# Linear Time Sorting for Large Data Sets with Specialized Processor

Teemu Kerola
University of Helsinki
Helsinki, Finland
teemu.kerola@cs.helsinki.fi

## ABSTRACT

A GPU-type special processor is proposed for sorting large data sets with, e.g., one billion keys. The parallel bubble sorter (PBS) processor implements parallel bubble sort in hardware with a very large set of special registers. A method to utilize the PBS for various large sorting problems is presented.

## CCS CONCEPTS

• **Computing methodologies → Massively parallel algorithms**;
• **Computer systems organization → Parallel architectures**;

## KEYWORDS

parallel, sort, processor, bubble sort

**ACM Reference Format:**
Teemu Kerola. 2018. Linear Time Sorting for Large Data Sets with Specialized Processor. In *Proceedings of 47th International Conference on Parallel Processing (ICPP 2018).* ACM, New York, NY, USA, 2 pages.

## 1 INTRODUCTION

Sorting is one of the key problems in data processing. It compasses arranging a given data set in ascending or descending order according to given key in each data set element. The key may be an integer or floating point value, or a character string. Various algorithms have been proposed for sorting, and in general they have time complexity $O(n \log n)$ where $n$ is the number of keys to be sorted. Many sorting algorithms have been modified for multiprocessor or multicore systems, but their speedup can at best be the number of processors or cores in the system which is still quite modest, usually tens or at most hundreds [1]. Graphical Processing Units (GPU) may be used for general purpose computing (GPGPU), and sorting algorithms have been fine-tuned for them [2][3][8]. However, again the best possible speedup is still the number of cores in such systems, and that is currently only in the thousands. Sorting can also be done in distributed systems, but there communication delays form a significant component in overall time [7][6].

We propose a GPU-like new processor, Parallel Bubble Sorter (PBS), which would be specialized only for sorting problems [5]. It would contain a very large (e.g., billions or even trillions) set of special processors to do sorting in linear time. We also propose a method on how PBS may be used in various sorting problems, including those that have more keys than current PBS implementation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*ICPP 2018, August 13-16, 2018, Eugene, Oregon, USA*
© 2018 Association for Computing Machinery.

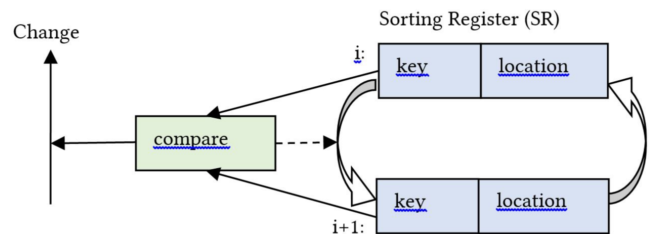


**Figure 1: PBS sorting element**

## 2 METHODOLOGY

In general, the bubble sort is one of the worst sorting algorithms, and it takes $O(n^2)$ time in one processor system. The PBS has a very large set ($N$) of special sorting registers, and, if $n \leq N$, PBS will perform $n$ comparisons in parallel and sort the data set in $O(n)$ time with very small factors. The number of sorting registers should be as large as technology allows to implement in one chip, e.g., millions, billions or trillions.

The PBS sorting registers (SR) contain two fields (see Fig. 1), a key-value on which sorting is done, and a location indicator for this key value in the original data set. The key values may be (e.g., 64-bit ) integer or floating point values, or short character strings. The location indicators may be (e.g.) 64-bit indexes for arrays, pointers to memory, data base record addresses in file or cloud servers. However, both fields should be relatively short so that we can implement as many as possible SRs in one PBS.

One sorting element (see Fig. 1) will also include comparison circuits, separately for integer, floating point, and (short) string comparisons. At one sorting step, the key values in two adjacent SRs are compared, and if they are in wrong order, all contents of those registers are interchanged.

Sorting is done alternating even and odd steps. In even step, each key in even numbered SR is compared to the key in the following SR, and register values are interchanged if needed. All pairwise comparisons and possible interchanges are completely independent of each other, and may be done in parallel. In odd step the same is done for all odd numbered SRs. With proper wiring, the same compare circuits can be used both in even and odd steps.

Sorting is completed at the latest after n steps [4]. Sorting may also be completed earlier, which is observed when two subsequent sorting steps happen without interchanges. If the key set is already

