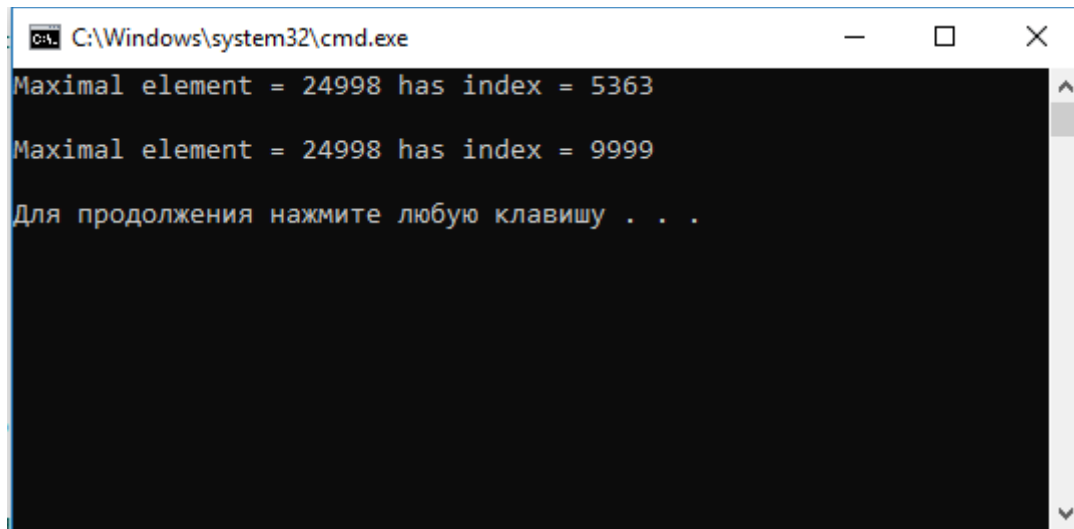


1. Разберите пример программы нахождения максимального элемента массива и его индекса. Запустите программу и убедитесь в корректности ее работы.

Запустим программу и убедимся в корректности ее работы.



```

C:\Windows\system32\cmd.exe
Maximal element = 24998 has index = 5363
Maximal element = 24998 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).

Реализуем функцию *ReducerMinTest(...)*.

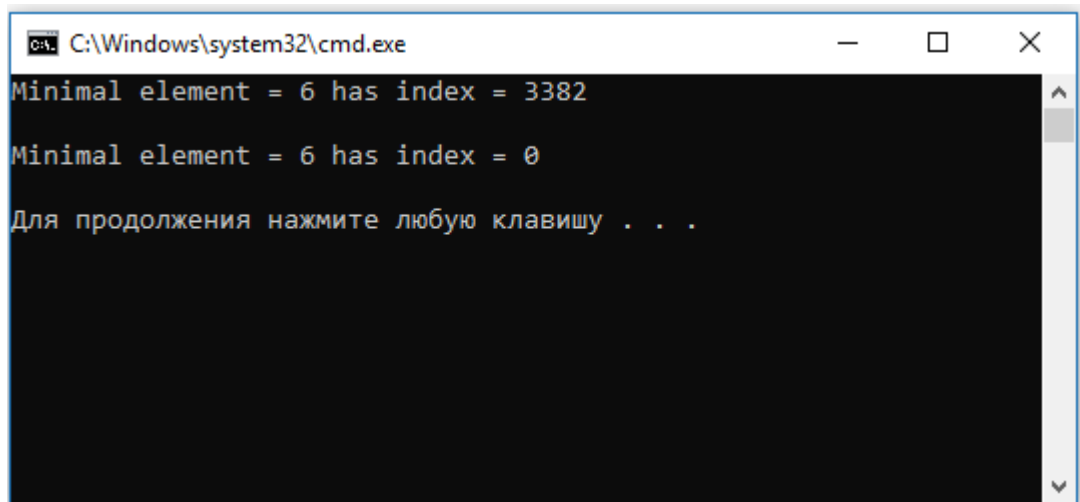
```

// Функция ReducerMinTest() определяет минимальный элемент массива,
// переданного ей в качестве аргумента, и его позицию
// mass_pointer - указатель исходный массив целых чисел
// size - количество элементов в массиве
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());
}

```

Вызовем данную функцию до сортировки исходного массива и после сортировки.



