An Agglomeration Law for Sorting Networks and its Application in Functional Programming

Lukas Immanuel Schiller
Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
schiller@mathematik.uni-marburg.de

In this paper we will present a general agglomeration law for sorting networks. Agglomeration is a common technique when designing parallel programmes to control the granularity of the computation thereby finding a better fit between the algorithm and the machine on which the algorithm runs. Usually this is done by grouping smaller tasks and computing them en bloc within one parallel process. In the case of sorting networks this could be done by computing bigger parts of the network with one process. The agglomeration law in this paper pursues a different strategy: The input data is grouped and the algorithm is generalised to work on the agglomerated input while the original structure of the algorithm remains. This will result in a new access opportunity to sorting networks well-suited for efficient parallelization on modern multicore computers, computer networks or GPGPU programming. Additionally this enables us to use sorting networks as (parallel or distributed) merging stages for arbitrary sorting algorithms, thereby creating new hybrid sorting algorithms with ease. The expressiveness of functional programming languages helps us to apply this law to systematically constructed sorting networks, leading to efficient and easily adaptable sorting algorithms. An application example is given, using the Eden programming language to show the effectiveness of the law. The implementation is compared with different parallel sorting algorithms by runtime behaviour.

1 Introduction

With the increased presence of parallel hardware the demand for parallel algorithms increases accordingly. Naturally this demand includes sorting algorithms as one of the most interesting tasks of computer science. A particularly interesting class of sorting algorithms for parallelization is the class of *oblivious* algorithms. We will call a parallel algorithm oblivious "iff its communication structure and its communication scheme are the same for all inputs the same size" [15].

Sorting networks are the most important representative of the class of oblivious algorithms. They have been an interesting field of research since their introduction by Batcher [1] in 1968 and are experiencing a renaissance in GPGPU programming [18]. They are based on comparison elements, mapping their inputs $(a_1,a_2) \mapsto (a'_1,a'_2)$ with $a'_1 = \min(a_1,a_2)$ and $a'_2 = \max(a_1,a_2)$ and therefore $a'_1 \leq a'_2$. A simple graphical representation is shown in Figure 1. The arrowhead in the box indicates where the minimum is output.

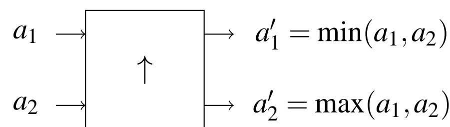


Figure 1: Comparison element (ascending).

S. Schwarz and J. Voigtländer (Eds.): 29th and 30th Workshops on (Constraint) Logic Programming and 24th International Workshop on Functional and (Constraint) Logic Programming (WLP'15/'16/WFLP'16).
EPTCS 234, 2017, pp. 165–179, doi:10.4204/EPTCS.234.12

© Lukas Schiller
This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike License.

A simple functional description of sorting networks results in a repeated application of this comparison element function with fixed indices for every step. For a sequence $(a_1,\ldots,a_n)$ of length n the specific steps are fixed:

$$(a_1,\ldots,a_n) \mapsto \ldots \mapsto (\tilde{a_1},\ldots,a_i,\ldots,a_j,\ldots,\tilde{a_n}) \mapsto (\tilde{a_1},\ldots,a'_i,\ldots,a'_j,\ldots,\tilde{a_n}) \mapsto \ldots \mapsto (a'_1,\ldots,a'_n)$$

with $i \neq j$. In a specific step a_i and a_j are sorted with a comparison element. Ultimatly resulting in the sorted sequence $(a'_1,\ldots,a'_n)$.

Figure 2 shows a simple sorting network for lists of length 4. For every permutation of the input $(a_1,\ldots,a_4)$ the output $(a'_1,\ldots,a'_4)$ is sorted – the comparisons are independent of the data base. Notice the obvious inherent parallelism in the first two steps of the sorting network. The restriction to a fixed structure of comparisons results in an easily predictable behaviour and easily detectable parallelism.

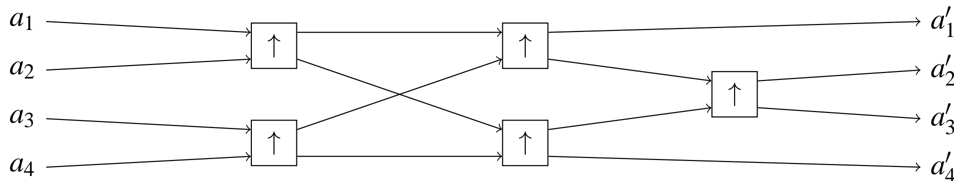


Figure 2: Simple sorting network with comparison elements. Source: [11]

Some well-known sorting algorithms, for example Bubble Sort [11], can be described as sorting networks. Especially in the case of recursively constructed sorting networks (e.g. Batcher's Bitonic Sort or Batcher's Odd-Even-Mergesort), with their inherent functional structure, an obviously correct description of the algorithm is easily possible in a functional programming language such as Haskell [16].

