A Comparative Study of Parallel Sort Algorithms

Davide Pasetto Albert Akhriev

IBM Dublin Research Lab, Mulhuddart, Dublin 15, Ireland
{pasetto_davide, albert_akhriev}@ie.ibm.com

Abstract

Sorting is among the first algorithm any computer science student encounters during college and it is considered a simple and well studied problem. In theory it contains a large amount of parallelism and should not be difficult to accelerate sorting of very large datasets on modern architectures. Unfortunately, practical Parallel Sort libraries are very hard to design, they must be carefully tuned and must rely on a number of trade-off depending on the target architecture and target application. In this paper we provide a qualitative and quantitative analysis of the performance of parallel sorting algorithms on modern multi-core hardware. We consider several general-purpose methods, which are widely regarded among the best algorithms available, with particular interest in sorting of database records and very large arrays (several gigabytes and more), whose size far exceed L2/L3 cache. We provide experimental results, a detailed analysis of shortcomings and advantages of each selected algorithm and an insight onto which algorithm is most suited for a specific application.

1 Introduction

Sorting algorithms have received a lot of attention in the past decades due to its high importance for data analysis in several application domains. A comprehensive survey would include a huge number of methods and variations like external/internal sorting, data-specific and hardware-specific sorting, etc.

Most modern SMP architectures are evolving towards a common design: a number of cores connected to a shared main memory via hierarchy of caches

with increasing size and decreasing performance. The difference in latency and bandwidth between the cache memory and the main memory can be so high that algorithms considered very efficient from a theoretical point of view could have poor performance in practice. It is well known that an efficient implementation of any algorithm cannot ignore the underlying hardware architecture and this is even more important when designing a parallel implementation. Therefore, some basic knowledge about cache/memory interaction should be embedded in the algorithm structure. When the implementation is meant to run on a parallel system, the effects of cache coherence protocols make things even more complex: data structures and algorithms must be designed to avoid false sharing and unnecessary cache coherent operations.

In order to properly take into account these issues, four internal sorting methods are investigated. Those methods have been shown to be usually fast and a good fit for the architecture of the modern multi-core/multi-thread, cache-coherent shared memory systems. This paper considers Mapsort [2], Mergesort with exact partition (whose implementation is similar to Multi-Core STL [7]), and Quicksort by Tsigas et al. [8] as well as an alternative version of the Quicksort method. Several classical algorithms [5] fell out of scope of this study either because they were found to be less efficient (e.g. SampleSort in [8]), or dedicated for distributed computing (e.g. BitonicSort) or data-specific (e.g. RadixSort).

This study approaches general-purpose sorting of very large arrays (several gigabytes and more). While there are a number of very efficient specialized methods for sorting integers, e.g. see [3], generic sorting using a custom comparison operator is much harder to optimise since data properties are not known in advance. This paper focuses on sorting of 100-byte records [6], which is very important in database applications. The performance and scalability of the algorithm on the x86_64 architecture are the central points of the evaluation. The paper considers algorithm efficiency in two different scenarios: direct sorting of 100-byte records and sorting of key/pointer pairs. This analysis is looking for the best way of optimising cache utilization.

The paper is organized as follows. Section 2 briefly outlines the implemented algorithms. Section 3 describes the target hardware used for performance evaluation. Section 4 contains the experimental results and finally section 5 provides some concluding remarks.

2 Parallel Sort

A large number of parallel sort algorithms are available in literature. Some solutions are designed or optimised for sorting specific data types while others can sort generic data; some require specific hardware configurations while others can run on any architecture. Every algorithm considered here has $O((N/P)\log N)$ time complexity, where P is the number of threads (processors), and N is the size of a data array to be sorted. The paper considers *unstable* sorting algorithms, which do not preserve the order of "equal" elements. While Mergesort and Mapsort can easily be made stable, the Tsigas-Zhang's Quicksort can not. It is, however, possible to construct a comparison operator using "the pointer value as a secondary key for stability, since input records are typically lined up in virtual memory space in ascending order" [10].