```

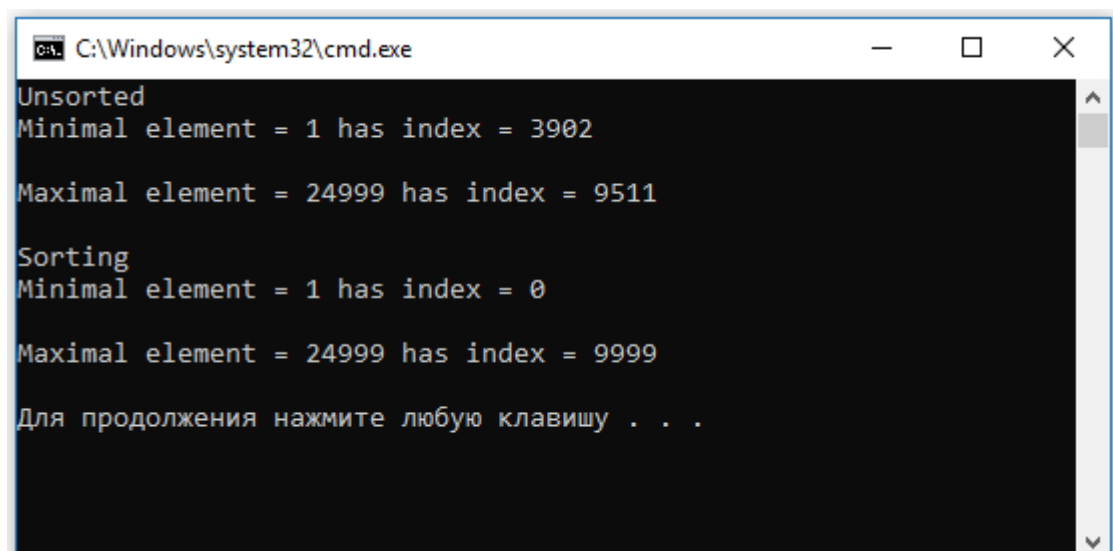
C:\Windows\system32\cmd.exe
Minimal element = 6 has index = 3382

Minimal element = 6 has index = 0

Для продолжения нажмите любую клавишу . . .

```

Убедимся в правильности работы функции *ParallelSort(...)*.