In practice straightforward implementations of these algorithms often struggle with too fine a granularity of computation and therefore do not scale well. Agglomerating parts of the algorithm is a common step in dealing with this problem when designing parallel programmes (compare Foster's PCAM method [7]). With recursive algorithms for example it is a common technique to agglomerate branches of the recursive tree by parallelising only until a specific depth of recursion. With a coarser granularity the computation to communication ratio improves. A common agglomeration for sorting networks is to place blocks or rows of comparison elements in one parallel process.

In this paper we discuss a different approach. We will agglomerate the input data and alter the comparison element to work on blocks of data. This approach is not based upon the structure of a specific sorting network and can therefore be applied to any sorting network. At the same time we will see that the limited nature of sorting networks is necessary for this modification to be correct. The application of this transformation will open a different access to sorting networks, allowing easy combination with other sorting algorithms. Working on data structures instead of single elements leads to a suitable implementation for modern multi-core computers, GPGPU concepts or computer networks. We will obtain an adequate granularity of computation and the width of the sorting network can correspond with the number of processor units. A second layer of traditional agglomeration (e.g. blocks or rows of comparison elements) is independently possible.

In Section 2 we will discuss which demands are necessary for altered comparison elements to preserve an algorithm's functionality and correctness. In Section 3 an example is given showing situations in which the application of this agglomeration is beneficial and tests with different approaches are evaluated. Section 4 discusses related work and Section 5 concludes.

2 Agglomeration Law for Sorting Networks

In general, sorting networks work on sequences of elements $A = (a_1, \ldots, a_n)$. Our improvement will work with a partition of a given sequence. In the following, we will use Haskell notation and lists instead of sequences to improve readability, even though a more general type would be possible.

Theorem 1 (Agglomeration Law for Sorting Networks)**.** *Let* $\mathtt{A} = [\mathtt{a_1}, \ldots, \mathtt{a_n}]$ *be a sequence where a total order "*$\leq$*" is defined on the elements,* $\mathtt{c} :: \mathtt{Ord\ a} \Rightarrow (\mathtt{a},\mathtt{a}) \rightarrow (\mathtt{a},\mathtt{a})$ *a comparison element as described before and*

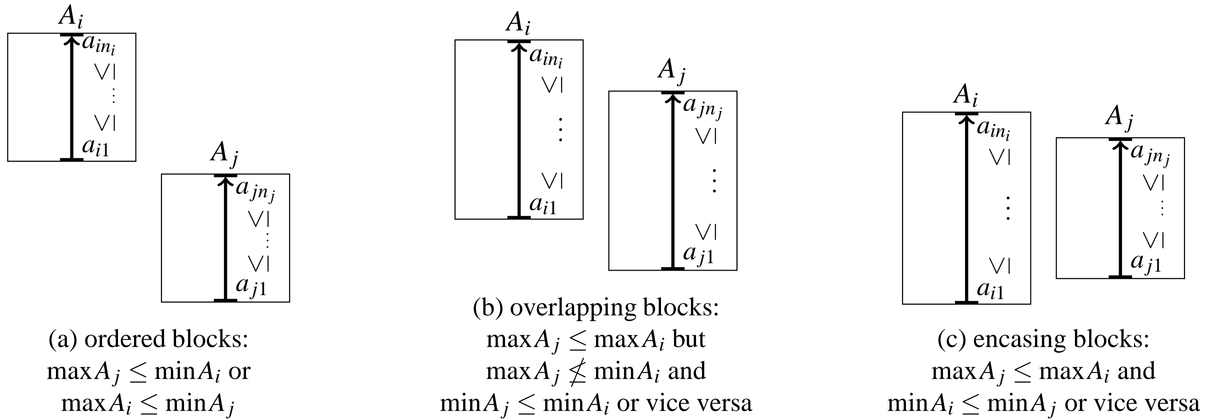
$$sN :: ((b,b) \rightarrow (b,b)) \rightarrow [b] \rightarrow [b]$$

