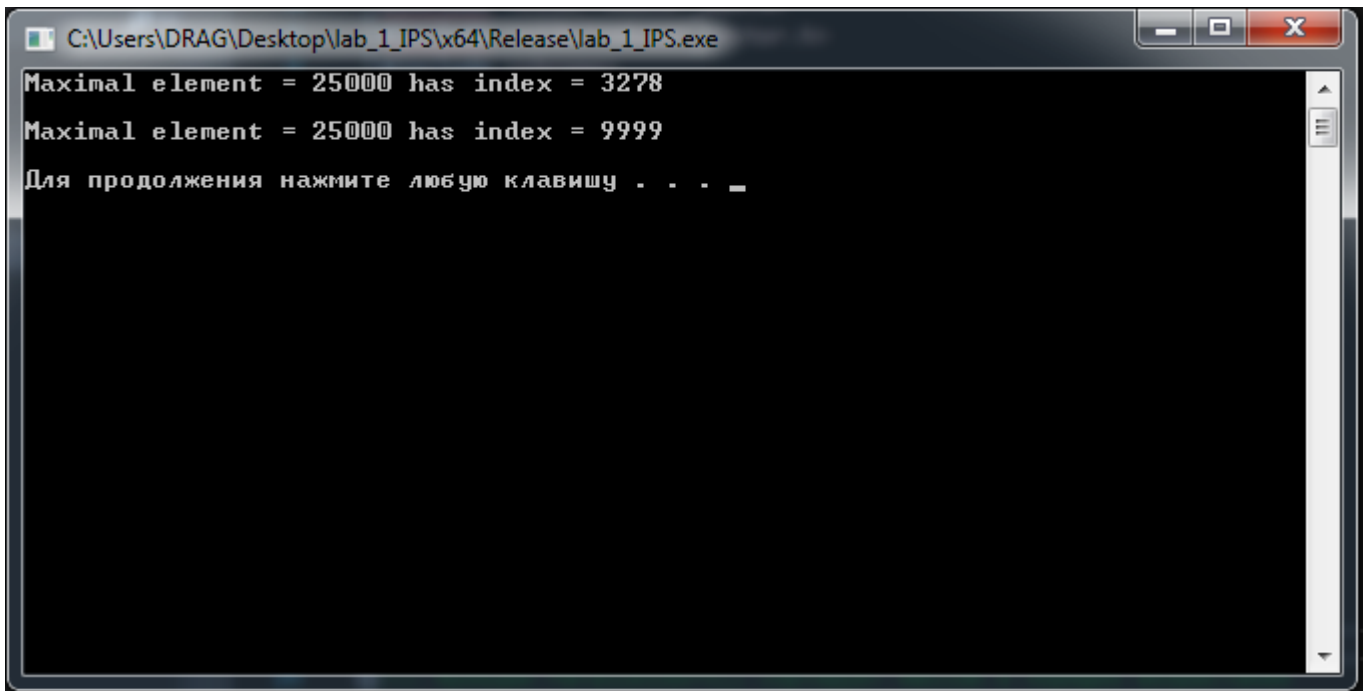


## Лабораторная работа №1

Выполнил: Прохоров А.В.

