


Задание к занятию 2

Теверовский Михаил, ИВТ-11М

Используемые программы и характеристики ПК:

1. ПО: VisualStudio 2017 и Intel Parallel Studio 2019;
2. ОС: Windows 7 Professional
3. Процессор: Intel core i5-4200 CPU, 2.50 GHz, 2 ядра, 4 потока

Задание 1: Разберите пример программы нахождения максимального элемента массива и его индекса [task for lecture2.cpp](#) . Запустите программу и убедитесь в корректности ее работы.

Результат:

Убедимся в корректности работы:

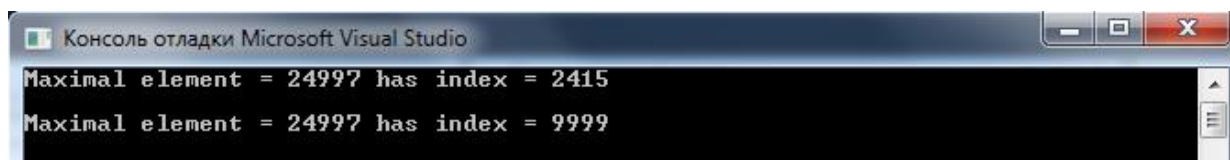


Рисунок 1. – Скриншот результата выполнения задания 1

Анализ полученных результатов:

Программа выполняется корректно. До сортировки и после сортировки находится верно максимальный элемент зарандомленного массива. Изменение индекса связано с изменением последовательности элементов после сортировки.

Задание 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(...)*:

```
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());
}
```

Изменённый код программы путём добавления пользовательской функции *ReducerMaxTest(...)* на базе *ReducerMinTest(...)*:

```

#include <cilk/cilk.h>
#include <cilk/cilk_api.h>
#include <cilk/reducer_max.h>
#include <cilk/reducer_min.h>
#include <cilk/reducer_vector.h>
#include <chrono>

using namespace std::chrono;

/// Функция ReducerMaxTest() определяет максимальный элемент массива,
/// переданного ей в качестве аргумента, и его позицию
/// mass_pointer - указатель исходный массив целых чисел
/// size - количество элементов в массиве
void ReducerMaxTest(int *mass_pointer, const long size)
{
    cilk::reducer<cilk::op_max_index<long, int>> maximum;
    cilk_for(long i = 0; i < size; ++i)
    {
        maximum->calc_max(i, mass_pointer[i]);
    }
    printf("Maximal element = %d has index = %d\n\n",
        maximum->get_reference(), maximum->get_index_reference());
}

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());
}

/// Функция ParallelSort() сортирует массив в порядке возрастания
/// begin - указатель на первый элемент исходного массива
/// end - указатель на последний элемент исходного массива
void ParallelSort(int *begin, int *end)
{
    if (begin != end)
    {
        --end;
        int *middle = std::partition(begin, end, std::bind2nd(std::less<int>()),
*end));
        std::swap(*end, *middle);
        cilk_spawn ParallelSort(begin, middle);
        ParallelSort(++middle, ++end);
        cilk_sync;
    }
}

int main()
{
    srand((unsigned)time(0));

    // устанавливаем количество работающих потоков = 4
    __cilkrts_set_param("nworkers", "4");

    long i;
    const long mass_size = 10000;
    int *mass_begin, *mass_end;
    int *mass = new int[mass_size];

