In the modern world, computational power is at the peak of importance. There are several ways to improve it such as increasing transistor density, reducing latency and increasing clock rate. However, one of the easiest ways is the parallelization of the task between multiple processing units.

The degree of the increase in the computational speed between a parallel algorithm and a corresponding sequential algorithm is called speedup and expressed by ratio of $T(\text{sequential})$ to $T(\text{parallel})$.

If the given ratio exceeds $p$, where $p$ is the number of processors (cores) used, super linear speedup takes place. The most common reason for it is the cache effect. It is called that due to increased total size of cache in multiprocessor system, hence increased data transfer rate between RAM and CPU, which is cardinal to the work with the large data sets. Traditional parallel computer performance evaluation has fixed problem size and varied the number of processors, the so-called fixed-size model. In mid’80s the scaled-size model was developed and subsequently substantiated by experiments on a 1024-processor hypercube. The scaled size model specifies that the storage complexity grows in proportion to the number of processors. A third model is the fixed-time model, in which the problem is scaled to take a constant time as processors are added and rarely used in real-world applications. Algorithm described here is optimized for the fixed-size model. It is a modification of the Quicksort algorithm by C. A. R. Hoare (1962) to be utilized on a system with several processors (or cores).

On the first step, original data set is viewed as blocks of twice the size of the L1 cache (which is typically 32 or 64 kB). Processor with the smallest PID chooses the pivot element. Then all processors in parallel invoke “neutralization” function on the leftmost and the rightmost remaining blocks, effectively swapping elements respective to the value of the pivot, which leaves only $\leq P+1$ blocks to be sorted. After that remaining not “neutralized” blocks are getting swapped with the “neutralized” ones and getting sorted sequentially.

The next step is to split given data set at the pivotal point and assign processors to each half according to its size. Stack is used to keep track of the state of the sorting algorithm and the sequential steps of the recursion are turned into PUSH and POP operations on this stack. Whenever a processor encounters a small subarray, which it can fit in the cache, it will use inserting sort to sort it without PUSHing it into the stack. When a processor finished its job, it begins helping other processors by POPing out unused (yet unsorted) arrays from their stacks.

Such optimization of algorithm brings average time of partition phase to $O(N/P)$, for $N>>B$, where $N$ – number of elements, $B$ – number of elements in one block and $P$ – number of processors, and the sorting phase yields us speedup $O(P)$, provided that all processors are largely independent from one another at this stage and no synchronization required. This bring total speedup up to $T(s)/T(p) = P$, i.e. linear speedup.

Also, reduced time of memory access due to cache effect further decreases overhead and yields super linear speedup.

Hereby, via small improvements in the resource-insensitive part of the algorithm, the possibility of reaching super linear speedup while working with large datasets on multi-processor machine was shown.