```

C:\Windows\system32\cmd.exe
Unsorted
Minimal element = 1 has index = 3902

Maximal element = 24999 has index = 9511

Sorting
Minimal element = 1 has index = 0

Maximal element = 24999 has index = 9999

Для продолжения нажмите любую клавишу . . .

```

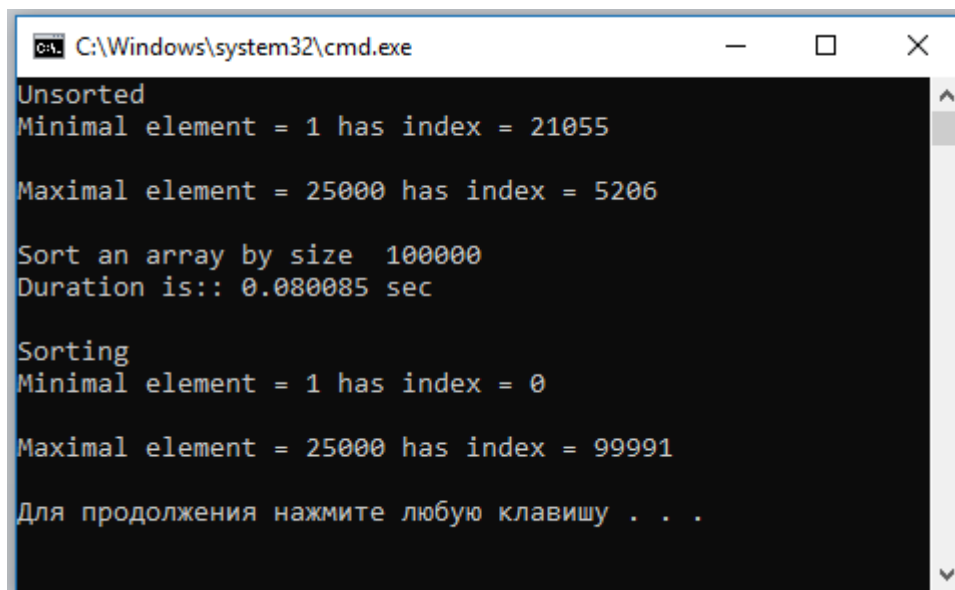
Как видно, индекс минимального элемента после сортировки равен 0, а индекс максимального элемента ($\text{mass_size} - 1$).

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 .

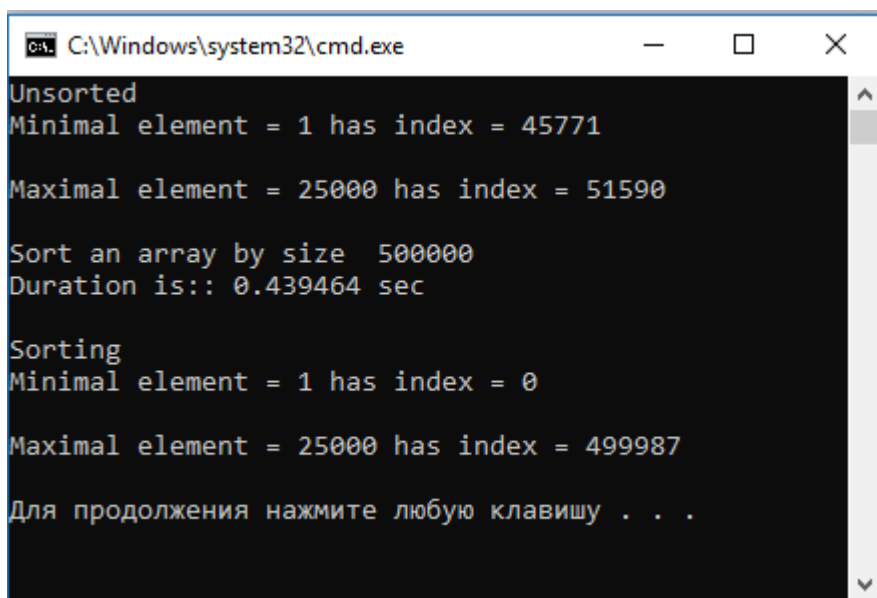
Добавим в функцию **ParallelSort(...)** строки кода для измерения времени, необходимого для сортировки исходного массива.

```
high_resolution_clock::time_point t1 = high_resolution_clock::now();
ParallelSort(mass_begin, mass_end);
high_resolution_clock::time_point t2 = high_resolution_clock::now();
duration<double> duration = (t2 - t1);
printf("Duration is:: %lf sec\n\n", duration.count());
```

Увеличим количество элементов исходного массива в 10, 50, 100 раз по сравнению с первоначальным.



```
C:\Windows\system32\cmd.exe
Unsorted
Minimal element = 1 has index = 21055
Maximal element = 25000 has index = 5206
Sort an array by size 100000
Duration is:: 0.080085 sec
Sorting
Minimal element = 1 has index = 0
Maximal element = 25000 has index = 99991
Для продолжения нажмите любую клавишу . . .
```



```
C:\Windows\system32\cmd.exe
Unsorted
Minimal element = 1 has index = 45771
Maximal element = 25000 has index = 51590
Sort an array by size 500000
Duration is:: 0.439464 sec
Sorting
Minimal element = 1 has index = 0
Maximal element = 25000 has index = 499987
Для продолжения нажмите любую клавишу . . .
```

