Suitability of Data Flow Computing for Number Sorting

Anton Kos
Faculty of Electrical Engineering, University of Ljubljana, Slovenia
Tržaška 25, 1000 Ljubljana, Slovenia
anton.kos@fe.uni-lj.si

***Abstract*—In this paper we study the suitability of data flow computing for number sorting. We briefly present and discuss the properties of sequential, parallel, and network sorting algorithms. The major part of this study is dedicated to the comparison of the most important network sorting algorithms and to the most used sequential and parallel sorting algorithms. We present the effects of sorting algorithm parallelization and further discuss its impact on sorting algorithms implementation on control flow and data flow computers. The obtained test results clearly show that under certain limitations, when measuring the time needed to sort an array of numbers, data flow computers can greatly outperform control flow computers. By finding solutions to current problems of data flow sorting implementation, important improvements to many applications that need sorting would be possible.**

I. Introduction

Sorting is one of the most important computer operations. Therefore, a constant quest for better sorting algorithms and their practical implementations is necessary. Sorting is also an indispensable part of many applications, often concealed from user. One of many such examples is searching for information on the Internet. Most common, search algorithms work with sorted data and search results are presented as an ordered list of items matching the search criteria [3].

To date most computer systems use well studied comparison based sequential sorting algorithms [2] that have practically reached their theoretical boundaries. Speedups are possible with parallel processing and the use of parallel sorting algorithms.

We can achieve parallelization through the use of *multi-core* or *many-core* systems that can speed up the sorting in the order proportional to the number of cores. Recently a new paradigm called *data flow computing* re-emerged. It offers immense parallelism by utilizing thousands of tiny simple computational elements, improving the performance by orders of magnitude.

The motivation of this paper is to investigate the possibilities of using the data flow computing paradigm for sorting algorithms and their implementation on a data flow computer. We have the possibility to work with the Maxeler MAX2 data flow computer system on which we have carried out all our tests.

II. Sorting Algorithms

Sorting algorithms can be classified on different criteria, such as computational complexity, memory usage, stability, general sorting method, and whether or not they are comparison sorting [5].We will concentrate only on the group of comparison based sorting algorithms. All of the most popular sorting algorithms, as well as network sorting algorithms, are members of this group.

Comparison based sorting algorithms examine the data by repeatedly comparing two elements from the unsorted list with a comparison operator, which defines their order in the final sorted list. In this paper we divide comparison based sorting algorithms into three groups based on the time order of the execution of compare operations: *sequential sorting* algorithms execute the comparison operations in succession, one after another, *parallel sorting* algorithms execute a number of comparison operations at the same time, *network sorting* algorithms are essentially parallel algorithms; they have the property that the sequence of comparison operations is the same for all possible input data.

A particular comparison based sorting algorithm can have one or more versions belonging to one or more of the above listed groups. For instance, *merge sort* can be executed sequentially, it has its parallel version, and it can be implemented as a network sorting algorithm.

A. Sequential Sorting

It has been proven [1] that comparison based sequential sorting algorithms require at least the time proportional to $O(N \cdot \log_2 N)$ on average, where N is the number of items to be sorted. Properties of some of the most used comparison based sorting algorithms are listed in Table I.

TABLE I
PROPERTIES OF THE MOST POPULAR COMPARISON BASED SORTING ALG.

Algorithm	Sorting time – $O(x)$ notation		
	Average	Best	Worst
Insertion	N^2	N	N^2
Selection	N^2	N^2	N^2
Bubble	N^2	N	N^2
Quicksort	$N \cdot \log_2 N$	$N \cdot \log_2 N$	N^2
Merge	$N \cdot \log_2 N$	$N \cdot \log_2 N$	$N \cdot \log_2 N$
Heap	$N \cdot \log_2 N$	$N \cdot \log_2 N$	$N \cdot \log_2 N$

It can be seen that the average, the best, and the worst sorting times vary considerably among algorithms. Especially the best sorting time is heavily dependent on the configuration of input data. For instance, insertion sort has the average and the worst sorting time of $O(N^2)$, but with the nearly sorted input data it needs only $O(N + d)$ operations, where d is the number of needed inversions. On the other hand, quick sort has the average and the best sorting time of $O(N \cdot \log_2 N)$, but in some special cases it has problems with the nearly sorted input data, where it has the worst sorting time of $O(N^2)$ [3].