2.1 Mapsort

Edahiro's Mapsort [2], shortened in **EM**, exploits a kind of multi-pivot strategy. A number of *keys* (pivots) divide the whole array into intervals. Keys are selected at the equidistant positions in the input array and are sorted using the comparison operator, e.g. in ascending order $k_0 \leq k_1 \leq k_2 \ldots$. After several attempts to find a unique set of keys at random positions any remaining duplicate key is removed. At this stage the number of elements in an interval $[k_i, k_{i+1})$ is unknown.

The second stage counts the number of elements in each interval by comparing each entry of input array with every key using a fast binary search algorithm and a data parallel approach over separate sub-arrays. The number of elements in each interval and its offset in the output array is then determined: the 1^{st} interval starts at the position 0 in the output array, the 2^{nd} interval starts at the position N_1, where N_1 is the number of elements in the 1^{st} interval, etc.

The third stage moves elements of the input array into the output one by using counters and offsets computed in the first stage. This process groups input elements into intervals in the output array, and the intervals are properly ordered. The last stage applies in parallel a sequential sort algorithm (STL sort) to each interval. Since intervals may contain very different number of entries, it is possible to improve load balancing between threads by increasing the number of intervals. However this might render the first stage slower since the binary search algorithm is very fast only when the set of keys fits into L1 due to the intensive usage of keys in comparisons.

The main advantage of Mapsort is the very efficient parallel partition pro-

cedure: it makes only two passes over the input data and moves each element only once from the input to the output array. The obvious disadvantage of Mapsort is the requirement of a separate output area, particularly restrictive when sorting huge arrays. In the current implementation we also save interval index of each element to avoid the second binary seaching. This optimization trick consumes even more memory. Moreover, each thread writes at unpredictable position while moving elements from the input array to appropriate interval in the output array and thus might not be able to use memory bandwidth effectively.

2.2 Mergesort

The idea of parallel Mergesort, shortened in **MS**, with exact partition can be traced back to the paper of Varman et al. [9] and it was popularized by the developers of the GNU Multi-Core Standard Template Library (MCSTL) [7].

Suppose N is the size of input array, and P is the number of threads (processors). During the 1^{st} stage, each processor independently sorts an interval of size N/P using a sequential sort. The 2^{nd} stage merges all sorted intervals into the output array using the partition algorithm detailed in [9]: the 1^{st} thread takes N/P smallest elements from all sorted intervals and merges them into the interval $[0 \dots N/P)$ in the output array; the 2^{nd} thread takes the next N/P smallest elements from all sorted intervals and merges them into the interval $[N/P \dots 2(N/P))$ in the output array, and so forth. Each thread can easily pick up the smallest elements independently of any other thread. Suppose that a thread picked up n_1 elements from the first interval, n_2 elements from the second interval and so on. The merge procedure combines sorted sub-sequences of size n_1, n_2, ..., n_P into the single ordered sequence of size $\sum_i n_i$ using a tournament tree [5]. The optimal implementation makes a single traverse over the tournament tree for each element. A number of subtle technical issues have been omitted in this description for the sake of clarity, see [9, 7] for more details.

The major advantage of this algorithm is that most operations are cache-optimal since they read/write data in sequence and can maximize the use of hardware prefetching, it also has minimal synchronization overhead and excellent scalability. Mergesort shortcomings comprise: requiring a separate output array; the partition procedure is based on an unbounded sorted container (e.g. priority queue), which should be carefully implemented to avoid memory fragmentation; and finally it is algorithmically and technically complex.

