MINISTRY FOR EDUCATION AND SCIENCE OF RUSSIA

SAINT PETERSBURG ELECTROTECHNICAL UNIVERSITY «LETI»

Laboratory work № 1
Computational Systems

Student
Group 4300
Bathaie N.

Teacher
PhD, associate prof.
of CE department
Paznikov A.A.

Saint Petersburg
ETU «LETI»
2018

MULTI-THREAD IMPLEMENTATION OF THE SHELL-SORT ALGORITHM.

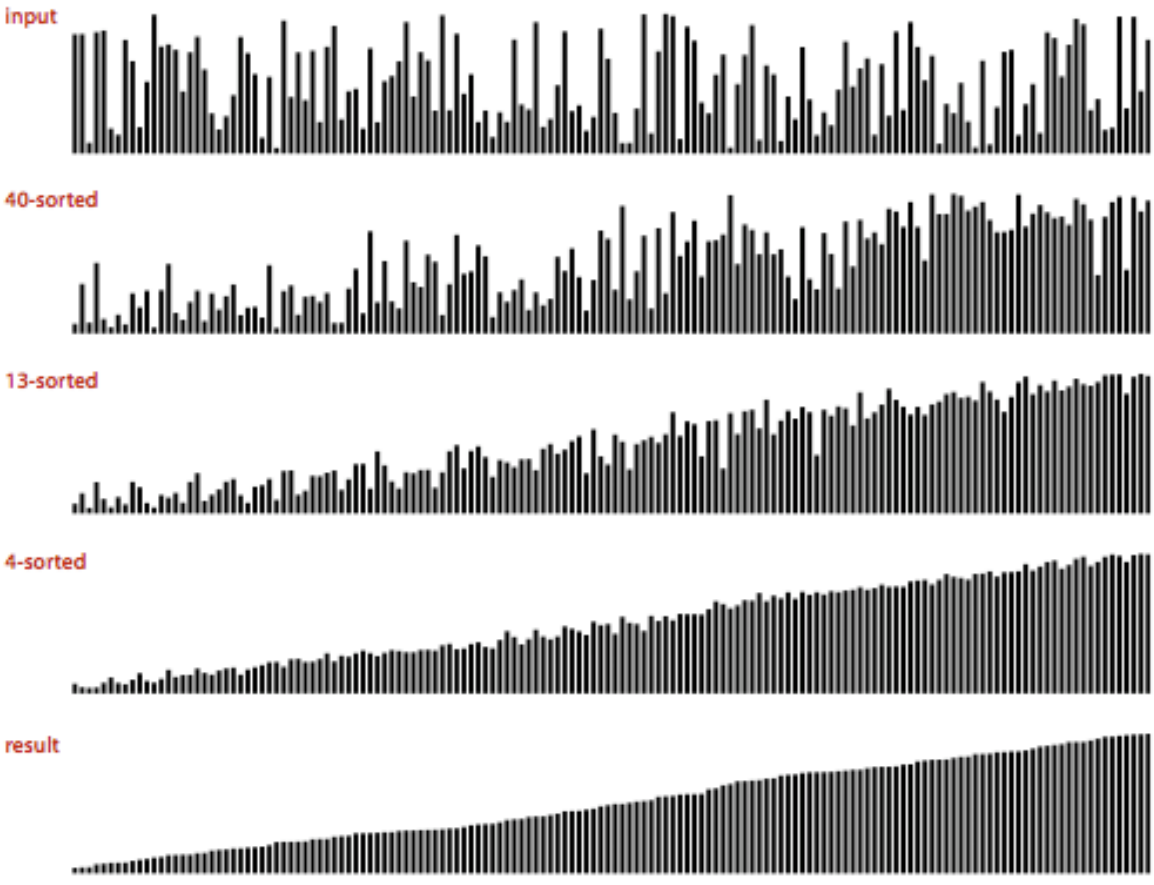
1. Shell-sort algorithm

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval. This interval is calculated based on Knuth's formula:

$$h = h \times 3 + 1 \quad \textit{Where } h \textit{ is interval with initial value } 1$$

This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is O(n), where n is the number of items. And the worst case space complexity is O(n).