```

```

    for (i = 0; i < mass_size; ++i)
    {
        mass[i] = (rand() % 25000) + 1;
    }

    mass_begin = mass;
    mass_end = mass_begin + mass_size;

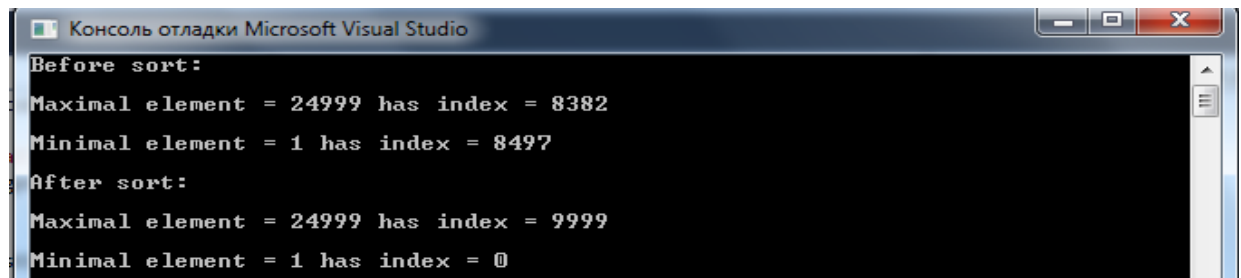
    printf("Before sort:\n\n");
    ReducerMaxTest(mass, mass_size);
    ReducerMinTest(mass, mass_size);

    printf("After sort:\n\n");
    ParallelSort(mass_begin, mass_end);
    ReducerMaxTest(mass, mass_size);
    ReducerMinTest(mass, mass_size);

    delete[] mass;
    return 0;
}

```

Результат:



```

Консоль отладки Microsoft Visual Studio
Before sort:
Maximal element = 24999 has index = 8382
Minimal element = 1 has index = 8497
After sort:
Maximal element = 24999 has index = 9999
Minimal element = 1 has index = 0

```

Рисунок 2. – Скриншот результата выполнения задания 2

Анализ полученных результатов:

Полученный результат верен, так как: индекс минимального элемента после сортировки должен равен **0**, индекс максимального элемента (***mass_size* - 1**).

Задание 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*.

Изменённая пользовательская функция *ParallelSort(...)*:

```

duration <double> ParallelSort(int *begin, int *end)
{
    high_resolution_clock::time_point t1 = high_resolution_clock::now();
    if (begin != end)
    {
        --end;
        int *middle = std::partition(begin, end, std::bind2nd(std::less<int>(),
*end));
        std::swap(*end, *middle);
        cilk_spawn ParallelSort(begin, middle);
        ParallelSort(++middle, ++end);
    }
    return high_resolution_clock::now() - t1;
}

```

```

        cilk_sync;
    }
    high_resolution_clock::time_point t2 = high_resolution_clock::now();
    duration<double> duration = (t2 - t1);
    return duration;
}

```

Изменённый код программы для изменения размера массива и с изменённой функцией *ParallelSort(...)*:

```

#include <cilk/cilk.h>
#include <cilk/cilk_api.h>
#include <cilk/reducer_max.h>
#include <cilk/reducer_min.h>
#include <cilk/reducer_vector.h>
#include <chrono>
#include <iostream>

using namespace std::chrono;

/// Функция ReducerMaxTest() определяет максимальный элемент массива,
/// переданного ей в качестве аргумента, и его позицию
/// mass_pointer - указатель исходный массив целых чисел
/// size - количество элементов в массиве
void ReducerMaxTest(int *mass_pointer, const long size)
{
    cilk::reducer<cilk::op_max_index<long, int>> maximum;
    cilk_for(long i = 0; i < size; ++i)
    {
        maximum->calc_max(i, mass_pointer[i]);
    }
    printf("Maximal element = %d has index = %d\n",
        maximum->get_reference(), maximum->get_index_reference());
}

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",
        minimum->get_reference(), minimum->get_index_reference());
}

/// Функция ParallelSort() сортирует массив в порядке возрастания
/// begin - указатель на первый элемент исходного массива
/// end - указатель на последний элемент исходного массива
duration<double> ParallelSort(int *begin, int *end)
{
    high_resolution_clock::time_point t1 = high_resolution_clock::now();
    if (begin != end)
    {
        --end;
        int *middle = std::partition(begin, end, std::bind2nd(std::less<int>(),
*end));
        std::swap(*end, *middle);
        cilk_spawn ParallelSort(begin, middle);
        ParallelSort(++middle, ++end);
        cilk_sync;
    }
    high_resolution_clock::time_point t2 = high_resolution_clock::now();
    duration<double> duration = (t2 - t1);
}

