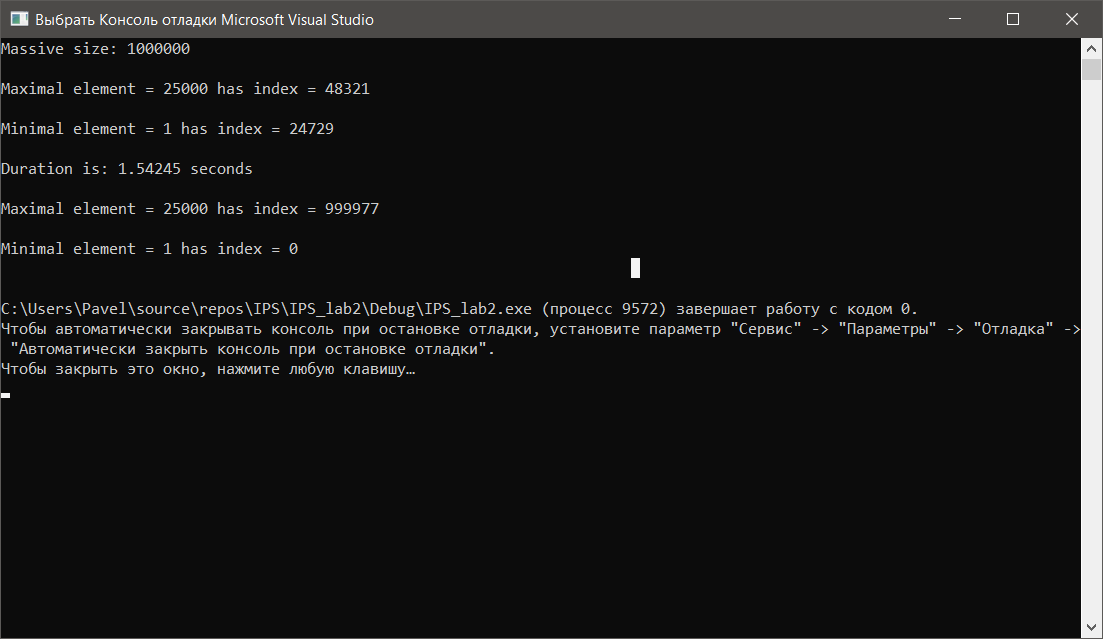


Рисунок 6 – пример работы программы при массиве из 1000000 элементов

4. Реализуйте функцию CompareForAndCilk\_For(size\_t sz). Эта функция должна выводить на консоль время работы стандартного цикла for, в котором заполняется случайными значениями std::vector (использовать функцию push\_back(rand() % 20000 + 1)), и время работы параллельного цикла cilk\_for от Intel Cilk Plus, в котором заполняется случайными значениями reducer вектор.