While on average, the best choice are Quicksort, Merge sort, Heap sort, and Binary tree sort, one would like to avoid Quicksort as its worst sorting time in some rare cases can reach $O(N^2)$. On the other hand, if the configuration of data is expected to be favourable (nearly sorted, for instance), the best choice could be one of the algorithms with sorting time that is linearly proportional to N (Insertion, Bubble, Binary tree, and Shell sort). We see that the choice of the best sorting algorithm is not at all an easy task and depends on the expected input data size and configuration.

B. *Parallel Sorting*

Parallelization of sorting algorithms can be implemented by using multi-core and many-core processors [6]. Generally the term multi-core is used for processors with up to twenty cores and the term many-core for processors with a few tens or even hundreds of cores. In most practical cases this approach is not optimal, as for a true parallel sorting, such a system would need the number of cores in the order of number of items to be sorted (N). In many applications N grows into thousands, millions and more.

Comparison based sorting algorithms are computationally undemanding as the computational operations are simple comparisons between two items. To sort a set of N items, we would need a set of $N/2$ very basic computational cores primarily designed to perform the mathematical operation of comparison. In addition to that, such computational cores would need some control logic in order to execute a specific sort algorithm.

Data flow computing is a good match for parallel sorting algorithms because of its possibility of executing many thousands of operations in parallel, each of them inside a simple computational core. The only limitation is the absence of control over the sorting process in dependence of intermediate results, meaning that the sequence of operations of the sorting process must be defined in advance. This fact prevents the direct use of sorting algorithms from Table I as they are designed for control flow computers; hence they determine the order of item comparisons based on the results of previous comparisons. The possible solution is the adaptation of those sorting algorithms in a way that ensures their conformance to data flow principles. For instance, if we can assure that the parallel sorting algorithm can be modeled as a directed graph, then the sorting process conforms to the data flow paradigm.

C. *Network Sorting*

Network sorting algorithms are parallel sorting algorithms with a fixed structure. Many network sorting algorithms have evolved from the parallel versions of comparison based sorting algorithms and they use the same sorting methods like insertion, selection, merging, etc. Sorting networks structure must form a directed graph, which ensures the output is always sorted, regardless of the configuration of the input data. Because of this constraint, network sorting algorithms that are derived from parallel sorting algorithms will in general perform some redundant operations. This makes them inferior to their originating parallel sorting algorithms in the number of operations (comparisons) that they must perform.

Sorting networks are the implementations of network sorting algorithms and they consist only of comparators and wires. Sorting networks can be described by two properties: the depth and the size. The ***size*** is defined as the total number of comparators it contains. The ***depth*** is defined as the maximum number of comparators along any valid path from any input to any output [2].

By inspecting the properties of network sorting algorithms in Table II, we can conclude that Bubble network sorting is inferior to the others in both properties. While the size of the Bitonic network is larger than the size of Odd-even merge network, its constant number of comparators at each stage can be an advantage in certain applications. If the later is not important, then the best choice would be the use of an Odd-even sorting network.

TABLE II
PROPERTIES OF SOME NETWORK SORTING ALGORITHMS.

Sorting network	Depth	Size
Bubble	$2N-3$	$\frac{N(N-1)}{2}$
Bitonic	$\frac{\log_2 N\cdot(\log_2 N+1)}{2}$	$\frac{N\cdot\log_2 N\cdot(\log_2 N+1)}{4}$
Odd-even merge	$\frac{\log_2 N\cdot(\log_2 N+1)}{2}$	$\frac{N\cdot\log_2 N\cdot(\log_2 N-1)}{4}+N-1$

Assuming that all the comparisons on each level of the sorting network are done in parallel, its depth defines the number of steps needed to sort N numbers on the inputs and thus defines the time needed to sort the input. The size of the sorting network tell us how many comparison is needed, hence how many comparators we need to construct a sorting network. For instance, in hardware implementations the size defines the required chip area.