a correct sorting network, meaning $\mathtt{sN\ c\ A} = \mathtt{A}'$ *with* $\mathtt{A}' = [\mathtt{a}'_1, \ldots, \mathtt{a}'_\mathtt{n}]$ *and* $\mathtt{a}'_1 \leq \ldots \leq \mathtt{a}'_\mathtt{n}$ *where* $\mathtt{a}'_1, \ldots, \mathtt{a}'_\mathtt{n}$ *is a permutation of* $\mathtt{a_1}, \ldots, \mathtt{a_n}$ *and the only operation used by the sorting network is a repeated application of the comparison element with a fixed, data independent structure for a given input size. Then there exists a comparison element* $c' :: Ord\ a \Rightarrow ([a],[a]) \rightarrow ([a],[a])$ *with which a sequence of sequences* $\mathfrak{A} = [A_1, \ldots, A_n]$ *with* $A_i = [a_{i1}, \ldots, a_{in_i}]$ *can be sorted with the same sorting network. Meaning that* $\mathrm{sN}\ \mathtt{c}'\ \mathfrak{A} = \mathfrak{A}'$ *with* $\mathfrak{A}' = [\mathtt{A}'_1, \ldots, \mathtt{A}'_\mathtt{n}]$ *and* $\mathtt{A}'_1 \preceq \ldots \preceq \mathtt{A}'_\mathtt{n}$*. Where* concat $\mathfrak{A}'$ *is a permutation of* concat $\mathfrak{A}$ *and* $A \preceq B$ *means that for two sequences* $A = [a_1, \ldots, a_p]$ *and* $B = [b_1, \ldots, b_q]$ *every element of A is less than or equal to every element from B:*

$$A \preceq B \Leftrightarrow \forall a \in A, \forall b \in B : a \leq b$$

With blocks of data the concatenation of the elements of $\mathfrak{A}'$ need to be a permutation of the concatenation of $\mathfrak{A}$, the elements themselves $(\mathtt{A}'_1, \ldots, \mathtt{A}'_\mathtt{n})$ do not need to be a permutation of $\mathtt{A_1}, \ldots, \mathtt{A_n}$.

Note that the order relation for blocks of data "$\preceq$" defines only a partial order whereas the elements inside the blocks are totally ordered. To this end we need to specialise the comparison element to deal with the case of overlapping or encasing blocks and still fulfill all properties necessary for the sorting network to work correctly (cf. Figure 3).



(a) ordered blocks: $\max A_j \leq \min A_i$ or $\max A_i \leq \min A_j$

(b) overlapping blocks: $\max A_j \leq \max A_i$ but $\max A_j \nleq \min A_i$ and $\min A_j \leq \min A_i$ or vice versa

(c) encasing blocks: $\max A_j \leq \max A_i$ and $\min A_i \leq \min A_j$ or vice versa

Figure 3: Cases for comparison elements for blocks of data: blocks can be ordered (with order relation $\preceq$), overlapping or encasing, where overlapping and encasing means that the blocks are not in an order relation between one another (meaning neither $\preceq$ nor $\succeq$ holds).

If for example the input lists overlap (e.g. $\mathtt{c}'\ ([1,2,3,4],[3,4,5,6])$) a simple swap would not fulfill the requirements. We would prefer a result such as $([1,2,3,3],[4,4,5,6])$ and therefore $\mathfrak{A}'$ can not be a permutation of $\mathfrak{A}$ but we expect that every element $\mathtt{a_{ij}}$ from $\mathtt{A_1}, \ldots, \mathtt{A_n}$ is in $\mathtt{A}'_1, \ldots, \mathtt{A}'_\mathtt{n}$. In the next step we will investigate which conditions a comparison element for blocks of data must fulfill.

2.1 Comparison element for partially ordered blocks of totally ordered elements

If we want to alter the comparison element while preserving the functionality and correctness of the sorting network we must understand which information is generated and preserved within a traditional comparison element. We will therefore investigate the capabilities and limits of comparison elements for totally ordered sequences: Let $a_1, a_2, a_1^1, a_1^2, a_2^1, a_2^2$ be elements where information about the following relations have already been gathered by the sorting network:

$$a_1^1 \le a_1 \le a_1^2 \qquad \text{and} \qquad a_2^1 \le a_2 \le a_2^2 \tag{0}$$

If we do sort a_1 and a_2 with a comparison element $(a_1, a_2) \mapsto (a_1', a_2')$ we receive new relations (e.g. $a_1^1 \le a_1 \Rightarrow a_1^1 \le a_2'$). We will distinguish between *direct relations* and *conditional relations*. In this case direct relations refer to all relations resulting directly from the relations which exists and are known before the application of the comparison element and which involve a_1, a_2, a_1' or a_2'. We expect the comparison element to be side-effect free and therefore we expect every relation between elements not touched by the comparison element to be unaffected by its application. Here the direct relations resulting from (0) are:

$$a_1' \le a_2' \tag{1}$$
$$a_1' \le a_i^2,\ i \in \{1,2\} \tag{2}$$
$$a_i^1 \le a_2',\ i \in \{1,2\} \tag{3}$$

If we have additional information, we get additional direct relations. For $\{i, j\} = \{1,2\}$:

$$a_i^1 \le a_j \Rightarrow a_i^1 \le a_1' \tag{4}$$
$$a_j \le a_i^2 \Rightarrow a_2' \le a_i^2 \tag{5}$$
$$a_i \le a_j \Rightarrow a_i^1 \le a_1' \wedge a_2' \le a_j^2 \tag{6}$$

We call these relations *direct relations* only if the left side is already known.