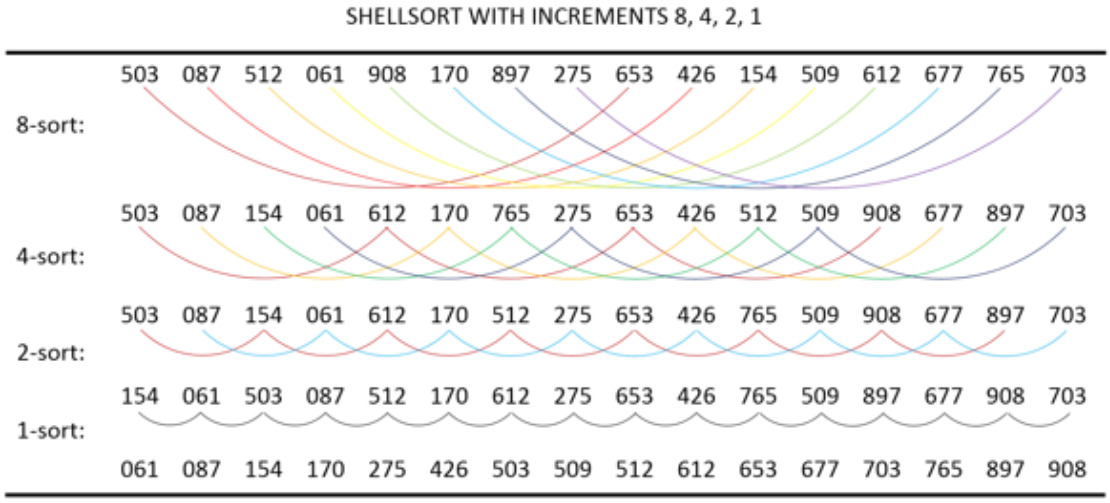
Visual trace of shellsort

2. Shell-sort function:

```
int shellSort(int arr[], int n)
{
	// Start with a big gap, then reduce the gap
	for (int gap = n/2; gap > 0; gap /= 2)
	{
		// Do a gapped insertion sort for this gap size.
		for (int i = gap; i < n; i += 1)
		{
			// add a[i] to the gap-sorted elements.
			// save a[i] in temp and make a hole at position i
			int temp = arr[i];

			// shift earlier gap-sorted elements up until the
			// correct location for a[i] is found
			int j;
			for (j = i; j >= gap && arr[j - gap] > temp;
								j -= gap)
				arr[j] = arr[j - gap];

				// put temp (the original a[i]) in its
				// correct location
			arr[j] = temp;
		}
	}
	return 0;
}
```

3. Using multi-thread for shell-sort

If we survey the code of the shell-sort algorithm we can comprehend that this algorithm actually performs some insertion sorts with a gap which is halved after each run.



For multi-thread implementation of the example above we can use 8 thread to implement the insertion-sorts with the gap 8. Similarly, we can use 4 thread for the next step with the gape 4 and so on. The important point here is that the data passed to each thread has no intersection with the others; so that, no lock is needed.