III. Network Sorting vs. Sequential and Parallel Sorting

Theoretically the number of sequential operations or comparisons for Quicksort sorting algorithm is in the order of $O(N\cdot\log_2 N)$ and for the network version of the Bitonic or Odd-even merge sorting in the order of $O(N(\log_2 N)^2)$ [1]-[4]; i.e. theoretically, Quicksort is better than Bitonic merge algorithm by a factor of $\log_2 N$. This statement is true when we disregard the influence of sorting algorithm constants.

A. *Sorting Algorithm Constants*

Considering the algorithm constants, the number of operations for Quicksort algorithm is in the order of $O(C_Q\cdot N\cdot\log_2 N)$ and for the Bitonic merge algorithm in the order of $O(C_B\cdot N\cdot(\log_2 N)^2)$; what gives us the ratio of $C_B\cdot\log_2 N/C_Q$ or $\log_2 N/\alpha$, where algorithm constants ratio α is defined as $\alpha = C_Q/C_B$. We expect, that for the discussed sorting algorithm pair, $\alpha > 1$.

Network sorting algorithms conform to the data flow paradigm, they have practically no computational overhead (they have no need for process control); therefore the Bitonic merge network sorting has a small constant C_B. On the other hand Quicksort decisions depend heavily on the results of previous operations and hence Quicksort has a large algorithm constant C_Q.

For small N values, where $\log_2 N < \alpha$, Quicksort algorithm should be slower than Bitonic algorithm and for large N values, where $\log_2 N > \alpha$, Quicksort algorithm should become faster. To prove our assumptions we have run a series of tests where we measured the sorting times of the Quicksort algorithm and the network version of the Bitonic merge sorting algorithm. All results for both algorithms, presented in Figure 1, were obtained by sequential computation (no parallelism is employed) on the PC using algorithms written in C code. We can observe, that sorting time curves cross at approximately $N = 128$. Below that number the sequential version of Bitonic network sorting is faster than Quicksort and the opposite above that number.

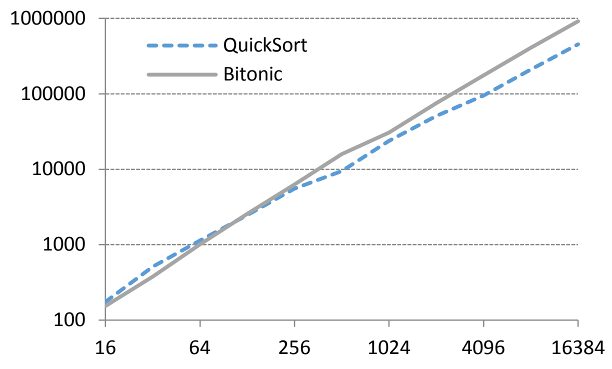

Figure 1. Comparison of sorting times for Quicksort and Bitonic Mergesort network algorithm in dependence on the number of items being sorted (N).

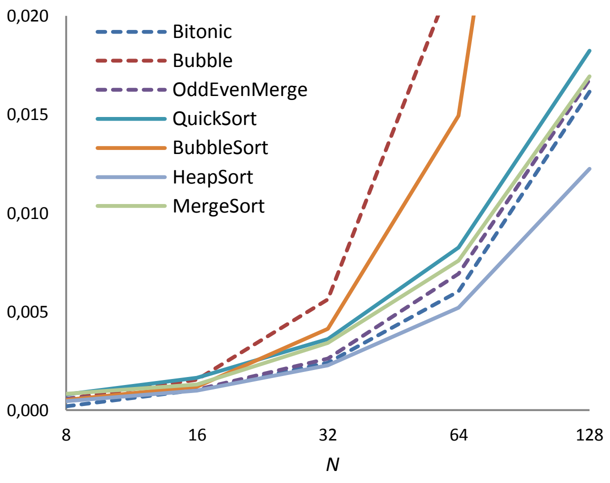

Figure 2. Comparison of the average sorting times between the popular sequential sorting algorithms (solid lines) and network sorting algorithms (dashed lines).

After we have proved, that algorithm constants for network sorting algorithms can be considerably smaller that the constants of sequential sorting algorithms, we have conducted similar tests and comparisons for the most popular sequential sorting algorithms and the most popular network sorting algorithms. The results are shown in Figure 2. Let us emphasize again that all the results for all algorithms are obtained by the sequential computation on a PC using C code. We can see that for the smallest values of N network sorting algorithms outperform any sequential sorting algorithms. When N grows, the higher order of computational complexity of network algorithms prevails over algorithm constants and sequential algorithms become faster.