Definition 1 (Valid comparison elements for blocks of data)**.** Let $\mathtt{A}_1, \mathtt{A}_2$ be sequences with a total order "$\le$" defined on the elements and $\mathtt{c}' :: \mathtt{Ord\ a} \Rightarrow ([\mathtt{a}],[\mathtt{a}]) \rightarrow ([\mathtt{a}],[\mathtt{a}])$ a block comparison element with $\mathtt{c}'(\mathtt{A}_1, \mathtt{A}_2) = (\mathtt{A}_1', \mathtt{A}_2')$ and $A_1' \preceq A_2'$.

$\mathtt{c}'$ is called *valid*, iff all elements from A_1 and A_2 which are less than $\mathtt{lb} = \max(\min(\mathtt{A}_1), \min(\mathtt{A}_2))$ must be in A_1', all elements which are greater than $\mathtt{ub} = \min(\max(\mathtt{A}_1), \max(\mathtt{A}_2))$ must be in A_2' and all elements between these limits can be either in A_1' or in A_2' as long as every element in A_1' is smaller than or equal to every element in A_2' (cf. Figure 4).

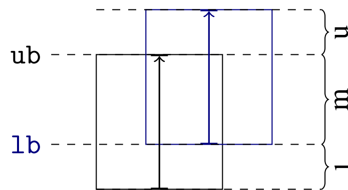


Figure 4: Sections of the comparison element for blocks of data. Elements from l must be in the lesser result (A_1'), elements from u must be in the greater result (A_2') and elements from m can be in either result as long as $A_1' \preceq A_2'$ holds.

Lemma 1. Valid *block comparison elements maintain the* direct relations *fulfilled by the elementary comparison elements.*

Proof. We show the validity of the above relations (1) to (6) for blocks of data:

1. $\mathtt{A}_1' \preceq \mathtt{A}_2'$ is included in the definition.
2. $\max \mathtt{A}_1' \leq \mathtt{ub} \leq \max \mathtt{A_i} \leq \min \mathtt{A}_\mathtt{i}^2 \Rightarrow \mathtt{A}_1' \preceq \mathtt{A}_\mathtt{i}^2,\ \mathtt{i} \in \{1,2\}$
3. $\max \mathtt{A}_\mathtt{i}^1 \leq \min \mathtt{A_i} \leq \mathtt{lb} \leq \min \mathtt{A}_2' \Rightarrow \mathtt{A}_\mathtt{i}^1 \preceq \mathtt{A}_2',\ \mathtt{i} \in \{1,2\}$
4. $\mathtt{A}_\mathtt{i}^1 \preceq \mathtt{A}_1 \wedge \mathtt{A}_\mathtt{i}^1 \preceq \mathtt{A}_2 \Rightarrow \mathtt{A}_\mathtt{i}^1 \preceq [\min(\min \mathtt{A}_1,\ \min \mathtt{A}_2)] \preceq \mathtt{A}_1' \Rightarrow \mathtt{A}_\mathtt{i}^1 \preceq \mathtt{A}_1',\ \mathtt{i} \in \{1,2\}$
5. $\mathtt{A}_1 \preceq \mathtt{A}_\mathtt{i}^2 \wedge \mathtt{A}_2 \preceq \mathtt{A}_\mathtt{i}^2 \Rightarrow \mathtt{A}_2' \preceq [\max(\max \mathtt{A}_1,\ \max \mathtt{A}_2)] \preceq \mathtt{A}_\mathtt{i}^2 \Rightarrow \mathtt{A}_2' \preceq \mathtt{A}_\mathtt{i}^2,\ \mathtt{i} \in \{1,2\}$
6. $\mathtt{A}_\mathtt{i}^1 \preceq \mathtt{A_i} \preceq \mathtt{A_j} \Rightarrow \mathtt{A}_\mathtt{i}^1 \preceq \mathtt{A_i} \wedge \mathtt{A}_\mathtt{i}^1 \preceq \mathtt{A_j} \Rightarrow \max \mathtt{A}_\mathtt{i}^1 \leq \min(\min \mathtt{A_i},\ \min \mathtt{A_j}) \Rightarrow \mathtt{A}_\mathtt{i}^1 \preceq \mathtt{A}_1'$
   $\mathtt{A_i} \preceq \mathtt{A_j} \preceq \mathtt{A}_\mathtt{j}^2 \Rightarrow \mathtt{A_i} \preceq \mathtt{A}_\mathtt{j}^2 \wedge \mathtt{A_j} \preceq \mathtt{A}_\mathtt{j}^2 \Rightarrow \max(\max \mathtt{A_i},\ \max \mathtt{A_j}) \leq \min \mathtt{A}_\mathtt{j}^2 \Rightarrow \mathtt{A}_\mathtt{j}' \preceq \mathtt{A}_\mathtt{j}^2$

□

The proof shows that these limits are not only sufficient but necessary to guarantee the *direct relations* on which sorting networks are essentially based. Counterexamples where a different limit selection leads to the failure of the sorting network can easily be found.