```

```

        return duration;
    }

int main()
{
    srand((unsigned)time(0));

    // устанавливаем количество работающих потоков = 4
    __cilkrts_set_param("nworkers", "4");

    long i;
    const long mass_size = 10000;

    long sizes[] = { 10, 50, 100, 500, 1000, 10000, 100000, 1000000 };
    int number_size = sizeof(sizes) / sizeof(long);

    for (int jj = 0; jj < number_size; jj++)
    {
        std::cout << "-----Size of massive: " << sizes[jj] << std::endl;
        int *mass_begin, *mass_end;
        int *mass = new int[sizes[jj]];

        duration <double> duration;

        for (i = 0; i < sizes[jj]; ++i)
        {
            mass[i] = (rand() % 25000) + 1;
        }

        mass_begin = mass;
        mass_end = mass_begin + sizes[jj];

        //printf("Before sort:\n");
        ReducerMaxTest(mass, sizes[jj]);
        ReducerMinTest(mass, sizes[jj]);

        //printf("After sort:\n");

        duration = ParallelSort(mass_begin, mass_end);
        std::cout << "-----Time sort: " << duration.count() << " seconds" <<
std::endl;
        ReducerMaxTest(mass, sizes[jj]);
        ReducerMinTest(mass, sizes[jj]);

        delete[]mass;
    }

    return 0;
}

```

Результат:

```

-----Size of massive: 10
Maximal element = 22740 has index = 4
Minimal element = 3596 has index = 8
-----Time sort: 4.2696e-05 seconds
Maximal element = 22740 has index = 9
Minimal element = 3596 has index = 0
-----Size of massive: 50
Maximal element = 24751 has index = 27
Minimal element = 163 has index = 39
-----Time sort: 5.0086e-05 seconds
Maximal element = 24751 has index = 49
Minimal element = 163 has index = 0
-----Size of massive: 100
Maximal element = 24555 has index = 14
Minimal element = 36 has index = 34
-----Time sort: 6.856e-05 seconds
Maximal element = 24555 has index = 99
Minimal element = 36 has index = 0
-----Size of massive: 500
Maximal element = 24988 has index = 298
Minimal element = 30 has index = 141
-----Time sort: 0.000301745 seconds
Maximal element = 24988 has index = 499
Minimal element = 30 has index = 0
-----Size of massive: 1000
Maximal element = 24980 has index = 844
Minimal element = 13 has index = 299
-----Time sort: 0.000732396 seconds
Maximal element = 24980 has index = 999
Minimal element = 13 has index = 0
-----Size of massive: 10000
Maximal element = 25000 has index = 52
Minimal element = 2 has index = 4530
-----Time sort: 0.00799314 seconds
Maximal element = 25000 has index = 9999
Minimal element = 2 has index = 0
-----Size of massive: 100000
Maximal element = 25000 has index = 3188
Minimal element = 1 has index = 15772
-----Time sort: 0.0587264 seconds
Maximal element = 25000 has index = 99994
Minimal element = 1 has index = 0
-----Size of massive: 500000
Maximal element = 25000 has index = 31095
Minimal element = 1 has index = 3465
-----Time sort: 0.290446 seconds
Maximal element = 25000 has index = 499981
Minimal element = 1 has index = 0
-----Size of massive: 1000000
Maximal element = 25000 has index = 38469
Minimal element = 1 has index = 35323
-----Time sort: 0.461046 seconds
Maximal element = 25000 has index = 999968
Minimal element = 1 has index = 0

```

Рисунок 3. – Скриншот результата выполнения задания 3

Для удобства, результаты времени выполнения при различных размерах массива в таблице:

| Размер | Время |
|---------|--------------------|
| 10 | 0.00004269 seconds |
| 50 | 0.00005008 seconds |
| 100 | 0.00006856 seconds |
| 500 | 0.00030174 seconds |
| 1000 | 0.00073239 seconds |
| 10000 | 0.00799314 seconds |
| 100000 | 0.0587284 seconds |
| 500000 | 0.290446 seconds |
| 1000000 | 0.461046 seconds |

Таблица 1. – Результаты выполнения задания 3

Анализ полученных результатов:

При увеличении количества элементов в массиве увеличивается время сортировки.

