A Comparative Study of Parallel Sort Algorithms

Davide Pasetto
IBM Dublin Research Lab, Dublin, Ireland
pasetto_davide@ie.ibm.com

Albert Akhriev
IBM Dublin Research Lab, Dublin, Ireland
albert_akhriev@ie.ibm.com

Abstract

In this paper we examine the performance of parallel sorting algorithms on modern multi-core hardware. Several general-purpose methods, with particular interest in sorting of database records and huge arrays, are evaluated and a brief analysis is provided.

Categories and Subject Descriptors F.2.2 [*Analysis of Algorithms and Problem Complexity*]: Nonnumerical Algorithms and Problems — parallel sorting

General Terms Algorithms, Performance

Keywords parallel sorting, sorting throughput, scalability

1. Outline of the work

Sorting is undoubtedly one of the most valuable algorithms in computer science. Recently, many parallel sorting methods have been proposed. However, despite the number of publications it is still difficult to decide which method best suits specific requirements. The lack of comparative studies has driven our interest to hardware-friendly sorting algorithms. In this research we try to shed some light on hardware specific restrictions that can make an algorithm to perform poorly even though it is theoretically efficient.

This study focuses on general-purpose sorting of huge arrays. The data-specific algorithms have been abandoned because usually an application does not know the data type in advance. A few *unstable* sorting methods, which well fit the modern SMP architectures, were carefully selected. Our choice is based on criticism found in other papers, inspection of available implementation code and intensive experimentation. Each method has $O((N/P)\log N)$ time complexity, where N is the size of input array, and P is the number of threads. The selected methods and their abbreviations are:
1) **STL.** The sort algorithm from the GNU C++ STL is used as a reference method and as a back-end for all other parallel methods. It is well designed from algorithmic and cache-usage perspectives.
2) **EM.** Edahiro's Mapsort [1] is based of the fast parallel partition procedure that splits the whole array into a number of pieces, each independently sorted by a thread using STL sort. Advantages: EM makes only two passes over the input data and moves each item only once from the input to the output array. Shortcomings: EM requires an output array and copies items to the random positions in the output array, using the memory bandwidth ineffectively.
3) **MS.** The parallel Mergesort with exact partition is based on the article [4] and popularized by developers of the GNU Multi-Core STL [2]. Advantages: most operations are cache-optimal since they read/write data in sequence maximizing the use of hardware prefetching; minimal synchronization overhead and good scalability. Shortcomings: MS requires a separate output array.
4) **TZ.** Tsigas-Zhang's Parallel Quicksort proposed in [3] is a fine-grain, block-based parallel extension to the classical quicksort algorithm. Advantages: in-place sorting; the parallel partition procedure compares/copies/swaps most of the elements only few times. Shortcomings: synchronization overhead might be high.
5) **TZJL.** A job-list extension to the previous method that divides the whole sorting job into a number of smaller jobs and pushes them into a list of pending jobs. Any free thread executes the first available job; this technique improves workload balancing.
6) **AQ.** Alternative Quicksort modifies the parallel partition procedure. At first, each thread splits its own sub-range against a pivot element, then all threads together split the whole range against the pivot in block-wise fashion, where block size can exceed L1 cache.

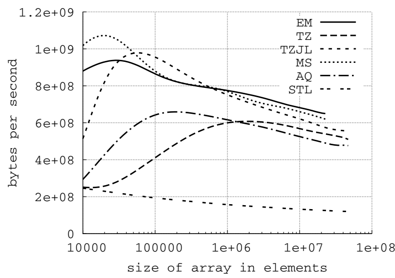
We have designed TZJL and AQ as the natural extensions to the quicksort method. The goal was to alleviate synchronization overhead and memory access bottleneck.

We have investigated several sorting scenarios like direct sorting, sorting by pointers, sorting by means of intermediate key-pointer array, etc. We were also experimenting with different data types – integers, floats and 100-byte records ordered by 10-byte keys. For the sake of space, results for direct sorting of arrays of 100-byte records will be only reported, which is our main interest here. Direct sorting implements the usual scenario when elements of array are swapped directly over the course of computation. Some methods require the output buffer while others sort in-place. The very brief summary for other sorting scenarios is that sorting by pointers is slower [5] and sorting via intermediate key-pointer array may be sometimes faster than the direct sorting.

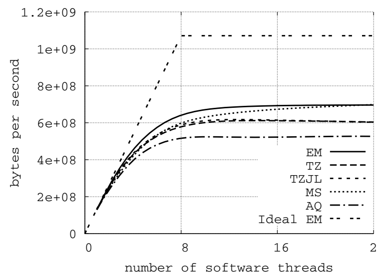
The two machines used in experiments (Core i7 architecture):
Nehalem: Xeon 5550, 2.67 GHz, 4 cores/8 threads, 3-channel memory controller, 6 Gb of memory, Linux 64-bit.
Westmere: Xeon 5670, 2.93 GHz, 6 cores/12 threads, 3-ch. mem. controller, **dual-socket** board (24 threads), 24 Gb, Linux 64-bit.

Four series of experiments have been conducted to investigate the following characteristics of the parallel sorting methods:
1) **Sorting throughput** in bytes per second that exposes the actual algorithm performance as a function of array size in elements.
2) **Scalability**, which measures the sorting throughput as a function of the number of software threads fixing the input array size.
3) **CPU affinity** influence. Two runs have been made on "Westmere" using only 12 hardware threads via affinity mask control. The first run disabled one socket, i.e. 12 threads and the cache memory of one CPU were used. The second run enabled 6 threads on both sockets, i.e. the caches of both CPUs were used. As expected, the *throughput ratio*: `(one socket throughput)/(dual socket throughput)` of the second run is smaller on *large* arrays despite synchronization overhead through the QPI connection.