All other producible information concerns *conditional relations* which depend on a yet unknown condition resulting in a disjunction or a conditional with unknown antecedent. For example

$$(a_1 \leq a_2 \vee a_2 \leq a_1) \wedge a_1^1 \leq a_1 \leq a_1^2 \Rightarrow a_1^1 \leq a_1' \vee a_2' \leq a_1^2$$

For ordered or overlapping blocks we can easily verify that all these relations can be preserved, as every input element has a direct descendant, analogous to the original comparison element. In this case a *direct descendant* $\mathtt{A}'$ of a block $\mathtt{A}$ is bounded by the extrema of the parental block, meaning that $\min \mathtt{A} \leq \min \mathtt{A}'$ and $\max \mathtt{A}' \leq \max \mathtt{A}$. $\mathtt{A}'$ can but need not contain elements from $\mathtt{A}$ as well as elements which are not in $\mathtt{A}$ due to the fact that the property is defined through boundaries not elements. Therefore, when applying the comparison element, the boundaries of each block can at the most approach each other, leaving all relations preserved. An example is given in Figure 5.

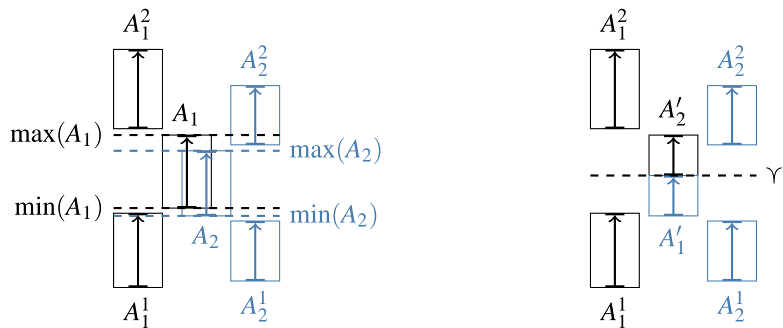


Figure 5: Split of overlapping blocks. In this case the minimal (maximal) element of A_2 is smaller than the minimal (maximal) element of A_1. Thereby A_2 "shrinks" from above, meaning that the maximum element of A_1' is smaller than $\max A_2$. This does not yet disclose any information about the number of elements in A_1'. A_1 "shrinks" from below. We can see A_1' as the descendant of A_2 and A_2' as the descendant of A_1. All relations are preserved.

With encased blocks (cf. Figure 3c) it is not necessarily possible to find a descendant for every element. If, for example, we have $A_1^1 \preceq A_1 \preceq A_1^2$ and $A_2^1 \preceq A_2 \preceq A_2^2$ there might be no output element A_i' with $A_1^1 \preceq A_i' \preceq A_1^2$ (cf. Figure 6).

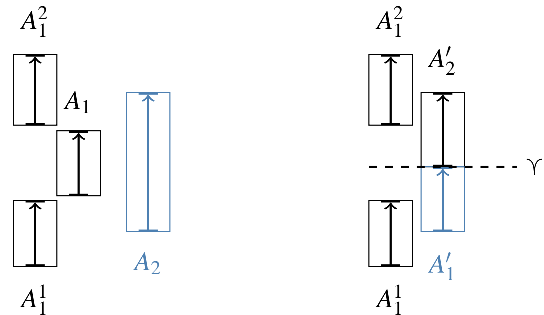


Figure 6: Split of encased blocks. There are no direct descendants. $A_1' \preceq A_1^2$ and $A_1^1 \preceq A_2'$ but neither A_1' nor A_2' is between A_1^1 and A_1^2.

Unfortunately, a consequence of this is that this technique of merging and splitting blocks can not necessarily be transferred to a more general sorting algorithm. In particular this does not work with pivot based sorting algorithms. However it does with sorting networks because the comparison element does not compare one fixed element with another element but rather returns two sorted elements for which we do not know which input element is mapped to which output element. The information $A_1^1 \prec A_1$ is reduced to $A_1^1 \preceq A_2'$ plus some *conditional* information. Some of this conditional information can no longer be guaranteed to hold but can not be used in a sorting network at all because of the limited operations of sorting networks. The relations of concern are

$$a_1 \leq a_2 \vee a_2 \leq a_1 \Rightarrow a_i^1 \leq a_1' \vee a_2' \leq a_i^2,\ i \in \{1,2\} \tag{7}$$

Sorting networks as described above can not produce the additional information needed for this conditional information to become useful.