4. Code

```
#include <iostream>
#include <random>
#include <algorithm>
#include <thread>
#include <mutex>
#include <chrono>
#include <fstream>
#define MIN(a,b) ((a) < (b) ? a : b)

using arr_type = double;

const auto nelem = 960000;
const auto maxrand = 10000000;
const auto nruns = 10;

static auto rand_gen()
{
    static std::random_device rnd_dev;
    static std::mt19937 mersenne_engine{rnd_dev()};
    static std::uniform_int_distribution<int> dist{1, maxrand};
    return dist(mersenne_engine);
}

void InsertionSort(arr_type a[])
{
    int j, k, temp;
    for(j = 1; j < nelem; j++)
    {
        for(k = j-1; k >= 0; k--)
        {
            // If value at higher index is greater, then break the
            // loop.
            if(a[(k+1)] >= a[k])
                break;
            // Switch the values otherwise.
            else
            {
                temp = a[k];
                a[k] = a[(k+1)];
                a[(k+1)] = temp;
            }
        }
    }
}
```

```
void InsertionSort(arr_type a[], int n, int gap)
{
    int j, k, temp;
    for(j = 1; j < n; j++)
    {
        for(k = j-1; k >= 0; k--)
        {
            // If value at higher index is greater, break the loop.
            if(a[(k+1)*gap] >= a[k*gap])
                break;
            // Switch the values otherwise.
            else
            {
                temp = a[k*gap];
                a[k*gap] = a[(k+1)*gap];
                a[(k+1)*gap] = temp;
            }
        }
    }
}

void shell_sort_test(arr_type a[])
{
    int i, j, k, temp;
    // Gap 'i' between the elements to be compared, initially n/2.
    for(i = (int)(nelem/2); i > 0; i = (int)(i/2))
    {
        for(j = i; j < nelem; j++)
        {
            for(k = j-i; k >= 0; k = k-i)
            {
                // If value at higher index is greater, then break
                // the loop.
                if(a[k+i] >= a[k])
                    break;
                // Switch the values otherwise.
                else
                {
                    temp = a[k];
                    a[k] = a[k+i];
                    a[k+i] = temp;
                }
            }
        }
    }
}

static void shell_sort(int thr_id,int gap,int max_threads, arr_type
arr[])
{
      // calculate the length of the array for insertion sort with
      // gap
    auto length =nelem-thr_id+gap-1;
    InsertionSort(&arr[thr_id],(int)(length/gap),gap);
```

```
      // calculate how many insertion sort each thread should do
      // it depends on how many element we have in each gape
      // if it's less than the number of our threads we just use
      // as many threads as we need.
      // if there are more, each thread has to perform
      // gap/max threads
      // insertion sort (some one more)
    auto g = gap - max_threads - thr_id;
    auto k = 1;
    while(g > 0)
    {
        InsertionSort(&(arr[thr_id + max_threads*k]),
                (int)((length - max_threads * k)/gap),gap);
        g -= max_threads;
        g++;
    }
}

void parallel_shellsort(arr_type arr[],auto max_threads)
{
    // create array of threads
    std::thread threads[max_threads];
    for(auto gap = (unsigned int)(nelem/2); gap > 1; gap = gap/2)
    {
        // to decide howmany threads is needed for this gap
        auto t_num = MIN(gap,max_threads);
        for (auto thr_id = 0u ; thr_id < t_num ; thr_id++ )
        {
            threads[thr_id] = std::thread{shell_sort,
                        thr_id, gap, max_threads, arr};
        }
        for (auto t = 0u; t < t_num; t++)
            threads[t].join();
    }
    // when gap equals to 1 whe have one thread and we have to do
    // one insertion sort, so i put it here.
    InsertionSort(arr);
}
int main()
{
    arr_type arr[nelem];

    // define max_tthread based one the number of cores.
    const auto max_threads = std::thread::hardware_concurrency();

    auto get_time = std::chrono::steady_clock::now;
    decltype(get_time()) start, end;

    /// Serial
    start = get_time();
    for (auto i = 0; i < nruns; i++)
    {
        std::generate(arr, arr+nelem-1, rand_gen);
        shell_sort_test(arr);
    }
    end = get_time();
```

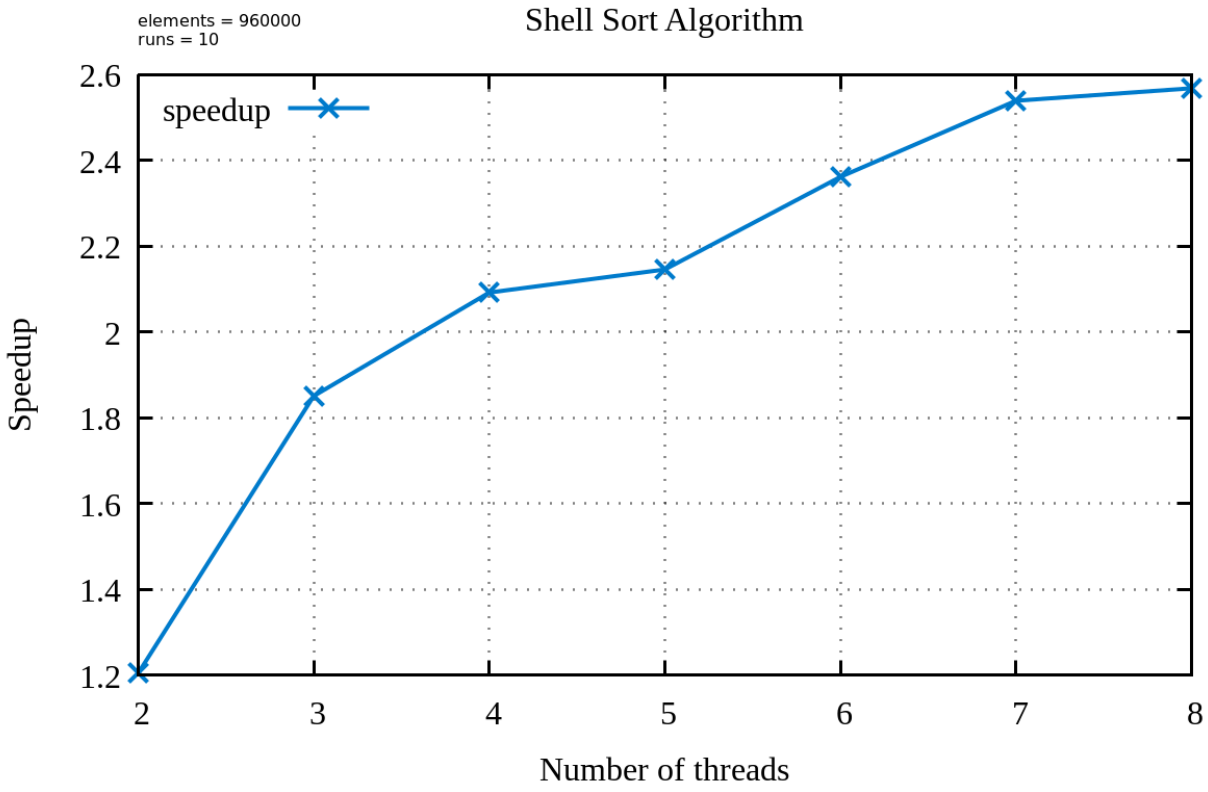
```
    auto elapsed = std::chrono::duration_cast
                    <std::chrono::milliseconds>(end - start).count();
    auto ser_time = double(elapsed) / nruns;

    std::cout << "Serial time: " << ser_time << "ms" << std::endl;

    /// Parallel
    std::cout << "Paralel time: " << std::endl;
    std::ofstream speedupfile{"speedup"};
    if (!speedupfile.is_open())
    {
        std::cerr << "can't open file" << std::endl;
        return 1;
    }

    for (auto nthr = 2u; nthr <= max_threads; nthr++)
    {
        start = get_time();

        for (auto i = 0; i < nruns; i++)
        {
            std::generate(arr, arr+nelem-1, rand_gen);
            parallel_shellsort(arr, nthr);
        }
        end = get_time();
        elapsed = std::chrono::duration_cast
                  <std::chrono::milliseconds>(end - start).count();
        auto par_time = double(elapsed) / nruns;
        std::cout << "Using " << nthr << " threads: "  << "elapsed
time "
                  << par_time << "ms _ speedup " << ser_time /
par_time << std::endl;
        speedupfile << nthr << "\t" << ser_time / par_time <<
std::endl;
    }
    speedupfile.close();
    return 0;
}
```