```

C:\Windows\system32\cmd.exe
Unsorted
Minimal element = 1 has index = 37990
Maximal element = 25000 has index = 38377
Sort an array by size 1000000
Duration is:: 1.153990 sec
Sorting
Minimal element = 1 has index = 0
Maximal element = 25000 has index = 999974
Для продолжения нажмите любую клавишу . . .

```

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`'а и `reducer` вектора.

Реализуем функцию `CompareForAndCilk_For(size_t sz)`

```

// Функция CompareForAndCilk_For выводит в консоль время работы
// стандартного цикла for и время работы параллельного цикла cilk_for
// sz - количество элементов в каждом векторе
void CompareForAndCilk_For(size_t sz)
{
    high_resolution_clock::time_point t1, t2;

    std::vector<int> vec;
    t1 = high_resolution_clock::now();
    for (size_t i = 0; i < sz; ++i)
    {
        vec.push_back(rand() % 20000 + 1);
    }
    t2 = high_resolution_clock::now();
    duration<double> duration_for = (t2 - t1);

    cilk::reducer<cilk::op_vector<int>>red_vec;
    t1 = high_resolution_clock::now();
    cilk_for (size_t i = 0; i < sz; ++i)
    {

```

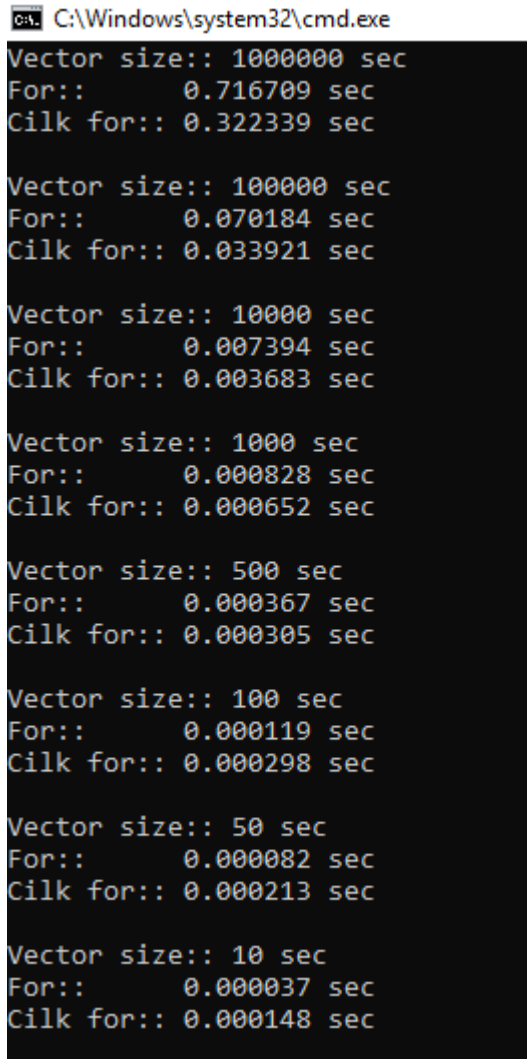
```

        red_vec->push_back(rand() % 20000 + 1);
    }
    t2 = high_resolution_clock::now();
    duration<double> duration_cilk_for = (t2 - t1);

    printf("Vector size:: %d sec\n", sz);
    printf("For::      %lf sec\n", duration_for.count());
    printf("Cilk for:: %lf sec\n\n", duration_cilk_for.count());
}

```

Полученные результаты работы



```

C:\Windows\system32\cmd.exe
Vector size:: 1000000 sec
For::      0.716709 sec
Cilk for:: 0.322339 sec