B. Parallelization

Despite the encouraging results from Figure 2, the following question remains: "Can we expect, that for any larger values of N, network sorting would outperform sequential comparison based sorting?" Even if we exploit parallelism, wouldn't it decrease the computational time by the same factor for all algorithms (parallel execution of sequential algorithms and parallel execution of network algorithms), and the performance ratio would stay the same? The answer lies in the change of computational paradigm and moving to the domain of data flow computing. Let us illustrate that through an example.

TABLE III
PARALLELIZATION OF SORTING ALGORITHMS

	Values for the best algorithm of type (expressions given in $O(x)$ notation)		
Measure	Parallel (N)	Parallel (P)	Network
Comparisons	$N \cdot \log_2 N$	$N \cdot \log_2 N$	$N \cdot (\log_2 N)^2$
Sorting time	$\log_2 N$	$(N/P) \cdot \log_2 N$	$(\log_2 N)^2$

For a true parallel execution of a sorting algorithm we need $O(N)$ computational cores. With that ensured, sorting times for such a parallel algorithm are in the order of $O(\log_2 N)$ for classical algorithms and $O((\log_2 N)^2)$ for network algorithms. Let us assume that the best parallel control flow system has a maximum of P computational cores. Eventually, with the growing N, we will get to the point where $N > P$ and sorting times of classical parallel algorithms will be in the order $O(N \cdot \log_2 N)/P$; again growing faster than linearly and not truly parallel. Since that is not desirable, the sorting should move to the data flow computers that can ensure enough cores for a true parallel execution.

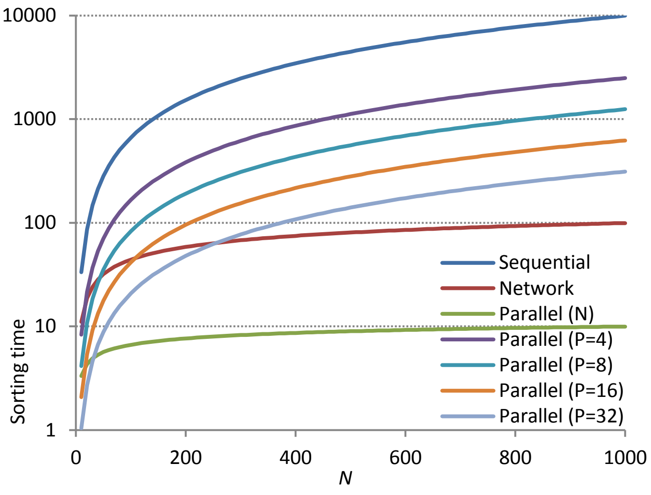

Figure 3. Expected sorting times for the algorithms from Table III. Sorting time is given in cycles (time to do one comparison) needed to sort an array of N items.

Number of comparisons and sorting times for different implementations of sorting algorithms and different degree of parallelization are listed in Table III. For the sequential algorithms the sorting time is proportional to the number of comparisons. With parallel algorithms we execute N (true parallel) or P (near parallel) comparisons at the same time and sorting times are for the corresponding factor smaller. Network algorithms execute all the comparisons of each step in parallel.

In Figure 3 we plot the expected sorting times for the algorithms from Table III without the consideration of the algorithm constant C. The curves show, that when N grows, the true parallel sorting algorithm is superior to all of them, followed by the network sorting algorithm and the near parallel sorting algorithm. The sequential sorting algorithm is the slowest.

C. *Control Flow vs Data Flow Computers*

Based on the results in Figure 3, we can state that in a control flow computer, true parallel sorting algorithm is clearly the first choice. But if we move to a data flow computer, things change considerably. In a data flow computer data flows between operations organized as a directed graph. In the case of network sorting algorithms, the sorting network structure is a directed acyclic graph with comparators organized to sort the input array of values.

When we sort one array, the sorting time is directly proportional to the depth of the sorting network and in each cycle only one layer of comparators is active, the other stay idle. One cycle is defined as time needed to do one comparison step. One layer of comparators represents all comparators of one step of the sorting algorithm. Such a sorting scenario is more suitable for control flow computers. Data flow computers are designed for data flows or data streams, in the case of sorting, that would be a stream of arrays of values to sort.

