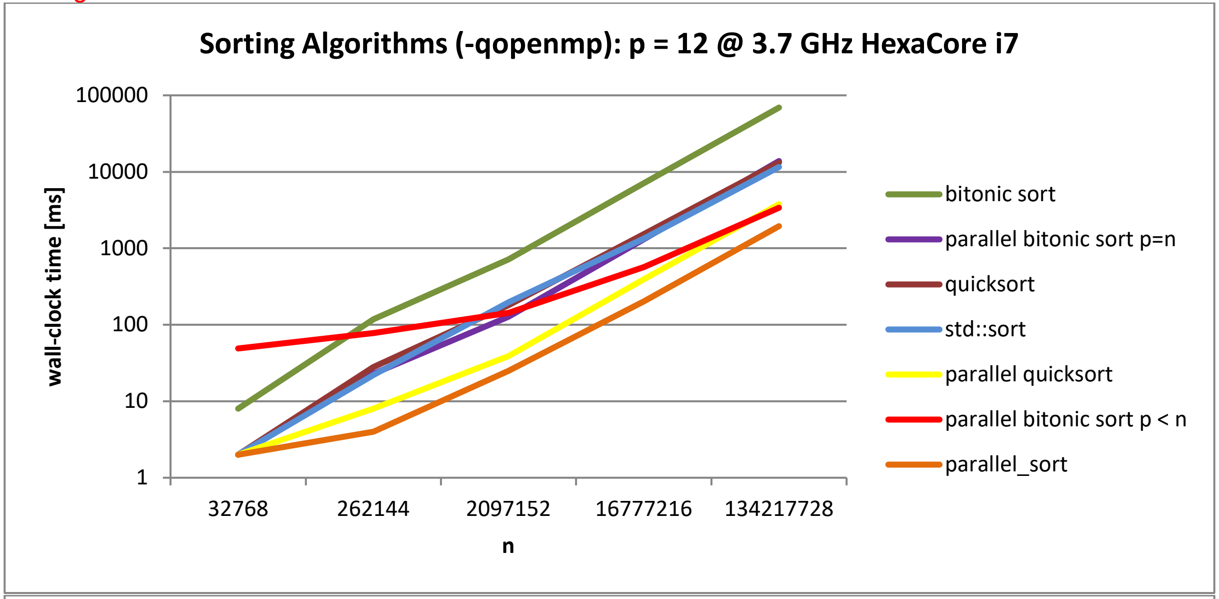
Prof. Dr. C. Stamm christoph.stamm@fhnw.ch