*После выполнения понял, что перепутал с 4 заданием, нужно было в 10, 50, 100 раз. Добавил в 50 раз и оставил остальные результаты.

Задание 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.

Пример объявления *reducer* вектора: *cilk::reducer<cilk::op_vector<int>>red_vec;*

Пример его заполнения: *red_vec->push_back(rand() % 20000 + 1);*

Параметр функции *sz* - количество элементов в каждом из векторов.

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.

Пользовательская функция *CompareForAndCilk_For(size_t sz):*

```
void CompareForAndCilk_For(size_t sz)
{
    std::vector<int> my_vect;
    cilk::reducer<cilk::op_vector<int>>red_vec;
    high_resolution_clock::time_point t1 = high_resolution_clock::now();
    for (int i = 0; i < sz; i++)
        my_vect.push_back(rand() % 20000 + 1);
    high_resolution_clock::time_point t2 = high_resolution_clock::now();
    cilk_for(int i = 0; i < sz; i++)
        red_vec->push_back(rand() % 20000 + 1);
    high_resolution_clock::time_point t3 = high_resolution_clock::now();
    duration<double> duration_for = (t2 - t1);
    duration<double> duration_cilk = (t3 - t2);

    std::cout << std::endl;
    std::cout << "-----Size of massive: " << sz << std::endl;
    std::cout << "Time for 'for': " << duration_for.count() << " seconds" <<
std::endl;
    std::cout << "Time for 'cilk_for': " << duration_cilk.count() << " seconds" <<
std::endl;
}
```

