Sample Sort with TBB

Damir Ferizovic, David Vogelbacher

Karlsruhe Institute of Technology, Karlsruhe, Germany
{damir.ferizovic, david.vogelbacher}@kit.edu

Abstract. We benchmark our implementation of the parallel sample sort algorithm using the Intel TBB framework [1]. We also compare it with the STL sort algorithm.

1 Algorithm - Sample Sort

Let p be the number of available threads. At the beginning of the sample sort algorithm, the input data is divided into p groups (p being the number of available threads) by using multiple pivot elements (similar to Quicksort). Then, each thread sorts one of the groups locally. The end result is just the concatenation of the local results. In order to split the input, $p-1$ pivot elements are selected and the elements of the input are grouped into p groups according to these pivot elements. Then, each thread sorts one of the groups. In order to get good pivot elements, oversampling is used: Initially, more than $S \cdot p$ elements are selected randomly as possible pivot elements for a constant S. These elements are then sorted (sequentially because the number of these elements is still small). After that, every Sth element is selected as pivot element. This oversampling helps finding good pivot elements, such that the resulting groups are are of similar size with high probability.

2 Inplementation Details

2.1 Pivot selection

We randomly select $S \cdot p$ elements as possible pivot elements, where $S = 4 \cdot \frac{\log m}{0.1^2}$ (m is the size of the input). We chose the value S according to the lecture [2]. The value of 0.1 for the parameter ε was found to be quite good but other, similar values work as well.
We then sort the possible pivot elements using the sequential STL sorting algorithm (for large inputs with a large number of possible pivot elements one could recursively use sample sort). Lastly, we select every Sth element as pivot element.

[1] https://www.threadingbuildingblocks.org/
[2] http://algo2.iti.kit.edu/sanders/courses/paralg15/

2.2 Grouping of elements

First, we calculate the group of each element by executing a *parallel-for-loop* over the number of elements. Each iteration of the for-loop calculates the group to which the element belongs. We also store the element in a two-dimensional ThreadLocal array, where the first dimension is for each group and the second for the different elements of the group.
Second, using a *parallel-for-loop* over the number of threads, each thread calculates the number of elements that belong to its group by looping over all arrays that were previously stored thread locally and summing the amount of elements that belong to its group.
Lastly, each thread moves the elements belonging to its group into the right place in the original array. Elements of group i are stored beginning at index $\sum_{j=0}^{i-1} \text{groupsize}[j]$. Thus, after this step the original array is reordered in a way that elements belonging to the same group are stored next to each other.

2.3 Local sorting

As a last step, we sort all groups locally using the STL sorting algorithm and are finished.

3 Experimental Results

3.1 Hardware

– OS: Linux (3.18.27-1-MANJARO)
– Processor: Intel(R) Core(TM) i7-3740QM CPU @ 2.70GHz with 4 cores and 8 threads
– RAM: 16GB
– Compiler: g++ (GCC) 5.3.0

3.2 Experiments

We benchmarked our implementation of sample sort and the STL sort using combinations of the following parameters:

– Type of elements to be sorted: 32-bit Integer or 32-bit float
– Generator for elements: Either random or all zero
– Number of threads: 1,2,4,8
– Number of elements: From 10^5 to 10^8 in magnitudes of 10

3.3 Plots

Running time **??** and speedup plots 2 (for each generator, 64-bit integer and 32-bit floating point (not for non-comparative integer sorting algorithms).
Interpretation.

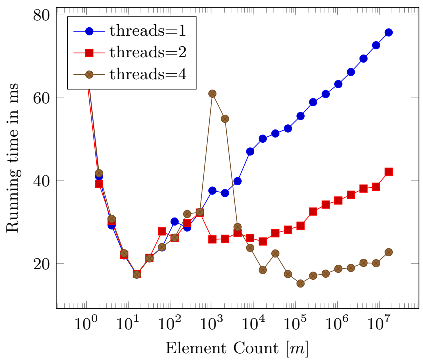

Fig. 1. Running times of `std::sort` and sample sort for random input of type Integer. Mean of 10 iterations.

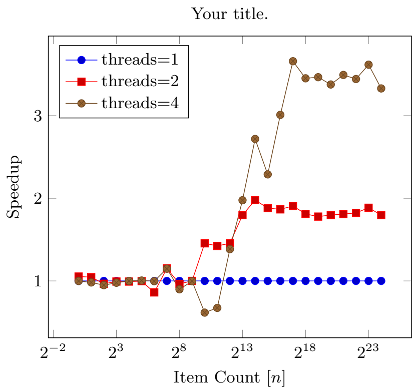

Fig. 2. Speedup of `std::sort` with uniform input. Mean of 10 iterations.

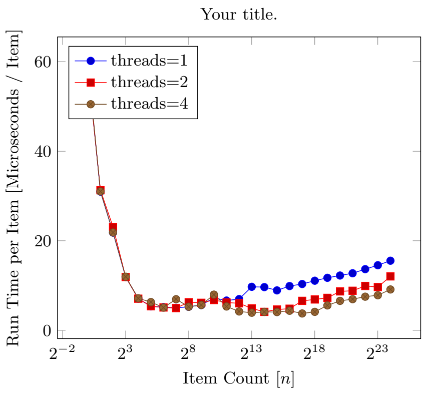

Fig. 3. Running times of `std::sort` with zero input. Mean of 10 iterations.

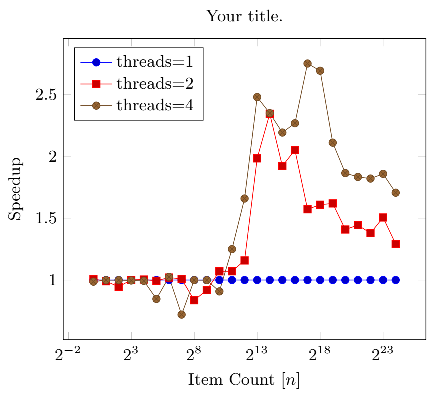

Fig. 4. Speedup of `std::sort` with zero input. Mean of 10 iterations.