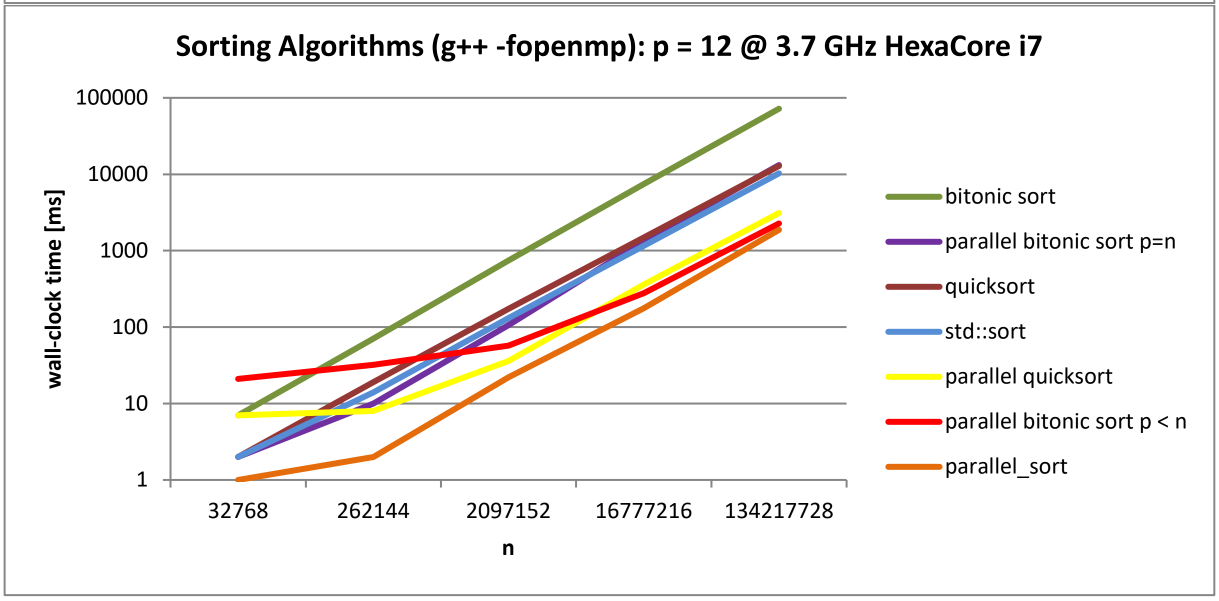
Solution 5

1 Bitonic Sort

a) The outermost for loop iterates over the $\log p \oplus \mathrm{BM}[n]$ networks connected in series and thus is not suitable for parallelization. The second for loop iterates over the sequential stages of a $\mathrm{BM}[n]$ network and again is not suitable for parallelization. Only the innermost for-loop iterates over all parallel data lines (= values) and is thus best suited for parallelization.
The OpenMP statement `#pragma omp parallel` creates an ensemble of threads, while the statement `#pragma omp for` uses the available threads for parallel execution of the immediately following for loop. These two statements are often combined into one: `#pragma omp parallel for`.
To avoid constantly creating new thread ensembles in a parallelized inner for-loop, the thread ensemble can be created ready for entry into the outermost for-loop and then parallelized with the separate statement `#pragma omp for` the innermost for-loop.
This naïve parallelization of the Bitonic Sort is slower than the sequential standard sorting algorithm.

b) Partitioning significantly increases the performance of the parallel Bitonic Sort and the speedup over the standard sequential sort algorithm is about 3 to 5. The chart below shows that the optimized parallel Bitonic Sort (red curve) performs for large array sizes similarly to the parallelized quicksort algorithm.





2 Parallel Quicksort

Test your implementation for arrays of different sizes $n = 2^{3k}$, $k \in [5, 9]$, and compare the performance with other sorting algorithms. Plot the performances and analyze and discuss the plotted curves.

a) The C++ standard sorting algorithm is the fastest of the tested serial sorting algorithms, but the sequential quicksort implementation has almost the same speed. The naïve parallelization of the bitonic sort behaves very similar. Only the sequential bitonic sort has an extremely poor performance.
The performance of the three other parallel sorting algorithms is quite different. The parallel standard sorting algorithm performs clearly better than the parallel quicksort implementation and the parallel bitonic sort for $p < n$. However, it's interesting to see that the choice of the C++ compiler influences the OpenMP performance, too. The second chart shows the performance of the g++ compiler on the same machine. All parallelized algorithms but parallel bitonic sort for $p = n$ have an improved performance.
The time complexity of the implemented parallel quicksort is $T_P = \Theta\left(\frac{n}{p}\log\frac{n}{p}\right) + \Theta\left(\frac{n}{p}\log p\right) + \Theta(p\log p)$, because of a simpler but linear algorithm to compute the prefix sums.

b) Assuming that the prefix sum is computed in Θ(log p) time, the parallel costs are:
$$\Theta(n\log n + p\log^2 p)$$
To ensure that these parallel costs do not exceed the optimal cost, the following must apply:
$$\Theta(n\log n + p\log^2 p) = O(n\log n) \Rightarrow \Theta(p\log^2 p) = O(n\log n)$$
Since $p < n$ must hold, we also require that $\Theta(\log^2 p) = O(\log n)$. Thus, we can try the approach $\log p = \sqrt{\log n}$ and thus obtain the $cost = O\left(n\log n + 2^{\sqrt{\log n}}\log n\right)$. For large n, log(n) is greater than 1 but less than n and so the square root of log(n) is still greater than 1 but also less than log(n). Therefore, holds: $cost = O\left(n\log n + 2^{\log n}\log n\right) = O(n\log n)$
Thus, if $n = 2^{25}$, then p must not be much greater than $2^{\sqrt{\log n}} = 2^{\sqrt{25}} = 2^5 = 32$ to sort the n values in parallel at optimal cost.