Группа: ПМ-21М

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



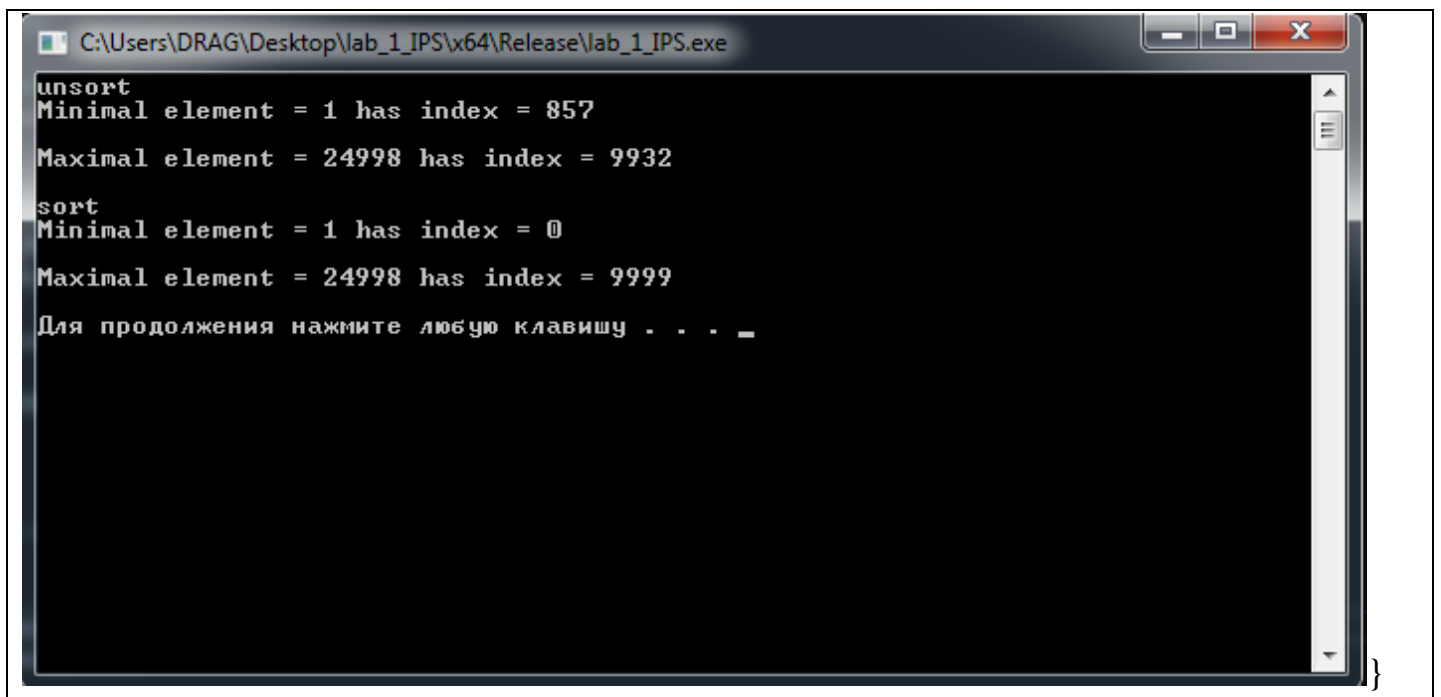
```
C:\Users\DRAG\Desktop\lab_1_IPS\x64\Release\lab_1_IPS.exe
Maximal element = 25000 has index = 3278
Maximal element = 25000 has index = 9999
Для продолжения нажмите любую клавишу . . . _
```

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**).

Функция **ReducerMinTest**:

```
/// ReducerMinTest()
void ReducerMinTest(int *mass_pointer, const long size)
{
    cilk::reducer<cilk::op_min_index<long, int>> minimum;
    cilk_for(long i = 0; i < size; ++i)
        minimum->calc_min(i, mass_pointer[i]);

    printf("Minimal element = %d has index = %d\n\n",
        minimum->get_reference(), minimum->get_index_reference());
}
```



```
C:\Users\DRAG\Desktop\lab_1_IPS\x64\Release\lab_1_IPS.exe
unsort
Minimal element = 1 has index = 857
Maximal element = 24998 has index = 9932
sort
Minimal element = 1 has index = 0
Maximal element = 24998 has index = 9999
Для продолжения нажмите любую клавишу . . . _
```

Видно, что функции работают правильно.