2.3 Tsigas-Zhang's Parallel Quicksort

Tsigas et al. proposed in [8] a simple, fine-grain parallel extension of the classical Quicksort algorithm, shortened in **TZ** (or **TZJL** for modified version, see below). Quicksort [1] employs a divide-and-conquer strategy to divide an array into two sub-arrays via three major steps: (1) choosing a median-3 *pivot* element from the array; (2) reordering the array so that all elements with values less than the pivot come before the pivot; (3) recursively process on sub-arrays. The key feature of Parallel Quicksort is parallel partitioning the data similar to the one presented in [4]. This procedure [8] operates with blocks of elements: each thread of a group picks up a block from the left side of array and a block from the right side, and tries to "neutralize" them. A block is *neutralized* if all its elements are either less or greater the pivot. At least one of two blocks is always neutralized by the end of iteration. At the end of the phase each thread leaves either zero or one non-neutralized blocks and one thread can neutralize all remaining blocks. Upon completion of the parallel partition, all elements in a range are split against the pivot element. The group of threads is then divided into two sub-groups, proportionally to the sizes of sub-arrays, and the procedure recursively continues.

When a single thread remains in a group, the sub-array is pushed into the stack of unsorted chunks and any free thread can apply a sequential sort to it. Ideally, the partition procedure divides an array into equal-size chunks but in practice unsorted chunks can have very different sizes, thus ruining load balancing. In the implementation described here, when a single thread remains in a group, a sequential partition is performed using few additional recursions to increase the number of chunks of smaller size in the stack and increasing the chance that all threads will finalize their tasks at approximately the same time.

The block size in [8] is set to one half of L1 cache size. Modern CPUs have at least several kilobytes of L1 cache memory, so blocks should be sufficiently large to guarantee a low cache-missing rate so that when a thread finishes comparison and swapping of an element, the new one is already in the cache. This property justifies the fine-grain approach. Other nice features of this method are: in-place processing, which is very important for sorting huge arrays; low number of inter-thread synchronization points and moderate contention as the result; the parallel partition procedure touches most of the blocks only once. The method described in [8] could be considered as one of best in-place approach. Its variant has been also implemented in MCSTL [7]. A modified version of Tsigas-Zhang's Parallel Quicksort using job lists, shortened in **TZJL**, divides the whole sorting job into a number of smaller jobs (find a pivot, parallel partition, pre/post-partition steps, etc) and pushes

them into a list of pending jobs. Any free thread reads the list and executes the first available job; this technique improves workload balancing.

2.4 Alternative Quicksort

This novel alternative version, shortened in **AQ**, combines Quicksort and Mapsort ideas. The parallel partition in this case comprises the following steps:

1. pick up a median-3 *pivot* element from an array and divide the whole array into intervals of equal size by the number of threads in a group;

2. each thread independently applies sequential partition [1] splitting its own interval against the pivot into two halves; elements the first half are less than the pivot ("blacks") and elements in the second half ("whites") are greater or equal to the pivot;

3. calculate the split-point position of the whole array by summing the sizes of "black" sets of each interval;

4. each thread takes a block of "whites", located below split-point, and swaps it with a block of "blacks", located above split-point; as the result elements below split-point are all "blacks" and elements above split-point are all "whites";

5. recursively proceed on the two sub-arrays.

In contrast to the previous algorithm, data swapping can be done efficiently with blocks larger than L1 cache size because data is always read in order. Another advantage of this approach is that its implementation is even simpler than [8] one. Touching the memory twice is the big disadvantage of this approach: the memory is accessed first when a thread divides an interval, and then when blocks of "black" and "white" elements are being swapped.

2.5 STL sort

In the performance evaluation we use a standard single-threaded sort, shipped with GNU C++ Standard Template Library, as a reference method, shortened in **STL**. This implementation is a masterpiece: it consists of three sorting algorithms applied to the input array in order. First, the quicksort, based on divide-and-conquer strategy, eliminates disorder among remote elements. To avoid very deep recursion, a Heap-sort might finalize the recursion branch. At the last stage, when elements to be swapped are sufficiently close

to each other, an Insertion-sort performs the final job. It is very difficult to significantly outperform STL sort, which is extremely good both from an algorithmic and cache-usage perspectives. Standard STL sort is used as a back-end for all parallel methods.

3 Target Architectures

The two machines used in experiments are based on the Core i7 architecture:

Nehalem: Intel Xeon 5550, 2.67 GHz, 4 cores/8 threads, 6 Gb of memory, 3-channel memory controller, OS Linux Fedora 13 64-bit.

Westmere: Intel Xeon 5670, 2.93 GHz, 6 cores/12 threads, **two sockets** on board (24 threads in total), 24 Gb of memory, 3-channel memory controller, OS Linux RedHat 5.4 64-bit.

The "Nehalem" is a 45nm quad core processor. Each core has a private L1 and L2 cache, while the L3 cache is shared across the cores on a socket. Each core supports Simultaneous Multi Threading (SMT), allowing two threads to share processing resources in parallel on a single core. The system has a 32KB L1, 256KB L2 and a 8MB L3 cache. As opposed to older processor configurations, this architecture implements an inclusive last level cache. Each cache line contains core valid bits that specify the state of the cache line across the processor. A set bit corresponding to a core indicates that the core may contain a copy of this line. When a core requests for a cache line that is contained in the L3 cache, the bits specify which cores to snoop, for the latest copy of this line, thus reducing the snoop traffic. The MESIF cache-coherence protocol extends the native MESI protocol to include "forwarding". This feature enables forwarding of unmodified data that is shared by two cores to a third one.

The "Westmere", on the other hand, is a 32nm six cores per socket solution, with a core configuration identical to the "Nehalem" one. This architecture expands the shared L3 cache size to 12 MB per socket and uses a fast kilobit-wide ringbus between caches to boost access speed. "Westmere" also implements two-way SMT per processing core and contains 3 DDR3 channels per chip. The 4-channel QPI interconnect at 6.4 GigaTransactions per second gives over a 100 GB/sec bandwidth to the other neighbouring sockets in a blade.

4 Performance Evaluation

Several technical remarks should be made before detailing performance result. Some methods require intermediate buffers of the same size as the input array. When sorting huge arrays memory management may take a long time, especially in a parallel environment. All described methods are implemented as C++ classes, which hold persistent intermediate buffers until the destructor is invoked. This approach is justified when sort is performed many times during application lifetime.

Some algorithms (Mapsort, Quicksort) can be slightly accelerated, if the number of threads used is larger than the number of CPU cores available, regardless of OS overhead. The rationale is better workload balance when the jobs, executed by separate threads, have a large variability in their computation time. However, too many software threads can cause additional contention. In any case, this is a marginal way for increasing performance and it is a clear sign of non-optimal workload balancing in the algorithm.

In this study we focus on sorting of 100-byte database records using two distinct techniques:

Direct sorting. Elements of an array are swapped directly. Some methods require an intermediate buffer while others can sort the input array in-place.

Key-Pointer sorting. The input data is an array of 100-byte records. A temporary array of key/pointer pairs is created, where *key* is a 10-byte excerpt of a record and a pointer points to the corresponding input record. A direct method sorts the array of key/pointer pairs by key values. Given the array of sorted key/pointer pairs and the output one, the elements of input array can be reordered accordingly. Three arrays are involved: the input one, the output one and the array of key/pointer pairs.

One possible option is sorting by pointers, however we found that it is slower than direct sorting, see also [10]. This probably happens due to the nature of STL sort, which is the back-end of any method reported here. STL sort recursively splits a range into two smaller sub-ranges. When a range becomes small enough the Insertion-sort (with average cost $O(N^2)$) is applied. It is important that on the late stages the elements to be swapped are located close to each other because swapping is intensive. When sorting pointers, on each iteration sequential elements do not get closer, but their pointers do. As the result, the algorithm often reads/writes remote elements and cache-missing rate gets high. Though direct sorting swaps the whole

records, the expense is compensated by the better cache utilization. This explains why Key-Pointer sorting might be fast. A key/pointer pair is compact (10+8 bytes) and can be compared and swapped efficiently. The pointers in the pairs are used only once: when copying elements from the input array into the ordered output one.

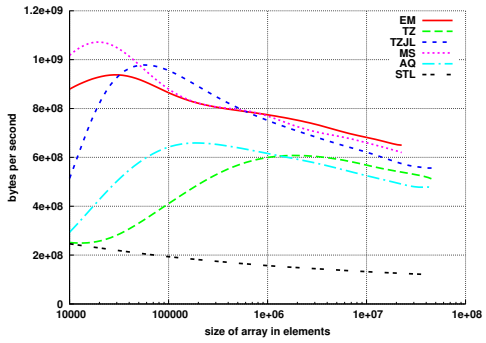

Figure 1: Nehalem. Sorting of 100-byte records. Throughput (bytes per second) as a function of array size in elements.

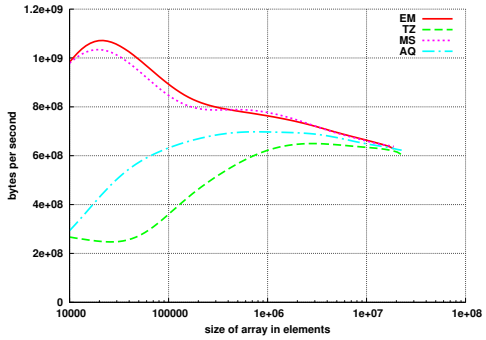

Figure 2: Nehalem. Key-Pointer sorting. Throughput (bytes per second) as a function of array size in elements.

4.1 Sorting throughput

The first set of experiments is focused on the sorting speed measured in bytes per second. Sorting throughput is the most important characteristic, since it exposes the actual performance of an algorithm as seen by the end-user. Fig. 1 to Fig. 4 detail sorting throughput as a function of input array size

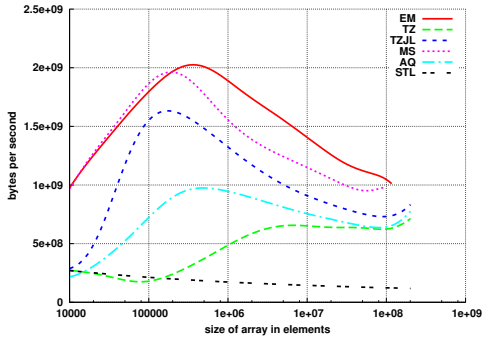

Figure 3: Westmere. Sorting of 100-byte records. Throughput (bytes per second) as a function of array size in elements.

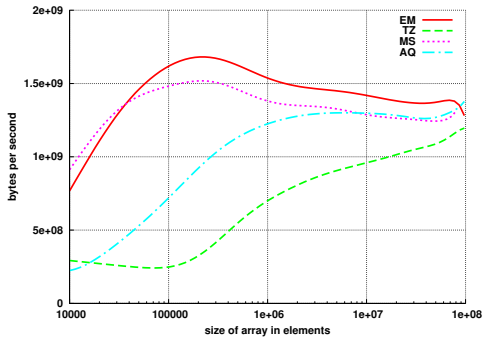

Figure 4: Westmere. Key-Pointer sorting. Throughput (bytes per second) as a function of array size in elements.

(in elements). The plots are smoothed by Bezier splines for better visibility. It is easy to see that Mapsort and Mergesort are the fastest algorithms. Quicksort methods, on the other hand, are not significantly slower than the leaders. Both in-place methods (Tsigas-Zhang's and Alternative Quicksort) demonstrate comparable performance.

Three other observations should be made. First, performance on short arrays is not good. Given the high runtime variability due to data distribution perfect workload balancing between threads is very hard to achieve when there is not enough data to handle. As the size the input array increases the probability that all jobs have similar complexity becomes higher. Second, maximum performance occurs when the size of array is several times larger than the size of L2/L3 cache, when often happens that the "distance" between a couple of elements, which should be compared and swapped, is

less than the cache size. Finally, performance degrades as the size of array increases beyond a certain limit. Large size causes many cache misses during comparison and swapping of remote elements and all these transactions quickly saturate the memory bus.

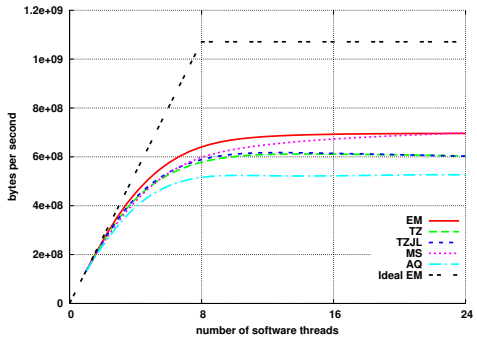

Figure 5: Nehalem. Strong scalability on a random array of 10^7 100-byte records.

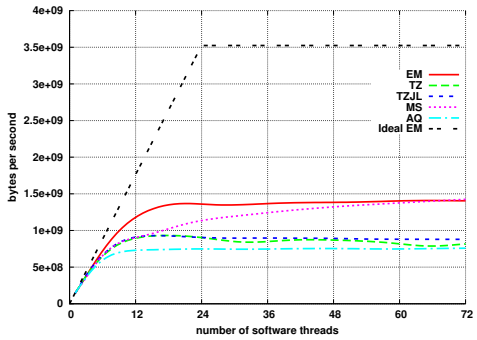

Figure 6: Westmere. Strong scalability on a random array of 10^7 100-byte records.

4.2 Scalability

Another important characteristics of a parallel algorithm is the strong scalability, which measures throughput as a function of the number of threads fixing the input data size. Fig.5 and Fig.6 report strong scalability results. It can be seen that the processing rate slows down well before the number of working threads reaches the number of cores. The "ideal" curve on Fig.5

and Fig.6 extrapolates single-thread performance and shows the theoretical limit for the MapSort algorithm.

Scalability of the algorithms varies but its characteristics do not depend on the target architecture. For example, MapSort always demonstrates the best scalability as the number of threads increases. Obviously, all algorithms show some amount of workload imbalance: the performance keeps growing (in some cases also significantly) by increasing the number of threads well above the number of cores (8 on Nehalem and 24 on Westmere). The algorithm with worst workload balance is Mergesort, but this implementation is able to outperform MapSort when the number of threads is 3 times (or more) the number of cores.

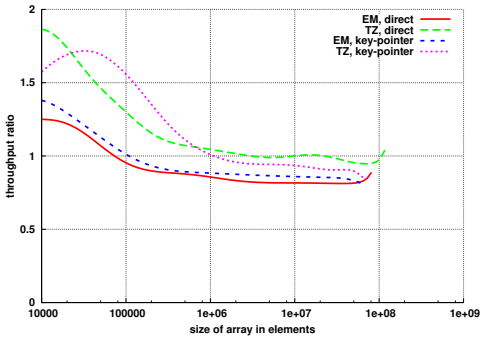

Figure 7: Throughput ratio between using a single or dual sockets in a Westmere system. Direct and key-pointer sorting of 100-byte records.

4.3 Influence of CPU affinity

Another option that might affect algorithm performance is CPU affinity. Several experiments were made by defining sub-sets of processing units on which the algorithm is eligible to run. One experiment is conducted on "Westmere" using only 12 cores. A first run uses a CPU affinity mask so that the second socket is not available and a second run enables 6 hardware threads on the first socket and 6 hardware threads on the second one. During the first run the cache memory is effectively halved, since it is restricted to the first socket, whereas in the second run the cache of both CPUs is utilized. Fig. 7 shows the throughput ratio:

$$\text{throughput ratio} = \frac{\text{single socket throughput}}{\text{dual socket throughput}}.$$

As expected the throughput of the second run is always higher on large arrays, even if the two sockets must run cache coherence protocol through the QPI connection. On mid-size arrays the situation is the opposite and small arrays usually demonstrate unstable performance figures because sorting time heavily depends on the underlying data distribution. Based on these results, it is generally beneficial to distribute the workload on large arrays across all sockets, while a small problem can be solved more efficiently within a single socket due to the reduced overhead of cache coherence protocol.

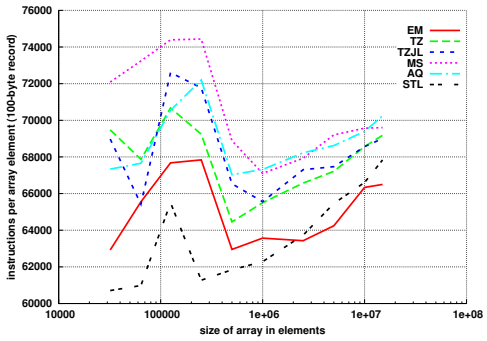

Figure 8: Instructions per element.

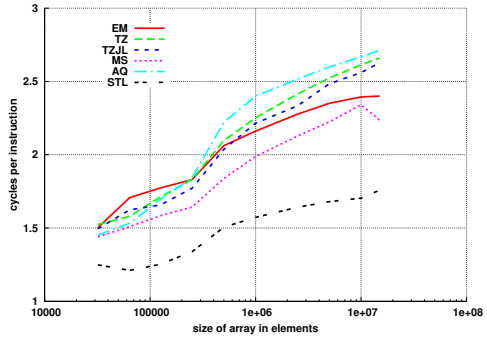

Figure 9: Clocks per instruction.

4.4 Micro-architecture analysis

The Intel Core i7 architecture is *in theory* able to execute multiple instructions per clock cycle, if the algorithm and its implementation do not exhibit too many resource lockups inside the instruction scheduling logic. Fig. 8 and

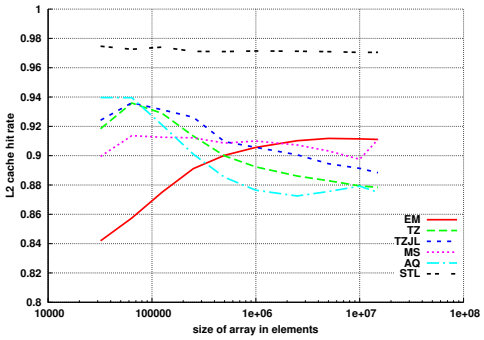

Figure 10: L2 cache hit rate.

Fig. 9 analyze, using hardware performance counters, the number of instructions required per input element and correlates it with the clock cycles per instruction measured for each algorithm. `OProfile` has been used to provide performance counter analysis.

The first graph (Fig. 8) shows an unstable behavior for small arrays since data distribution heavily influences algorithmic performance. On larger arrays the behavior is more consistent. It is easy to see that (as expected) single threaded sorting uses the least number of instructions per input byte and: when processing is parallelized a large number of instructions (overhead) are added to coordinate the various threads. MergeSort is the worst algorithm in this respect adding about 15% more instructions to its sequential counterpart.

Looking at the clock per instruction figures (Fig. 9) we can see that no algorithm is able to achieve even one CPI (clock per instruction). Sorting is heavily data intensive. It requires constant load and store of words through the memory hierarchy. Modern CPU are capable of executing several instruction per cycle only when data come mostly from internal registers and/or L1 cache. Sequential sort of course obtains the best CPI, since it has more cache and memory bandwidth resources available, while among the parallel implementation MapSort is the best one in term of core utilization. Processing of large arrays is of course memory bound and achieves a very low core utilization.