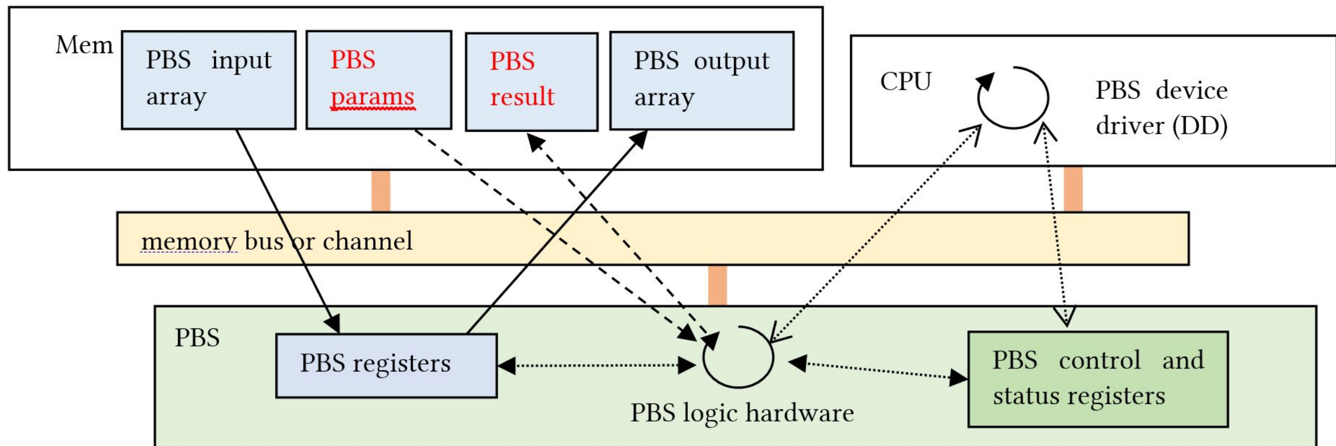


**Figure 2: PBS as external device in system**

sorted, sorting terminates immediately after the $1^{st}$ two steps. The termination test is easy to implement. If any one of the sorting elements needs a value interchange, it asserts the shared Change signal (see Fig. 1). The Change signal value is stored and two subsequent not-asserted values indicate sorting to be completed. Also, a separate counter can be kept to count the number of sorting steps needed, and returned to the driver using PBS.

If the key set is too large for the PBS ($n > N$), then it must be partitioned into size n segments, which are sorted separately with PBS and then merged. Splitting such ultra-large key set and subsequent merging may be done at various levels, e.g., by the application, by the PBS device driver, or even by the PBS device controller which may control many PBS processors.

Sorting of the original data set is done in multiple phases. The application first defines and extracts the input array (with key, location pairs) from the original data set (see Figure 2). The application calls PBS device driver, which invokes PBS to do the sorting. PBS reads the input array from memory to sorting registers (e.g., utilizing DMA or shared memory), sorts it, and saves the resulting output array back to memory. Finally, the application reorganizes the original data set according to the location fields in the output array. Alternatively, the application may just retain the output array as an index with which it can access the data set in wanted key order. If the data set was originally already in wanted order, the device driver will inform the application on it, and the application will not need to do any actions on the original data set.

If the key set is very large, sorting could be speeded up with many PBSs. They could be controlled by the same device driver which first spreads the work by giving each PBS their own block to sort, and then merges the output arrays from multiple PBSs before returning the control to calling application. Another alternative to utilize multiple PBSs would be to implement them as separate devices under higher level PBS device controller, just like one device controller may control multiple hard disks. The device driver would now communicate with the PBS device controller, and not with individual PBSs. In this case the device controller could have more intelligence and memory, and it could perform the merge operations for the PBSs that it controls. Either way, each individual PBS should be implemented as large as technology allows, because sorting in PBS has time complexity $O(n)$, whereas merge has time complexity $O(n \log{(n/N)})$.

## 3 SUMMARY

We have proposed to build a very simple device, PBS, which can be used in most circumstances to sort large data sets in linear time with very small factors. PBS should be implemented with as many sorting registers as current technology allows, so that sorting could be done without merging for as large data sets as possible. We have introduced a method on how PBS can be used to sort arbitrary large data sets, or to create indexes to access arbitrary large data sets in wanted order.

## REFERENCES

[1] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. 2008. Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1313–1324. https://doi.org/10.14778/1454159.1454171

[2] A. Greb and G. Zachmann. 2006. GPU-ABiSort: optimal parallel sorting on stream architectures. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium.* 10 pp.–. https://doi.org/10.1109/IPDPS.2006.1639284

[3] Linh Ha, Jens Krüger, and Cláudio T. Silva. [n. d.]. Fast Four-Way Parallel Radix Sorting on GPUs. *Computer Graphics Forum* 28, 8 ([n. d.]), 2368–2378. https://doi.org/10.1111/j.1467-8659.2009.01542.x

[4] Nico Habermann. 1972. *Parallel neighbor-sort (or the glory of the induction principle).* Technical Report. Carnegie-Mellon University. 12 pages. http://repository.cmu.edu/compsci/2087.

[5] Teemu Kerola. 2018. Device, system and method for parallel data sorting. Patent application FI20185206.

[6] Changkyu Kim, Jongsoo Park, Nadathur Satish, Hongrae Lee, Pradeep Dubey, and Jatin Chhugani. 2012. CloudRAMSort: Fast and Efficient Large-scale Distributed RAM Sort on Shared-nothing Cluster. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12).* ACM, New York, NY, USA, 841–850. https://doi.org/10.1145/2213836.2213965

[7] Owen O'Malley and Arun C. Murthy. 2010. Winning a 60 Second Dash with a Yellow Elephant. (December 2010), 9p. http://sortbenchmark.org/Yahoo2009.pdf

[8] N. Satish, M. Harris, and M. Garland. 2009. Designing efficient sorting algorithms for manycore GPUs. In *2009 IEEE International Symposium on Parallel Distributed Processing.* 1–10. https://doi.org/10.1109/IPDPS.2009.5161005