Код всей программы:

```
#include <cilk/cilk.h>
#include <cilk/cilk_api.h>
#include <cilk/reducer_max.h>
#include <cilk/reducer_min.h>
#include <cilk/reducer_vector.h>
#include <chrono>
#include <iostream>
#include <stdlib.h>
#include <vector>

using namespace std::chrono;

void CompareForAndCilk_For(size_t sz)
{
    std::vector<int> my_vect;
    cilk::reducer<cilk::op_vector<int>>red_vec;
    high_resolution_clock::time_point t1 = high_resolution_clock::now();
    for (int i = 0; i < sz; i++)
        my_vect.push_back(rand() % 20000 + 1);
    high_resolution_clock::time_point t2 = high_resolution_clock::now();
    cilk_for(int i = 0; i < sz; i++)
```

```

        red_vec->push_back(rand() % 20000 + 1);
        high_resolution_clock::time_point t3 = high_resolution_clock::now();
        duration<double> duration_for = (t2 - t1);
        duration<double> duration_cilk = (t3 - t2);

        std::cout << std::endl;
        std::cout << "-----Size of massive: " << sz << std::endl;
        std::cout << "Time for 'for': " << duration_for.count() << " seconds" <<
std::endl;
        std::cout << "Time for 'cilk_for': " << duration_cilk.count() << " seconds" <<
std::endl;
    }

int main()
{
    srand((unsigned)time(0));

    // устанавливаем количество работающих потоков = 4
    __cilkrts_set_param("nworkers", "4");

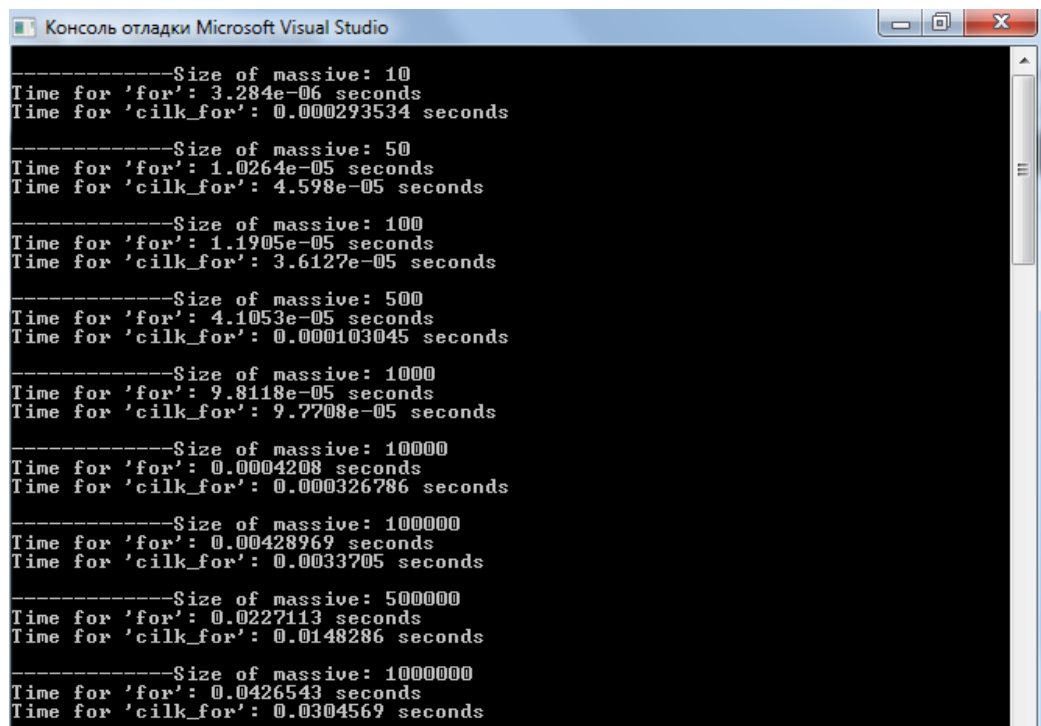
    long i;
    const long mass_size = 10000;

    long sizes[] = { 10, 50, 100, 500, 1000, 10000, 100000, 500000, 1000000 };
    int number_size = sizeof(sizes) / sizeof(long);

    for (int jj = 0; jj < number_size; jj++)
    {
        CompareForAndCilk_For(sizes[jj]);
    }
    return 0;
}

```

Результат:



```

Консоль отладки Microsoft Visual Studio

-----Size of massive: 10
Time for 'for': 3.284e-06 seconds
Time for 'cilk_for': 0.000293534 seconds

-----Size of massive: 50
Time for 'for': 1.0264e-05 seconds
Time for 'cilk_for': 4.598e-05 seconds

-----Size of massive: 100
Time for 'for': 1.1905e-05 seconds
Time for 'cilk_for': 3.6127e-05 seconds

-----Size of massive: 500
Time for 'for': 4.1053e-05 seconds
Time for 'cilk_for': 0.000103045 seconds

-----Size of massive: 1000
Time for 'for': 9.8118e-05 seconds
Time for 'cilk_for': 9.7708e-05 seconds

-----Size of massive: 10000
Time for 'for': 0.0004208 seconds
Time for 'cilk_for': 0.000326786 seconds

-----Size of massive: 100000
Time for 'for': 0.00428969 seconds
Time for 'cilk_for': 0.0033705 seconds

-----Size of massive: 500000
Time for 'for': 0.0227113 seconds
Time for 'cilk_for': 0.0148286 seconds

-----Size of massive: 1000000
Time for 'for': 0.0426543 seconds
Time for 'cilk_for': 0.0304569 seconds