For instance, if we have M ararys of N values to be sorted, we can send them to the sorting network one after another. Arrays enter the sorting network in one cycle intervals. Similarly, after the first array is sorted on the output, the subsequent arrays exit the sorting network in a one cycle intervals. Each step of the algorithm operates on a different array. When M reaches the depth of the sorting network, all comparators of the network are active. In such scenario the sorting time for the first array is in the order of $O((\log_2 N)^2)$, all the rest follow in one cycle intervals and their sorting time is essentially in the order of $O(1)$.

Comparing the sorting of M ararys of N values on a data flow computer (sorting network) and on a control flow computer (true parallel sorting) gives us interesting results. The sorting time for the control flow computer with the true parallel operation is in the order of $O(M \cdot \log_2 N)$ and for the data flow computer with sorting network in the order of $O((\log_2 N)^2 + M)$. When $M \cong \log_2 N$ both sorting times are comparable, but when $M \gg \log_2 N$, network sorting on a data flow computer becomes much faster.

The conclusion of this consideration is that for small M and small N, the best choice is parallel sorting algorithm on a control flow computer, for large M and small N data flow computer will always perform better, for large N and small M control flow computer will always perform better, when both M and N are large, we can not be conclusive because much depends on the ratio $M/\log_2 N$.

D. *Experimental Results*

To demonstrate the validity of the above conclusions, we have devised a number of tests on a control flow computer (PC) and on a data flow computer (Maxeler MAX2 card). Figure 4 shows the speedup in sorting times. The speedup is the ratio between sorting times on a PC and sorting times on a MAX2 card. We can see that with the growing number of arrays (M), the speedup becomes higher, what confirms our assumptions and we can state, that under certain conditions, data flow computing is suitable for number sorting and outperforms control flow computing.

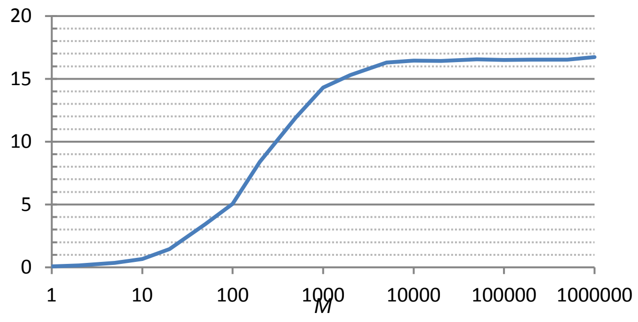

Figure 4. The sorting speedup for arrays of size $N = 128$ in dependence from the stream size M.

Conclusion

Not all algorithms are suitable for data flow computing. In this paper we show that number sorting is suitable for implementation on data flow computers and can, under certain conditions, greatly outperform the control flow computers. There is a lot of work still to be done. One of the main obstacles to date is the small array sizes that can be implemented on data flow computers. We expect that with the advances in data flow computers. By finding solutions to the above and other problems and obstacles, serious improvements to many applications that need sorting, would be possible.

References

[1] Donald E. Knuth, "The art of computer programming. Vol. 3, Sorting and searching", Addison-Wesley, 2002

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to Algorithms, Second Edition", Cambridge (Massachusetts), London, The MIT Press, cop. 2009

[3] Robert Sedgewick, "Algorithms in Java, Third Edition, Parts 1-4", Addison-Wesley, 2010

[4] K.E. Batcher, "Sorting networks and their applications", Proceedings of the AFIPS Spring Joint Computer Conference 32, 307–314, 1968

[5] "Sorting Algorithm", http://en.wikipedia.org/wiki/Sorting_algorithm, accessed 20.1.2014

[6] "Parallel computing", http://en.wikipedia.org/wiki/Parallel_computing, accessed 20.1.2014

[7] "Sorting Network", http://en.wikipedia.org/wiki/Sorting_network, accessed 20.1.2014

[8] Nadathur Satish, Mark Harris, Michael Garland, "Designing Efficient Sorting Algorithms for Manycore GPUs", IEEE, International Symposium on Parallel & Distributed Processing, 2009

[9] http://www.maxeler.com/technology/dataflow-computing, accessed 27.4.2013