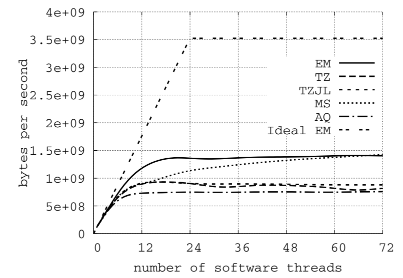
Copyright is held by the author/owner(s).
SPLASH'11 Companion, October 22–27, 2011, Portland, Oregon, USA.
ACM 978-1-4503-0940-0/11/10.



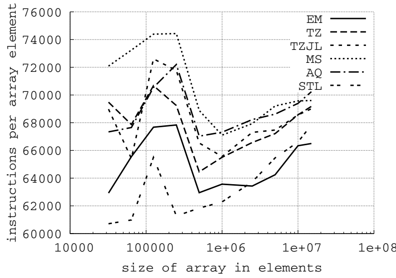
Nehalem. Sorting of 100-byte records by different methods. Sorting throughput is defined in bytes per second as a function of array size in elements.

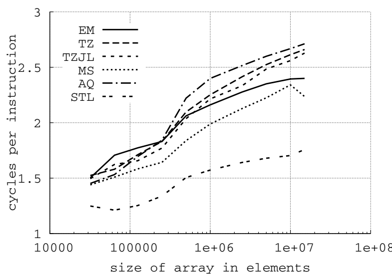


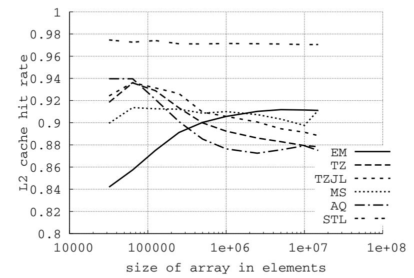
Nehalem. Scalability in the number of threads on a random array of 10^7 100-byte records. “Ideal” curve extrapolates the single-thread performance up to the number of *hardware* threads.



Westmere. Scalability in the number of threads on a random array of 10^7 100-byte records. “Ideal” curve extrapolates the single-thread performance up to the number of *hardware* threads.

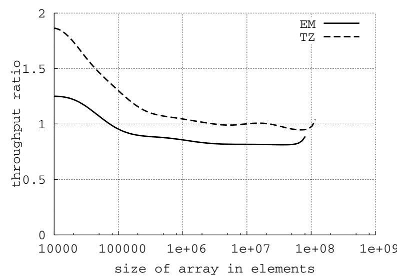






Nehalem. Performance metrics as the functions of array size in elements, where array element is a 100-byte record. *Left*, the number of instructions per element. *Middle*, the number of clock cycles per instruction. *Right*, L2 cache hit rate.

4) **Micro-architecture** analysis. The Intel Core i7 architecture is *in theory* able to execute multiple instructions per clock cycle, if implementation does not exhibit too many resource lockups inside the instruction scheduling logic. Statistics collected by `OProfile` shows the number of instructions required per input element as well as the clock cycles per instruction measured for each algorithm.



Westmere. CPU affinity test. Throughput *ratio* between using a single socket and dual sockets as a function of array size in elements, where array element is a 100-byte record.

Several observations can be made from presented diagrams. First, in-place methods TZ, TZJL and AQ demonstrate poor performance comparing to the single-threaded STL when sort small arrays (less than 100,000 elements), see the sorting throughput diagram for Nehalem. More attention should be paid to alleviate synchronization overhead, which is the main reason here.

Second, some algorithms can be slightly accelerated, if the number of *software* threads is larger than the number of available *hardware* CPU threads regardless of contention, see the scalability diagrams. The phenomenon can be explained by gradual improvement of workload balance as the number of software threads increases. There is room for further refining of sort algorithms.

Third, the scalability diagrams also make clear that achievable scalability is far below the “ideal” one. The powerful dual-socket Westmere system (24 threads) is only twice as fast than Nehalem one (8 threads). To avoid excessive inter-socket communications, the possible remedy is to divide an array into two halves on early stage and then sort both sub-arrays separately on each CPU socket.

Fourth, the figures reveal that no algorithm is able to achieve even one clock per instruction. Sorting is heavily data intensive. It requires constant load and store of words through the memory hierarchy. Modern CPUs are capable of executing several instruction per cycle only when data come mostly from registers or L1 cache.

The contribution of this paper is twofold. It gives an insight into typical problems arising from hardware limitations and presents improvements of in-place quicksort method (TZJL and AQ). We would recommend MS method, if double buffer is not a concern, and TZJL as in-place one. The full paper can be found at: https://researcher.ibm.com/researcher/view.php?person=ie-pasetto_davide, https://researcher.ibm.com/researcher/view.php?person=ie-albert_akhriev.

References

[1] M. Edahiro. Parallelizing fundamental algorithms such as sorting on multi-core processors for EDA acceleration. In *Proc. of the Asia and South Pacific Design Automation Conf.*, pages 230–233, Japan, 2009.

[2] J. Singler, P. Sanders, and F. Putze. MCSTL: The Multi-core Standard Template Library. *Lect. Notes in Comp. Science*, 4641: 682–694, 2007.

[3] P. Tsigas and Y. Zhang. A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000. In *Proceedings of the 11th Euromicro Conference on Parallel Distributed and Network based Processing*, pages 372–384, 2003.

[4] P. Varman, S. Scheufler, B. Iyer, and G. Ricard. Merging multiple lists on hierarchical-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12 (2): 171–177, 1991.

[5] C. Wu, G. Kandiraju, and P. Pattnaik. Analysis of High-Performance Sorting Algorithms on AIX for Mainframe Operation Offload. In *Proc. of the International Computer Symposium*, 2008.