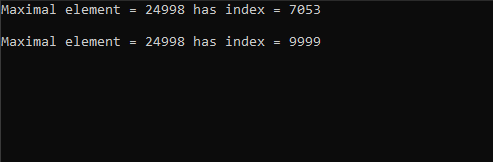
1. **Разберите пример программы нахождения максимального элемента массива и его индекса task\_for\_lecture2.cpp. Запустите программу и убедитесь в корректности ее работы.**

**На рисунке представлен результат работы программы task\_for\_lecture2.cpp:**



В первой строчке показан индекс максимального элемента до сортировки, во второй после.

1. **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).**

**Ниже приведен листинг функции 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]);

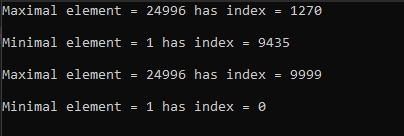
}

printf("Minimal element = %d has index = %d\n\n",

minimum->get\_reference(), minimum->get\_index\_reference());

}

На рисунке приведен результат выполнения программы, после добавления функции ReducerMinTest(…) :



1. **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.**

**Ниже приведен листинг измененной функции 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);

cilk\_sync;

}

high\_resolution\_clock::time\_point t2 = high\_resolution\_clock::now();

duration<double> duration = (t2 - t1);

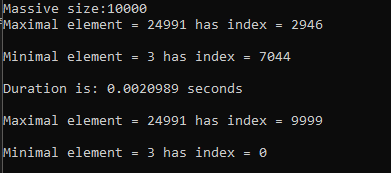
return duration;

}

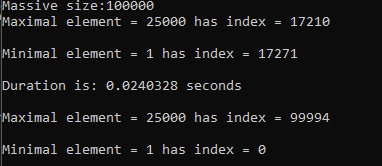
В таблице приведены данные по времени выполнения для различных значений размера массива:

|  |  |
| --- | --- |
| **Размер** | **Время** |
| 10000 | 0.0014905 seconds |
| 10000\*10 = 100000 | 0.0317465 seconds |
| 10000\*50 = 500000 | 0.0759892 seconds |
| 10000\*100 = 1000000 | 0.159344 seconds |

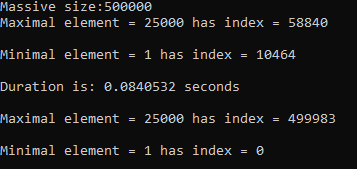
10000:



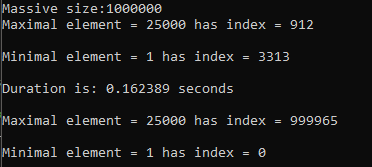
100000:



500000:



1000000:



1. **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.**

Ниже приведен листинг функции CompareForAndCilk\_For(size\_t sz) и результат ее выполнения:

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);

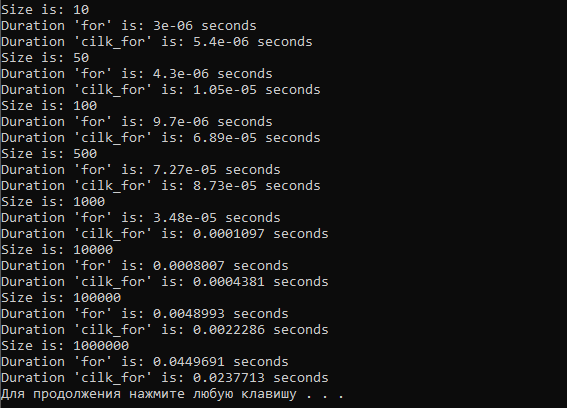
duration<double> duration\_cilk = (t3 - t2);

std::cout << "Size is: " << sz << std::endl;

std::cout << "Duration 'for' is: " << duration\_for.count() << " seconds" << std::endl;

std::cout << "Duration 'cilk\_for' is: " << duration\_cilk.count() << " seconds" << std::endl;

}



Из результатов видно, что для заполнения вектора, размер которого меньше 10000, следует применять for, так как он занимает меньше времени. Однако, начиная с 10000, целесообразнее использовать функцию cilk\_for.

1. **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?**

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

Потому что при использование параллельного cilk\_for затрачивается дополнительно время на создание поток, распределения задач между ними, а также на переключение контекста.

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

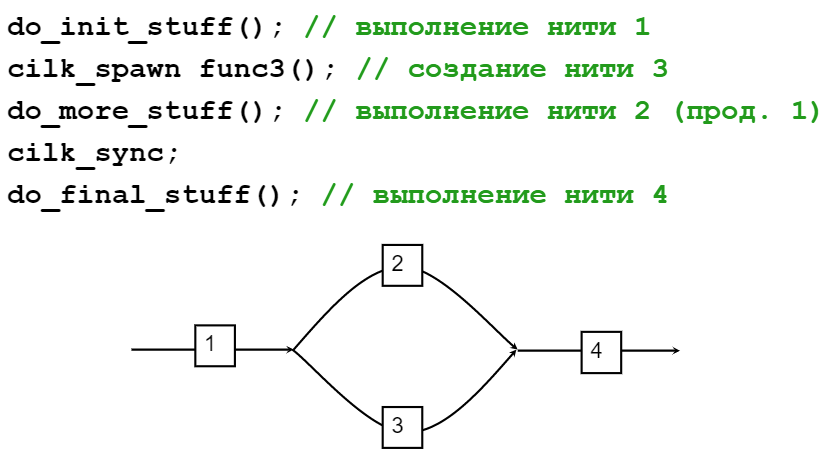
Cilk\_for целесообразно использовать при работе с большими массивами данных (порядка 10000 и более), если работу над ними можно распараллелить.

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

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.

На рисунке ниже приведен пример использования cilk\_spawn в паре с cilk\_sync.



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.

Ниже приведена диаграмма нитей при использовании cilk\_for.

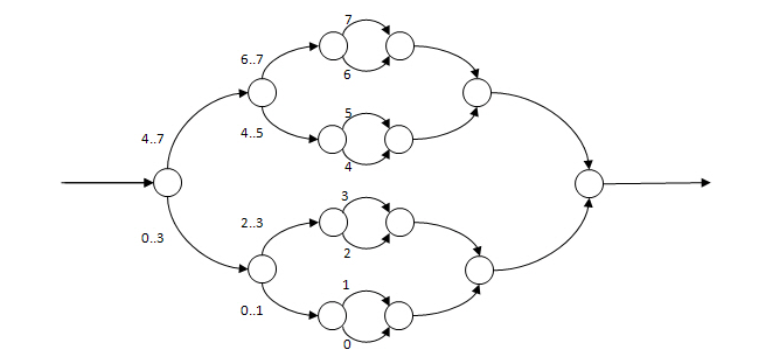


Диаграмма при многократном вызове cilk\_spawn:

