A Parallel Quicksort Algorithm on Manycore Processors in Sunway TaihuLight

Siyuan Ren, Shizhen Xu, and Guangwen Yang

Tsinghua University, China

Abstract. In this paper we present a highly efficient parallel quicksort algorithm on SW26010, a heterogeneous manycore processor that makes Sunway TaihuLight the Top-One supercomputer in the world. Motivated by the software-cache and on-chip communication design of SW26010, we propose a two-phase quicksort algorithm, with the first counting elements and the second moving elements. To make the best of such manycore architecture, we design a decentralized workflow, further optimize the memory access and balance the workload. Experiments show that our algorithm scales efficiently to 64 cores of SW26010, achieving more than 32X speedup for int32 elements on all kinds of data distributions. The result outperforms the strong scaling one of Intel TBB (Threading Building Blocks) version of quicksort on x86-64 architecture.

1 Introduction

This paper presents our design of parallel quicksort algorithm on SW26010, the heterogeneous manycore processor making the Sunway TaihuLight supercomputer currently Top-One in the world [4]. SW26010 features a cache-less design with two methods of memory access: DMA (transfer between scratchpad memory (SPM) and main memory) and Gload (transfer between register and main memory). The aggressive design of SW26010 results in an impressive performance of 3.06 TFlops, while also complicating programming design and performance optimizations.

Sorting has always been a extensively studied topic [6]. On heterogeneous architectures, prior works focus on GPGPUs. For instance, Satish et al. [9] compared several sorting algorithms on NVIDIA GPUs, including radix sort, normal quicksort, sample sort, bitonic sort and merge sort. GPU-quicksort [2] and its improvement CUDA-quicksort [8] used a double pass algorithm for parallel partition to minimize the need for communication. Leischner et al. [7] ported samplesort (a version of parallel quicksort) to GPUs, claiming significant speed improvement over GPU quicksort.

Prior works give us insights on parallel sorting algorithm, but cannot directly satisfy our need for two reasons. First, the Gload overhead is extremely high so that all the accessed memory have to be prefetched to SPM via DMA. At the same time, the capacity of SPM is highly limited (64KiB). Second, SW26010 provides a customized on-chip communication mechanism, which opens new opportunities for optimization.

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_61

Based on these observations, we design and implement a new quicksort algorithm for SW26010. It alternates between parallel partitioning phase and parallel sorting phase. During first phase, the cores participate in a double-pass algorithm for parallel partitioning, where in the first pass cores count elements, and in the second cores move elements. During the second phase, the cores sort its assigned pieces in parallel.

To make the best of SW26010, we dispense with a central manager common in parallel algorithms. Instead we duplicate the metadata on SPM of all worker cores and employ a decentralized design. The tiny size of the SPM warrants special measures to maximize its utilization. Furthermore, we take advantage of the architecture by replacing memory access of value counts with register communication, and improving load balance with a simple counting scheme.

Experiments show that our algorithm performs best with int32 values, achieving more than 32 speedup (50% parallel efficiency) for sufficient array sizes and all kinds of data distributions. For double values, the lowest speedup is 20 (31% efficiency). We also compare against Intel TBB's parallel quicksort on x86-64 machines, and find that our algorithm on Sunway scales far better.

2 Architecture of SW26010

SW26010 [4] is composed of four core-groups (CGs). Each CG has one management processing element (MPE) (also referred as manager core), 64 computing processing elements (CPEs) (also referred as worker cores). The MPE is a complete 64-bit RISC core, which can run in both user and kernel modes. The CPE is also a tailored 64-bit RISC core, but it can only run in user mode. The CPE cluster is organized as an 8x8 mesh on-chip network. CPEs in one row and one column can directly communicate via register, at most 128 bit at a time. In addition, each CPE has a user-controlled scratch pad memory (SPM), of which the size is 64KiB.

SW26010 processors provide two methods of memory access. The first is DMA, which transfers data between main memory and SPM. The second is Gload, which transfers data between main memory and register, akin to normal load/store instructions. The Gload overhead is extremely high, so it should be avoided as much as possible.

Virtual memory on one CG is usually only mapped to its own physical memory. In other words, four CGs can be regarded as four independent processors when we design algorithms. This work focuses on one core group, but we will also briefly discuss how to extend to more core groups.

3 Algorithm

As in the original quicksort, the basic idea is to recursively partition the sequence into subsequences separated by a pivot value. Values smaller than the pivot shall be moved to the left, larger to the right. Our algorithm is divided into two phases to reduce overhead. The first phase is parallel partitioning with a two

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_61

pass algorithm. When the pieces are too many or small enough, we enter the second phase, when each core independently sorts its pieces. Both phases are carried out by repeated partitioning with slightly different algorithms.

3.1 Parallel Partitioning

Parallel partitioning is the core of our algorithm. We employ a two pass algorithm similar to [2,1,10]. in order to avoid concurrent writes. In the first pass, each core counts the total number of elements strictly smaller than and strictly larger than the pivot in its assigned subsequence. It does so by loading consecutively the values from main memory into its SPM and accumulating the count. The cores then communicate with one another about their counts, with which they can calculate the position by cumulative sum where they should write to in the next pass.

In the second pass, each core does their own partitioning again, this time directly transferring the partitioned result into their own position in the result array. This step can be done in parallel since all of the reads and writes are disjoint. After all cores commit their result, the result array is left with a middle gap to be filled by the pivot values. The cores then fill the gap in parallel with DMA writes.

The synchronization needed by the two pass algorithm is hence limited to only these places: a barrier at the end of counting pass, the communication of a small number of integers, and the barrier after the filling with pivots.

3.2 Communication of Value Counts

Because the value counts needed for calculation of target location are small in number, exchanging them through main memory among worker cores, either via DMA or Gload, would result in a great overhead. We instead decide to let the worker cores exchange the counts via register communication, with which the worker cores can transfer values at most 128bit at a time. The smaller and larger counts are both 32-bit, so they can be concatenated into one 64-bit value and communicated in one go.

Each worker core needs only two combined values: one is the cumulative sum of counts for cores ordered before it, another is the total sum of all counts. The information flow is arranged in a zigzag fashion to deal with the restriction that cores can only communicate with one another in the same row or column.

3.3 Load Balancing

Since Sunway has 64 cores, load imbalance is a serious problem in phase II. If not all the cores finish their sorting at the same time, those that finish early will have to sit idle, wasting cycles. To reduce the imbalance, we employ a simple dynamic scheme based on an atomic counter.

To elaborate, we dedicate a small fraction of each SPM to hold the metadata of array segments that all of them are going to sort independently in parallel.

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_61

When the storage of metadata is full, each core will enter phase II and choose one segment to sort. When any core finishes, it will atomically increment an counter in the main memory to get the index of next segment, until the counter exceeds the storage capacity, and the algorithm either returns to phase I or finishes.

3.4 Memory Optimization

As SPM is very small (64KiB), any memory overhead will reduce the number of elements it can buffer at a time, thereby increasing the rounds of DMAs. Memory optimization is therefore critical to the overall performance. We employ the following tricks to further reduce memory overhead of control structures.

For one, we use an explicit stack, and during recursion of partitioning at all levels, we descend into the smaller subarray first. This bounds the memory usage of the call stack to $O(\log_2 N)$, however the pivot is chosen [5].

For another, we compress the representation of subarrays by converting 64-bit pointers to 32-bit offsets, and by reusing the sign bit to denote the base of the offset (either the original or the auxiliary array). The compression can reduce the number of bytes needed for each subarray representation from 16 bytes to 8 bytes, a 50% save.

3.5 Multiple Core Groups

To apply our algorithm to multiple core groups, we may combine the single core group algorithm with a variety of conventional parallel sorting algorithms, such as samplesort. Samplesort on n processors is composed of three steps [3]: partition the array with $n-1$ splitters into n disjoint buckets, then distribute them onto n processors so that i-th processors have the i-th bucket, and finally sort them in parallel. To adapt our algorithm to multiple core groups, we simply regard each core group as a single processor in the sense of samplesort, and do the first step with our parallel partitioning algorithm (Sect. 3.1) with a slight modification (maintain n counts and do multi-way partitioning).

4 Experiments

To evaluate the performance of our algorithm, we test it on arrays of different sizes, different distributions, and different element types. We also test the multiple CG version against single CG version. To evaluate how our algorithm scales, we experiment with different number of worker cores active. Since there is no previous work on Sunway or similar machines to benchmark against, we instead compare our results with Intel TBB on x86-64 machines.

Sorting speed is affected by data distributions, especially for quicksort since its partitioning may be imbalanced. We test our algorithm on five different distributions of data. See Fig. 1 for the visualizations of the types of distributions.

For x86-64 we test on an AWS dedicated instance with 72 CPUs (Intel Xeon Platinum 8124M, the latest generation of server CPUs in 2017). The Intel TBB

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_61

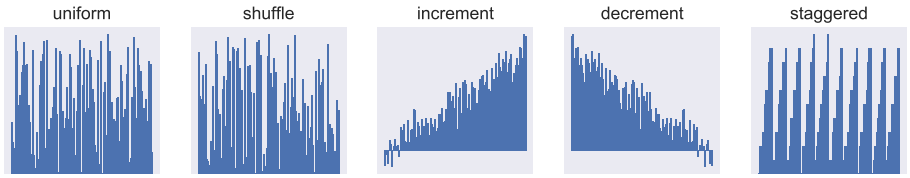


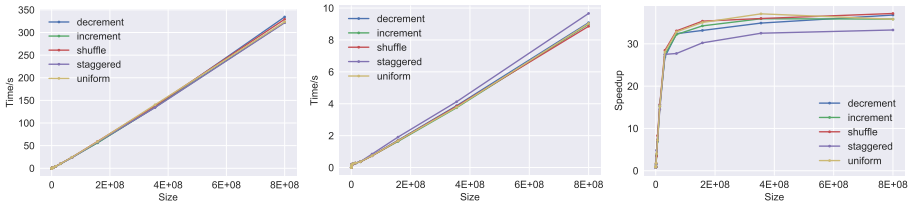
Fig. 1: Visualizations of data distributions. The horizontal axis represents the index of element in the array, and the vertical axis the value.

library is versioned 2018U1. Both the library and our test source are compiled with `-O3 -march=native` so that compiler optimizations are fully on.

4.1 Results on Sunway TaihuLight

We compare the running time of our algorithm on Sunway TaihuLight against single threaded sorting on the MPE with `std::sort`. The STL sort, as implemented on libstdc++, is a variant of quicksort called introsort.

Fig. 2 shows the runtime results for sorting 32-bit integers. From the graph we can see that the distribution matters only a little. Fig. 3 shows sorting different types of elements with the size fixed. The reason for the reduced efficiency with 64-bit types (int64 and double) is evident: the number of elements buffered in SPM each time is halved, and more round trips between main memory and SPM are needed. The reason for reduced efficiency of float32 values is unknown. Fig. 4 shows the timings and speedups of multiple CG algorithm (adapted samplesort).



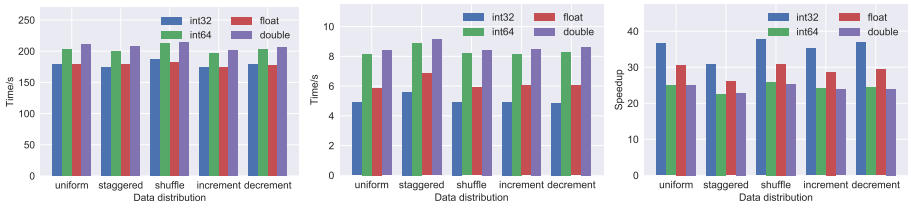
(a) STL (b) Ours (c) Speedup

Fig. 2: Results for int32 values

4.2 Comparison against Intel TBB on x86-64

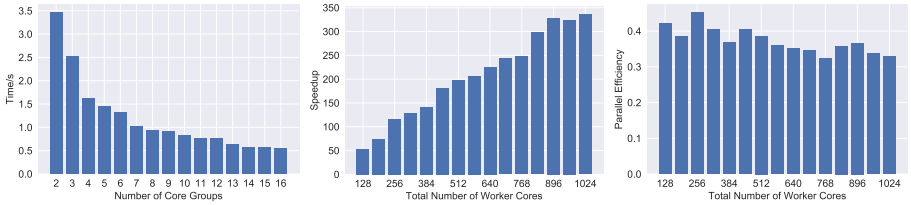
We compare our implementation against Intel TBB on Intel CPU. TBB is a C++ template library of generic parallel algorithms, developed by Intel, and

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_61



(a) STL (b) Ours (c) Speedup

Fig. 3: Results for different element types



(a) Timings (b) Speedups (c) Parallel Efficiency

Fig. 4: Results for different number of core groups

most optimized for their own processors. For a fairer comparison, we choose a machine with one of the most powerful Intel processors available to date.

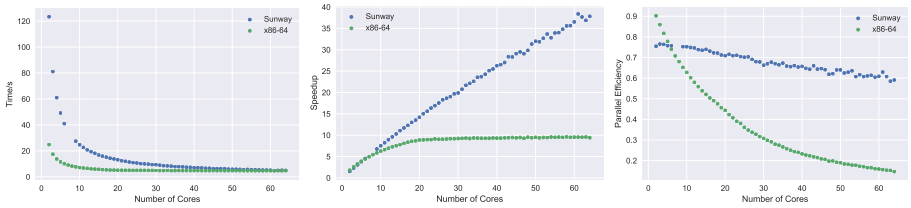
The result is illustrated at Fig. 5. We can see that an individual x86-64 core is about six times as fast as one SW26010 worker core, but our algorithm scales much better with the number of cores. The performance of TBB's algorithm saturates after about 20 cores are in use, whereas our algorithm could probably scale further from 64 cores, judging from the graph. Even though the comparison isn't direct since the architecture is different, it is evident that our algorithm on top of Sunway TaihuLight is much more efficient than traditional parallel sorting algorithms implemented on more common architectures.

5 Conclusion

In this paper, we present a customized parallel quicksort on SW26010 with significant speedup relatively to single core performance. It is composed of two-pass parallel partitioning algorithm with the first counting elements and the second moving elements. This design is able to leverage the on-chip communication mechanism to reduce synchronization overhead, and fast on-chip SPM to minimize the data movement overhead. Further, we design a cooperative scheduling scheme, and optimize memory usage as well as load balancing.

Experiments show that for int32 values, our algorithm achieves a speedup of more than 32 on 64 CPEs and a strong-scaling efficiency 50% for all distri-

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_61



(a) Sorting time (b) Speedup (c) Parallel Efficiency

Fig. 5: Results for different cores on SW26010 (our algorithm) vs on x86-64 (TBB)

butions. Compared with Intel TBB's implementation of parallel quicksort on x86-64 architecture, our design scales well even when using all of 64 CPEs while TBB's implementation hardly benefit from more than 20 cores.

References

1. Blelloch, G.E.: Prefix sums and their applications. Tech. rep., Synthesis of Parallel Algorithms (1990), `https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf`
2. Cederman, D., Tsigas, P.: GPU-Quicksort: A practical quicksort algorithm for graphics processors. Journal of Experimental Algorithmics **14**, 4 (2009)
3. Frazer, W.D., McKellar, A.C.: Samplesort: A sampling approach to minimal storage tree sorting. J. ACM **17**(3), 496–507 (Jul 1970)
4. Fu, H., Liao, J., Yang, J., Wang, L., Song, Z., Huang, X., Yang, C., Xue, W., Liu, F., Qiao, F., Zhao, W., Yin, X., Hou, C., Zhang, C., Ge, W., Zhang, J., Wang, Y., Zhou, C., Yang, G.: The Sunway TaihuLight supercomputer: system and applications. Science China Information Sciences **59**(7), 072001 (Jun 2016)
5. Hoare, C.A.R.: Quicksort. The Computer Journal **5**(1), 10–16 (1962)
6. Knuth, D.E.: The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1998)
7. Leischner, N., Osipov, V., Sanders, P.: GPU sample sort. In: 2010 IEEE International Symposium on Parallel Distributed Processing. pp. 1–10 (April 2010)
8. Manca, E., Manconi, A., Orro, A., Armano, G., Milanesi, L.: CUDA-quicksort: an improved GPU-based implementation of quicksort. Concurrency and Computation: Practice and Experience **28**(1), 21–43 (2016)
9. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore GPUs. In: 2009 IEEE International Symposium on Parallel Distributed Processing. pp. 1–10 (May 2009)
10. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for GPU computing. In: Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware. pp. 97–106. GH '07, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2007)

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_61