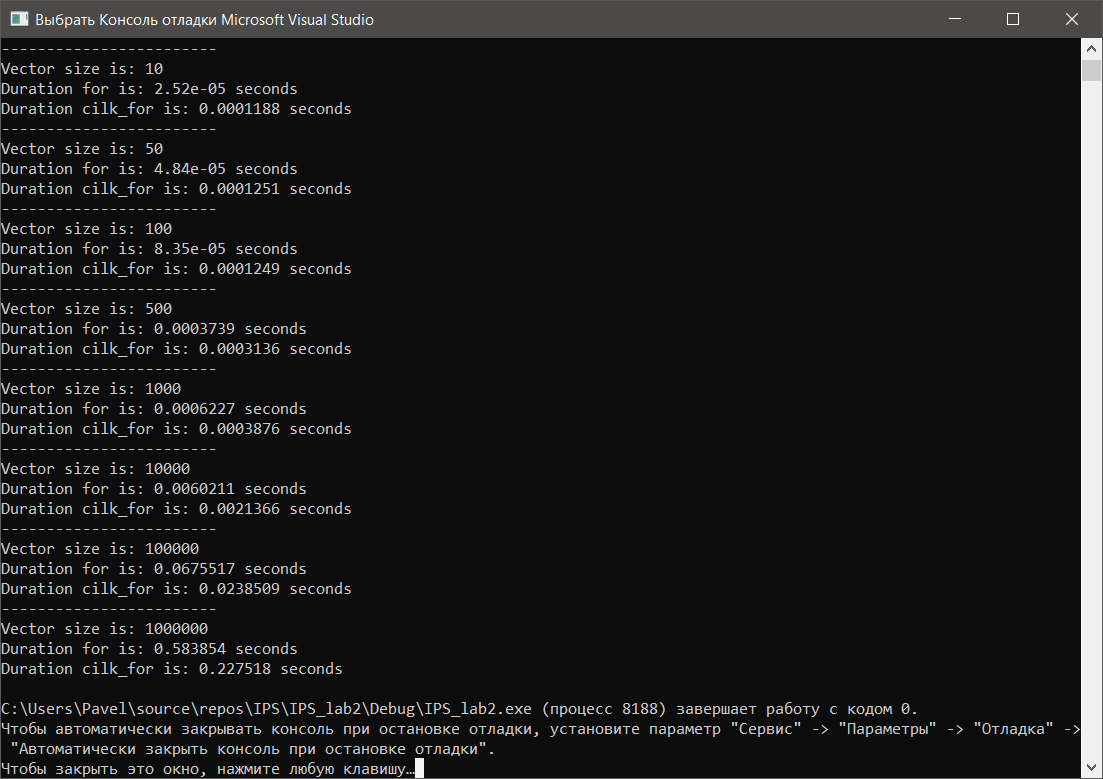


Рисунок 7 – сравнение cilk\_for и for в зависимости от размера массива

На рисунке видно, что при размерности вектора меньше 500, желательно использовать цикл for, при размерности массива данных больше 500, выгоднее использовать цикл cilk\_for.

5. Ответьте на вопросы: почему при небольших значениях ***sz***цикл ***cilk\_for***уступает циклу ***for***в быстродействии? В каких случаях целесообразно использовать цикл ***cilk\_for***? В чем принципиальное отличие параллелизации с использованием ***cilk\_for*** от параллелизации с использованием ***cilk\_spawn***в паре с ***cilk\_sync***?

1) Почему при небольших значениях ***sz***цикл ***cilk\_for***уступает циклу ***for***в быстродействии?

Потому что, операция создания потоков и распределения вычислительных задач между ними при использовании ***cilk\_for***, занимает некоторое время и в случае с небольшим размером массива данных, не имеет особого выигрыша по времени, а иногда даже на порядок выше.

2) В каких случаях целесообразно использовать цикл ***cilk\_for***?

Cilk\_for гораздо выгоднее использовать с массивами данных большой размерности, на моем железе выигрыш в скорости идет уже при размерности больше 500. Соответственно, чем больше размерность, тем больше выигрыш в скорости выполнения. Cilk\_ for выгодно будет использовать в алгоритмах брутфорса, где нужно просто перебирать много данных.

3) В чем принципиальное отличие параллелизации с использованием ***cilk\_for*** от параллелизации с использованием ***cilk\_spawn***в паре с ***cilk\_sync***?

Ключевое слово cilk\_spawn сообщает компилятору, что функция, которой предшествует cilk\_spawn, может работать параллельно с вызывающей функцией.

cilk\_sync – это точка синхронизации, которая заставляет, созданные ранее задачи, ждать завершения друг друга.

Пример использования, данных конструкций, приведен в таблице 2. В примере слово Done гарантировано выведется после того, как будут выполнены функции hello() и world()

Таблица 2 – пример использования cilk\_spawn и cilk\_sync

|  |
| --- |
| #include <iostream>  #include <cilk/cilk.h>  using namespace std;    int main(){  cilk\_spawn hello();  cilk\_spawn world();  cilk\_sync;  cout << "Done! ";  } |

Конструкция cilk\_for предназначена, для введения параллелизма в циклах for.

Четкая разница между cilk\_for и cilk\_spawn вместе с cilk\_sync видна на примере ниже:

|  |
| --- |
| for (int x = 0; x < n; ++x) { cilk\_spawn f(x); }  cilk\_for (int x = 0; x < n; ++x) { f(x); } |

В первом случае, мы на каждой итерации будем создавать по задаче, а операция захвата чужой задачи весьма затратная с точки зрения производительности. Если в каждой итерации «мало» работы, то мы больше потеряем, чем получим с помощью такой «параллельной» программы.

**Вывод:** при выполнении задания, я познакомился с функцией **ReducerMinTest(…).** Сравнил, затраченное на сортировку массива при разной размерности массива данных. Реализовал функцию **CompareForAndCilk\_For(size\_t sz)**, которая замеряет время, затраченное на выполнение циклов for и cilk\_for при разных размерностях массива. Также стало очевидным, что cilk\_for гораздо выгоднее использовать при больших размерах массива.