5. Conclusion

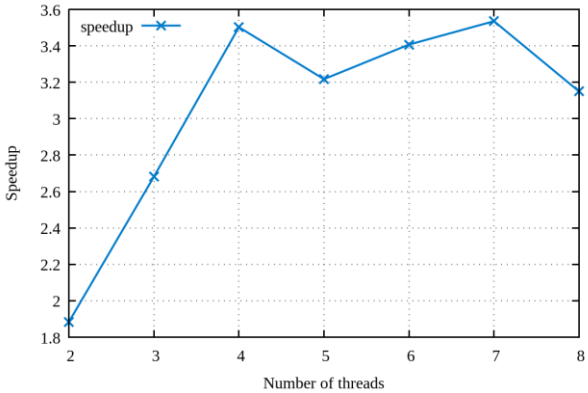
The result of this code shows that we can increase the speed of shell-sort algorithm by multi-thread implementation.

Also it could be comprehend that the amount of the speed up is highly depended on the system we run the code on. The output of the same code can be vary a lot running it in two different computers.

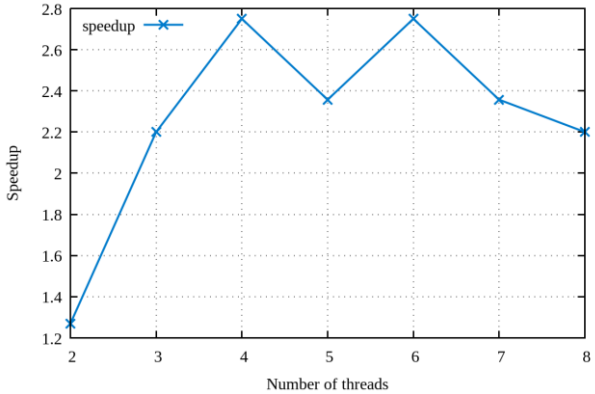
The output graph for this experience was as below.



To have a better estimation of how much this code is efficient the code presented in the class was ran on the same system.



speed-up for vecmult function for 9,600,000 elements



speed-up for vecmult function for 960,000 elements

By comparing the results of the two functions, we can see that the speed-up graph for the same amount of elements is almost the same from which it can be concluded that although the speed-up is not that close to what Amdahl's low predicts, the code is working well and the output speed-up is acceptable.