Lemma 2. *Information about the* conditional relations *(7) that can not be preserved by the altered comparison element* c' *can not be used by a sorting network.*

Sketch of Proof. The condition of a *conditional relation* is unknown by definition – otherwise it would be a direct relation. Therefore the implication can not be used to gather additional information. The remaining disjunction can result in useful information in a non-trivial way only if one side of the disjunction is known to be false (modus tollendo ponens) or if both sides of the disjunction are equal. It is not possible to equalise an output element of the comparison element with another element and therefore it is not possible to test whether $a_i \leq a_j$ or not. In particular the information that $a_i \not\leq a_j$ can not be produced for any i and j. We can not test whether one side of such an equation is false or if both sides are equal and therefore can not use *conditional relations*. Non-trivial, productive information from these disjunctions can only be used in non-oblivious algorithms.

Lemma 1 and Lemma 2 imply that a comparison element c' as demanded in Theorem 1 exists with the given limitations from Lemma 1. Therefore all usable information is preserved and this technique of merging and splitting two blocks in a comparison element can be used with every sorting network.

If the elements inside the blocks are sorted, we can define a linear time comparison element that splits the two blocks into blocks as equal in size as possible. An implementation of such a comparison element

can be found in Section 3, Listing 5. Balancing the blocks is advantageous in many cases because it limits the maximal block size to the size of the largest block in the initial sequence. This is beneficial especially in the situation of limited memory for different parts of the parallelised algorithm, for example if the parallelization is done with a computer cluster. By preserving the inner sorting of the blocks, the resulting sequence of the sorting network can be easily combined to a completely sorted sequence by concatenation.

Every suitable sorting algorithm can be used for the initial sorting inside the blocks. Consequently the sorting network can be used as a skeleton to parallelise arbitrary sorting algorithms and work as the merging stage of the newly combined (parallel) algorithm. A concept that will prove it's worth in the following example.

3 Application of the Agglomeration Law on the Bitonic Sorter

We will now apply the agglomeration law to Batcher's Bitonic Sorting Network. It is a recursively constructed sorting network that works in two steps. In the first step an unsorted sequence (of length 2^l with $l \in \mathbb{N}$) is transformed into a bitonic sequence. A bitonic sequence is the juxtaposition of an ascending and a descending monotonic sequence or the cyclic rotation of the first case (Figure 7).

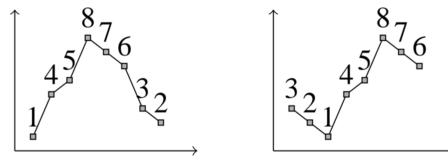


Figure 7: Examples of bitonic sequences.

The bitonic sequence is thereafter sorted by a *Bitonic Merger*. We will call the function implementing this Bitonic Merger `bMerge` and the function transforming an unsorted sequence into a bitonic sequence `prodBList`. The Bitonic Sorter works with the nested divide-and-conquer scheme of the sorting-by-merging idea. This means that the repeated generation of shorter sorted lists is done by Bitonic Sorters of smaller size. A Bitonic Sorter for eight input elements is depicted in Figure 8.

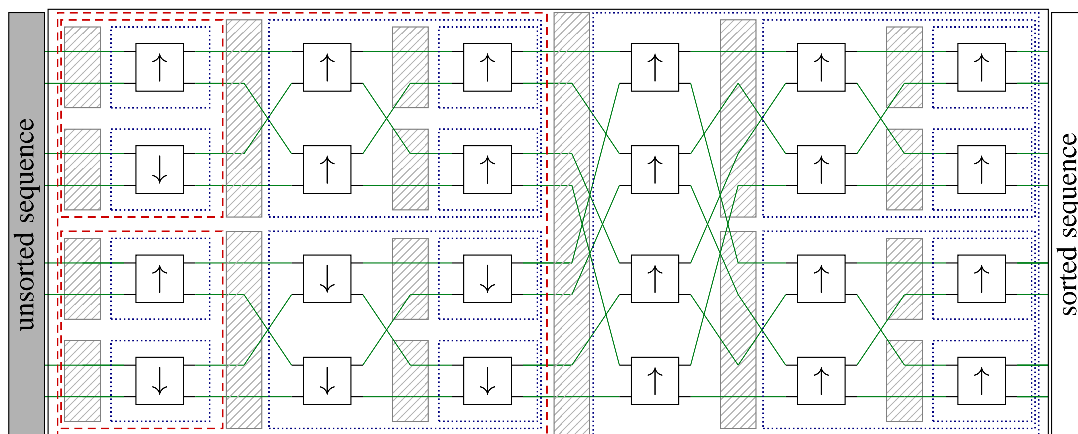


Figure 8: Bitonic Sorter of order 8. The function `prodBList` is represented by a red dashed rectangle, the function `bMerge` by a blue dotted one. Bitonic Sequences are represented by shaded rectangles.