Vector size:: 100000 sec
For::      0.070184 sec
Cilk for:: 0.033921 sec

Vector size:: 10000 sec
For::      0.007394 sec
Cilk for:: 0.003683 sec

Vector size:: 1000 sec
For::      0.000828 sec
Cilk for:: 0.000652 sec

Vector size:: 500 sec
For::      0.000367 sec
Cilk for:: 0.000305 sec

Vector size:: 100 sec
For::      0.000119 sec
Cilk for:: 0.000298 sec

Vector size:: 50 sec
For::      0.000082 sec
Cilk for:: 0.000213 sec

Vector size:: 10 sec
For::      0.000037 sec
Cilk for:: 0.000148 sec

```

Как видно, из полученных результатов при больших значениях размера вектора параллельный цикл `cilk_for` работает примерно в 2 раза быстрее, чем стандартный цикл `for`. При небольших значениях размера вектора параллельный цикл `cilk_for` работает в разы медленнее, чем стандартный цикл `for`.

5. Ответьте на вопросы: почему при небольших значениях `sz` цикл `cilk_for` уступает циклу `for` в быстродействии?

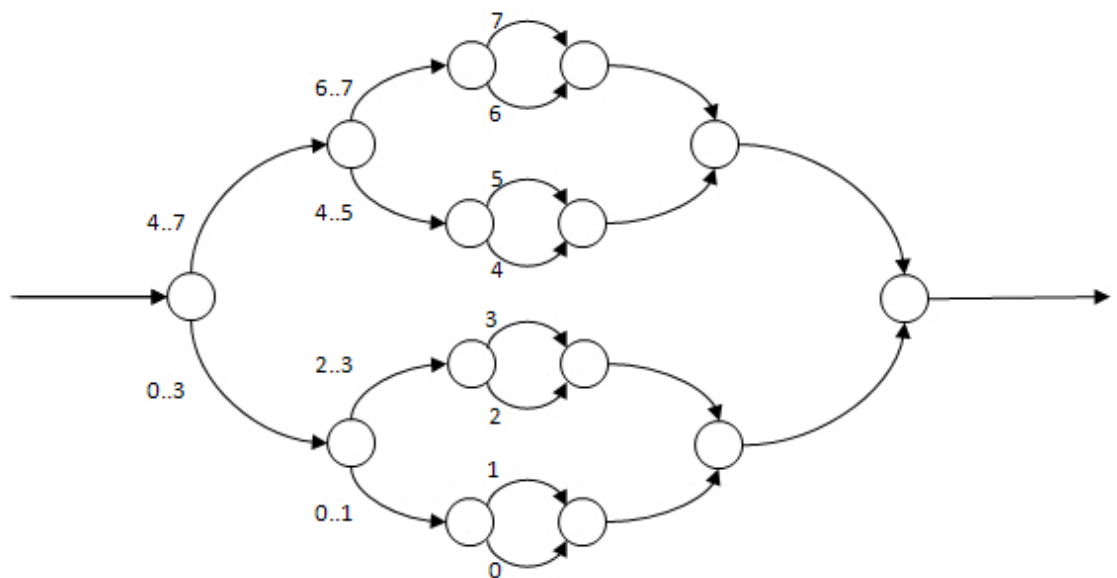
Когда выполняется цикл `cilk_for`, компилятор рекурсивно разбивает каждый цикл пополам, пока количество итераций каждого цикла не станет меньше или равно размеру зерна. Поэтому при небольших размерах цикла накладные расходы на разбиение и на планировку занимают больше времени, чем стандартный цикл `for`.

В каких случаях целесообразно использовать цикл `cilk_for` ?

Цикл `cilk_for` целесообразно использовать при большой разнице между последним и первым значением счетчика цикла.

В чем принципиальное отличие параллелизации с использованием `cilk_for` от параллелизации с использованием `cilk_spawn` в паре с `cilk_sync`?

Главное отличие состоит в том, что `cilk_for` использует алгоритм «разделяй и властвуй». На каждом уровне рекурсии половина оставшейся работы выполняется потомком, вторая половина – продолжением.



Вызов `cilk_spawn` обозначает точку порождения. В этой точке создается новая задача, выполнение которой может быть продолжено данным потоком или захвачено другим параллельным потоком. Это ключевое слово является указанием системе исполнения на то, что данная функция может, но не обязана выполняться параллельно с функцией, из которой она вызвана.