

Лабораторная работа №1

1. Разберите пример программы нахождения максимального элемента массива и его индекса [task_for_lecture2.cpp](#). Запустите программу и убедитесь в корректности ее работы.

```
Unsorted:
Maximal element = 24999 has index = 6466

Sorted:
Maximal element = 24999 has index = 9999
```

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***).

```
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());
}
```

```
Unsorted:
Minimal element = 1 has index = 4510

Sorted:
Minimal element = 1 has index = 0
```

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** засекаать время с помощью библиотеки ***chrono***.

```

auto t0 = high_resolution_clock::now();
ParallelSort(mass_begin, mass_end);
auto t1 = high_resolution_clock::now();
duration<double> duration = t1 - t0;
printf("Size of array: %d\n", mass_size);
printf("Duration is %f seconds\n", duration.count());

```

```

Size of array: 100000
Duration is 0.010596 seconds
Sorted:
Minimal element = 1 has index = 0
Maximal element = 25000 has index = 99998

```

```

Size of array: 500000
Duration is 0.024640 seconds
Sorted:
Minimal element = 1 has index = 0
Maximal element = 25000 has index = 499986

```

```

Size of array: 1000000
Duration is 0.044367 seconds
Sorted:
Minimal element = 1 has index = 0
Maximal element = 25000 has index = 999962

```

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.

```

void CompareForAndCilk_For(size_t sz)
{
    std::vector<int> vec;
    auto t0 = high_resolution_clock::now();
    for (size_t i = 0; i < sz; ++i)
        vec.push_back(rand() % 20000 + 1);

    auto t1 = high_resolution_clock::now();

```

```

    const duration<double> duration_vec = t1 - t0;

    cilk::reducer<cilk::op_vector<int>> red_vec;
    t0 = high_resolution_clock::now();
    cilk_for(size_t i = 0; i < sz; ++i)
        red_vec->push_back(rand() % 20000 + 1);

    t1 = high_resolution_clock::now();
    const duration<double> duration_red_vec = t1 - t0;

    printf("Size of array: %d\n", sz);
    printf("std::vector time: %f seconds\n",
duration_vec.count());
    printf("cilk::reducer time: %f seconds\n",
duration_red_vec.count());
}

```

```

Size of array: 1000000
std::vector time: 0.025973 seconds
cilk::reducer time: 0.012981 seconds

Size of array: 100000
std::vector time: 0.003406 seconds
cilk::reducer time: 0.001638 seconds

Size of array: 10000
std::vector time: 0.000268 seconds
cilk::reducer time: 0.000189 seconds

Size of array: 1000
std::vector time: 0.000020 seconds
cilk::reducer time: 0.000067 seconds

Size of array: 500
std::vector time: 0.000016 seconds
cilk::reducer time: 0.000078 seconds

Size of array: 100
std::vector time: 0.000025 seconds
cilk::reducer time: 0.000025 seconds

Size of array: 50
std::vector time: 0.000005 seconds
cilk::reducer time: 0.000030 seconds

Size of array: 10
std::vector time: 0.000003 seconds
cilk::reducer time: 0.000008 seconds

```

На большом количестве элементов (10000-1000000) cilk_for работает быстрее, на меньшем количестве обычный for работает лучше.

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

При выполнении **cilk_for** компилятор разбивает вектор на мелкие блоки фиксированного размера. Если количество итераций мало, то накладные ресурсы на разбиение и планировку занимают слишком много времени, что приводит к тому, что обычный **for** работает быстрее.

В каких случаях целесообразно использовать цикл **cilk_for** ?

При большом размере вектора.

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 оставшуюся передает потомкам.

Вызов **cilk_spawn** обозначает точку порождения. Она создает новую задачу, выполнение которой может быть продолжено либо данным потоком, либо новым. Это ключевое слово является указанием системе исполнения, что данная функция может выполняться параллельно с той, из которой она была вызвана.