```

Рисунок 4. – Скриншот результата выполнения задания 4

Для удобства, результаты времени выполнения при различных размерах данных в таблице:

| Размер | Тип цикла | Время |
|---------|-----------|---------------------|
| 10 | for | 0.000003284 seconds |
| | cilk_for | 0.000293534 |
| 50 | for | 0.000010264 seconds |
| | cilk_for | 0.00004598 seconds |
| 100 | for | 0.000011905 seconds |
| | cilk_for | 0.000036127 seconds |
| 500 | for | 0.000041053 seconds |
| | cilk_for | 0.000103045 seconds |
| 1000 | for | 0.000098118 seconds |
| | cilk_for | 0.000097708 seconds |
| 10000 | for | 0.0004208 seconds |
| | cilk_for | 0.0003267 seconds |
| 100000 | for | 0.004208 seconds |
| | cilk_for | 0.003371 seconds |
| 500000 | for | 0.022711 seconds |
| | cilk_for | 0.014828 seconds |
| 1000000 | for | 0.042654 seconds |
| | cilk_for | 0.030457 seconds |

Таблица 2. – Результаты выполнения задания 3

Анализ полученных результатов:

- 1) Вплоть до количества элементов данных в 500 элементов цикл for выполнялся быстрее, чем cilk_for.
- 2) С количества элементов данных в 1000 cilk_for начал обгонять for. При этом, чем больше элементов данных – тем быстрее срабатывал cilk_for, по сравнению с for.

Задание 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***?

Ответы на данные вопросы представьте в **pdf** документе, который необходимо загрузить в систему в качестве отчета к этому заданию.

*Ответ на данные вопросы продублирован в обоих отчётах – на Github и отправленном отдельно

1) Почему при небольших значениях *sz* цикл cilk_for уступает циклу for в быстродействии?

Потому что при использовании `cilk_for`, работающего параллельно, затрачивается дополнительно время на создание поток, распределения задач между ними, а также на переключение контекста.

2) В каких случаях целесообразно использовать цикл `cilk_for`?

`Cilk_for` целесообразнее использовать при работе с большими массивами данных, если работу над ними можно распараллелить. Лучшие результаты по сравнению с `for` были показаны `cilk_for` при количестве элементов в массиве 10.000 и больше.

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

Определения:

`cilk_for` – конструкция, предназначенная для распараллеливания циклов с известным количеством повторений. В процессе компиляции тело цикла конвертируется в функцию, которая вызывается рекурсивно. Планировщик автоматически распределяет поддеревья рекурсии между обработчиками.

`cilk_spawn` – конструкция, которая может быть использована непосредственно перед вызовом функции, чтобы указать системе, что данная функция может выполняться параллельно с вызывающей.

`cilk_sync` – точка синхронизации функций. Используется, когда дальнейшие вычисления в родительской функции невозможны без результатов дочерней.

Иначе говоря: `cilk_spawn` определяет возможность асинхронного запуска; `cilk_sync` – предписывает завершение всех инструкций, заключённых между `cilk_spawn` и `cilk_sync` перед продолжением выполнения программы.

Ответ:

В случае использования `cilk_spawn` и `cilk_sync`, 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. Таким образом, `cilk_spawn` нужно делать не на каждой итерации, а скажем, только один раз, а все остальные итерации пусть воспринимаются как продолжение работы программы.

Тогда как `cilk_for` распараллеливает цикл `for`. В процессе компиляции тело цикла конвертируется в функцию, которая вызывается рекурсивно. Планировщик автоматически распределяет поддеревья рекурсии между обработчиками.

Выводы:

1) В рамках задания к занятию 2 были изучены такие функции, как:

1) `reducer<op_max_index<long, int>>`

2) `reducer<op_min_index<long, int>>`

3) `reducer< op_vector<int>>`

- 2) Были рассмотрены зависимости скорости выполнения циклов `for` и `cilk_for` в зависимости от количества элементов данных.
- 3) Была теоретически рассмотрена возможность использования `cilk_spawn` и `cilk_sync` и недостатки по сравнению с использованием `cilk_for`.