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*

```
printf("sort\n");
printf("size of array: %d\n", mass_size);

auto begin_time = high_resolution_clock::now();
ParallelSort(mass_begin, mass_end);
auto end_time = high_resolution_clock::now();
duration<double> time = end_time - begin_time;
printf("duration of execution: %f sec\n", time.count());

ReducerMinTest(mass, mass_size);
ReducerMaxTest(mass, mass_size);
```

```
sort
size of array: 100000
duration of execution: 0.002196 sec
Minimal element = 1 has index = 0
Maximal element = 24993 has index = 9998
```

```
sort
size of array: 500000
duration of execution: 0.009837 sec
Minimal element = 1 has index = 0
Maximal element = 250000 has index = 49997
```

```
sort
size of array: 100000
duration of execution: 0.015080 sec
Minimal element = 1 has index = 0
Maximal element = 25000 has index = 99994
```

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. Вызывайте функцию *CompareForAndCilk\_For()* для входного

параметра *sz* равного: **1000000, 100000, 10000, 1000, 500, 100, 50, 10**. Проанализируйте результаты измерения времени, необходимого на заполнение *std::vector'a* и *reducer* вектора.

```
/// Функция CompareForAndCilk_For() выводит на консоль время работы стандартного цикла for
/// в котором заполняется случайными значениями std::vector и время
/// параллельного цикла cilk_for от Intel Cilk Plus
void CompareForAndCilk_For(size_t sz)
{
    std::vector<int> vec;
    auto begin_time_for = high_resolution_clock::now();

    for (size_t i = 0; i < sz; ++i)
        vec.push_back(rand() % 20000 + 1);

    auto end_time_for = high_resolution_clock::now();
    const duration<double> time_for = end_time_for - begin_time_for;

    cilk::reducer<cilk::op_vector<int>> red_vec;

    auto begin_time_cilk_for = high_resolution_clock::now();

    cilk_for(size_t i = 0; i < sz; ++i)
        red_vec->push_back(rand() % 20000 + 1);

    auto end_time_cilk_for = high_resolution_clock::now();
    const duration<double> time_cilk_for = end_time_cilk_for - begin_time_cilk_for;

    printf("Size of array: %d\n", sz);
    printf("time 'for' = %f sec\n", time_for.count());
    printf("time 'cilk_for' = %f sec\n", time_cilk_for.count());
}
```

```

Size of array: 1000000
time 'for' = 0.033693 sec
time 'cilk_for' = 0.023038 sec

Size of array: 100000
time 'for' = 0.003191 sec
time 'cilk_for' = 0.002893 sec

Size of array: 10000
time 'for' = 0.000347 sec
time 'cilk_for' = 0.001834 sec

Size of array: 1000
time 'for' = 0.000036 sec
time 'cilk_for' = 0.000050 sec

Size of array: 500
time 'for' = 0.000019 sec
time 'cilk_for' = 0.000036 sec

Size of array: 100
time 'for' = 0.000007 sec
time 'cilk_for' = 0.000014 sec

Size of array: 50
time 'for' = 0.000007 sec
time 'cilk_for' = 0.000014 sec

Size of array: 10
time 'for' = 0.000003 sec
time 'cilk_for' = 0.000009 sec

```

5. Ответьте на вопросы: почему при небольших значениях *sz* цикл ***cilk\_for*** уступает циклу ***for*** в быстродействии?

При выполнении ***cilk\_for*** компилятор разбивает его на мелкие блоки, фиксированного размера. Если количество итераций, которые должен сделать цикл мало, тогда накладные ресурсы на разбиение и планировку начинают занимать много времени, в результате обычный ***for*** работает быстрее.

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

При малом количестве итераций в цикле.

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

***cilk\_for*** использует алгоритм «разделяй и властвуй». На каждом уровне рекурсии поток выполняет половину работы, а остальную передает потомкам. Вызов ***cilk\_spawn*** обозначает точку порождения. Она создает новую задачу, выполнение которой может быть продолжено либо данным потоком, либо выполнено новым потоком. Это ключевое слово является указанием системе исполнения на то, что данная функция может выполняться параллельно с функцией, из которой она вызвана.