Another interesting analysis is shown in Fig. 10 which details the hit rate of L2 cache for the algorithms considered vs input array size. It is easy to see that great care was taken in the design of STL sequential sort: this implementation is really capable of achieving extremely high cache hit rates across the entire range of input sizes. The parallel implementations, on the other

hand, are not that capable since the use of caches across threads is not coordinated: the various cores go mostly "on their own" and keep stealing cache space each other thus effectively reducing cache effectiveness. As expected, the grater the input data the worse the situation is, since the probability that independent threads work on data far from each other increases. However, the MapSort shows somewhat counter-intuitive: this algorithm exhibits an awful cache usage for small input sizes, but the situation greatly improves as the input size increases.

5 Conclusion

This paper considers several powerful parallel sorting algorithms with particular interest in sorting very large arrays of (database) records. Algorithms' performance is investigated under two different sorting scenarios and on two machines. The study also details experiments with CPU affinity masks and a microcode analysis of the dynamic behavior.

The paper has been focused on general-purpose sorting that uses a custom comparison operator. This rules out many nice specialized algorithms from the consideration, but it is helpful when data structure is not known in advance.

The main conclusion is that parallel sorting algorithms are very hard to parallelise because they are data intensive: the procedure should pick-up elements from arbitrary locations in the input (unsorted) array and bring them together upon completion. This implies a large number of memory transactions between CPU cores and main memory through the cache hierarchy. When an array does not fit in L2/L3 cache, the memory sub-system always becomes a bottleneck.

In fact, a large memory bandwidth plays a vital role for sorting. For example, one of the systems used for testing has a three-channel memory controller, but initially only two channels were in operation. After inserting additional DIMMS into the free slots we gained more than 20% throughput on all tested algorithms – a clear sign where the problem is.

Strong scalability, in terms of speed-up when increasing the number of cores, is very hard to achieve: it is not possible to achieve significant speed-up, with respect to single-threaded STL sort. For example Westmere, which has 24 hardware threads and a huge L3 cache, demonstrates speed-up factor around 10. This means that more attention should be paid to cache-usage patterns in future design of sort algorithm.

Authors recommend the Mergesort and MapSort algorithms when memory is not an issue. Both these methods require an intermediate array, while

the slower Quicksort methods (Tsigas-Zhang's and Alternative Quicksort) work in-place and are almost equivalent. The recommendation covers only the case of *large arrays* ($> 100,000$ elements). On small arrays the synchronization overhead comes into play and a separate research should be done to reveal a winner.

One should bear in mind that figures reported in this paper are subject to change with new compilers and libraries. For example, after updating GNU G++ compiler to the version 4.5.2 we have noticed a drop in performance for all methods. Brief investigation revealed that comparison operator, built on the standard function `memcmp()`, makes a problem. The function `memcmp()`, which was used to compare 100-byte records, has been replaced by explicit loop. As the result, performance increased even comparing to the previous figures. Also optimization strategy should be carefully selected. For example, the best optimization flag `-O3` does not necessarily produce the best code; for some implementations `-O2` is slightly better. All these small things have to be taken into account to make a fair comparison of different sort algorithms.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.

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

[3] Y. Han and M. Thorup. Integer Sorting in $O(n\sqrt{\log\log n})$ Expected Time and Linear Space. In *In Proceedings of the 43rd Symposium on Foundations of Computer Science*, pages 135–144, IEEE Computer Society, Washington, DC, Nov. 2002.

[4] P. Heidelberger, A. Norton, and J. T. Robinson. Parallel quicksort using Fetch-and-Add. *IEEE Transactions on Computers*, 39(1):133–137, January 1990.

[5] D.E. Knuth. *Introduction to Algorithms*, volume 3: Sorting and Searching. Addison-Wesley, 1998.

[6] Nyberg, C. and Shah, M. and Govindaraju, N. Sort Benchmark (web resource). Available from `http://sortbenchmark.org/`.

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

[8] 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.

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

[10] C.E. Wu, G. Kandiraju, and P. Pattnaik. Analysis of High-Performance Sorting Algorithms on AIX for Mainframe Operation Offload. Technical report, IBM Research Division, 2008. Available from `http://dspace.lib.fcu.edu.tw/handle/2377/11094`.