The basic component of the sorting network – the original comparison element – can be defined as:

Listing 1: Original comparison element

```
data Direction = Up | Down deriving Show

compElem :: Ord a ⇒ Direction → [a] → [a]
compElem Up [x,y] = if x ≤ y then [x,y] else [y,x]
compElem Down xs = reverse $ compElem Up xs
```

We will use a two-element-list variant instead of pairs for reasons of code elegance. We define the actual algorithm using the Eden [14, 13] programming language which extends Haskell by the concept of parallel *processes* with an implicit communication as well as a Remote Data [5] concept. We can instantiate a process that is defined by a given function with `($#)`:

```
($#) :: (Trans b, Trans a) ⇒
        (a → b)      -- Process function
        → a → b      -- Process input and output
```

The class `Trans` consists of *transmissible* values. The expression `f $# expr` with some function `f :: a → b` will create a (remote) child process. The expression `expr` will be evaluated (concurrently by a new thread) in the parent process and the result `val` will be sent to the child process. The child process will evaluate `f $ val` (cf. Figure 9).

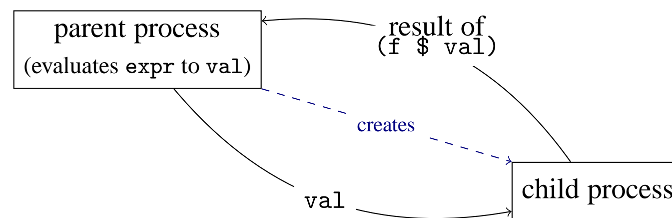


Figure 9: The scheme for process instantiation. Source: [13]

Hereafter we will essentially use Eden's `parMapAt`, a parallel variant of `map` with explicit placement of processes on processor elements (PEs), also called (logical) machines, which are numbered from 1 to the number of processor elements.

```
parMapAt :: (Trans a, Trans b) ⇒
               [Int]       -- ^places for instantiation
            → (a → b)     -- ^worker function
            → [a] → [b] -- ^task list and ^result list
```

The explicit placement is determined by the first argument, a list of PE numbers specifying the places where the processes will be deployed. Additionally we will use the constants `noPe` and `selfPe` provided by Eden to calculate the correct placements:

```
noPe   :: Int -- Number of (logical) machines in the system
selfPe :: Int -- Local machine number (ranges from 1 to noPe)
```

For our implementation we will place each comparison element of the same row on the same PE. In Listing 2 a parallel definition of the algorithm is given.

Listing 2: Parallel bSort

```
36 bSort :: Trans a
37      ⇒ (Direction → [a] → [a]) -- ^specialized comparison element
38      → Direction                -- ^sorting direction
39      → [a] → [a]                -- ^input and ^output
40 bSort _          _ [ ] = [ ]
41 bSort _          _ [x] = [x]
42 bSort sCompElem d xss = (bMerge sCompElem d) ∘ prodBList $ xss where
43     prodBList = unSplit ∘ pMap bSort' ∘ zip [Up, Down] ∘ splitHalf
44     bSort' =  uncurry (bSort sCompElem)
45     pMap = parMapAt [selfPe, selfPe+hcc]
46     hcc = (length xss) `div` 4 {- half comparator count -}
```

The bSort function takes three arguments: an oriented comparison element, a Direction denoting whether the result should be sorted in an ascending or descending order and an input list. The main part of the algorithm is a composition of the prodBList and the bMerge function (cf. Line 42 in Listing 2). The prodBList function splits the input list and sorts both parts with the Bitonic Sorter, one half ascending and one half descending (cf. Line 43). It uses two helper functions splitHalf and unSplit. With the help of Eden's splitIntoN, which splits the input list blockwise into as many parts as the first parameter determines, we define:

```
splitHalf :: [a] → [[a]]
splitHalf = splitIntoN 2
```

Both resulting lists are of the same size because the width of the Bitonic Sorter and therefore its input list's length are powers of two (not to be confused with the size of the blocks which can be of arbitrary size). The needed reverse function – unSplit – can be defined as:

```
unSplit :: [[a]] → [a]
unSplit = concat
```

The correct placement by line is calculated depending on the width of the sorting network (cf. Line 46). Two elements are needed for every comparison element, therefore hcc is half the size of the sorting network in the actual recursion step. The bMerge function does have the same type signature as the bSort function but the input list must be a bitonic list for the function to work correctly:

Listing 3: Parallel bMerge

```
49 bMerge :: Trans a
50      ⇒ (Direction → [a] → [a]) -- ^specialized comparison element
51      → Direction                -- ^sorting direction
52      → [a] → [a]                -- ^input and ^output
53 bMerge sCompElem d xss@[x,y] = sCompElem d xss
54 bMerge sCompElem d xss = unSplit ∘ pMap (bMerge sCompElem d) ∘ bSplit $ xss where
55     bSplit = splitHalf ∘ shuffle ∘ pMap' (sCompElem d) ∘ perfectShuffle
56     pMap = parMapAt [selfPe, selfPe+hcc]
57     hcc = (length xss) `div` 4 {- half comparator count -}
58     pMap' = parMapAt [selfPe..]
```

The main part of the bMerge function is the function bSplit which splits a bitonic sequence into two bitonic sequences with an order between each other. This function uses a communication structure referred to as a *perfect shuffle*[1] by Stone [21]. With this communication scheme the element i and $i + \frac{p}{2}$ are compared, resulting in a split depicted in Figure 10.

[1]This structure can be found in various algorithms, e.g. in the Fast Fourier transform or in matrix transpositions.

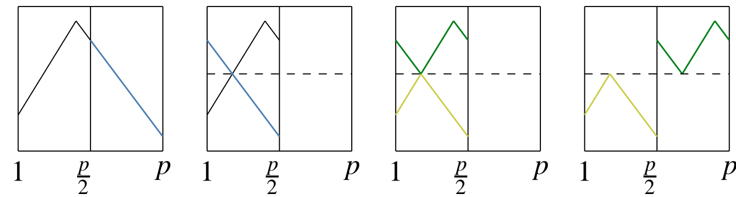


Figure 10: Concept of splitting a bitonic sequence.

In Eden this *perfect shuffle* is easily defined with the help of the offered auxiliary functions:

```
-- Round robin distribution and inverse function called shuffle
unshuffle :: Int → [a] → [[a]]
shuffle :: [[a]] → [a]
```

The first parameter of `unshuffle` specifies the number of sublists in which the list is split, e.g.:

```
unshuffle 3 [1..10] = [[1,4,7,10],[2,5,8],[3,6,9]]
shuffle [[1,4,7,10],[2,5,8],[3,6,9]] = [1..10]
```

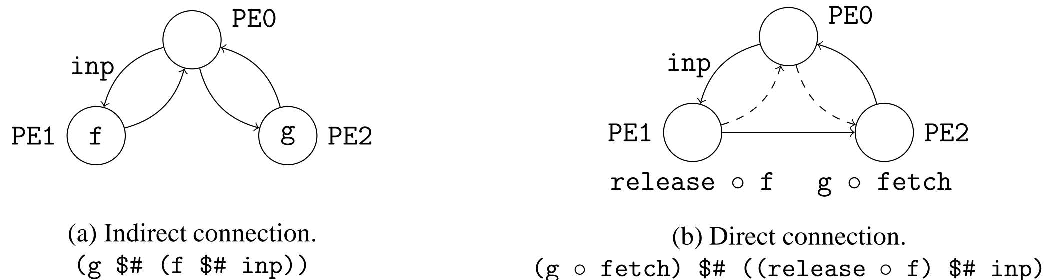
The *perfect shuffle* is then defined as:

```
perfectShuffle :: [a] → [[a]]
perfectShuffle xs = unshuffle halfSize xs
                    where halfSize = (length xs) `div` 2
```

A direct communication between consecutive comparison elements can be realised with Eden's Remote Data concept in which a smaller handle is transmitted instead of the actual data. The data itself is fetched directly when needed from the PE where the handle was created. The more intermediate steps are involved, the more effective the benefits of this concept become. This can be done by the operations:

```
type RD a -- remote data
-- converts local data into corresponding remote data and vice versa
release :: Trans a ⇒ a → RD a
fetch :: Trans a ⇒ RD a → a
releaseAll :: Trans a ⇒ [a] → [RD a] -- list variants
fetchAll :: Trans a ⇒ [RD a] → [a]
```

In Figure 11 the communication scheme of a Remote Data connection is pictured.



(a) Indirect connection.
`(g $# (f $# inp))`

(b) Direct connection.
`(g ∘ fetch) $# ((release ∘ f) $# inp)`

Figure 11: Remote Data scheme. Source: [13]. The processes computing the results of `f` and `g` are placed on two different `PE`s. Without RD, the result of `f` is transferred via the parental process. With RD a handle is generated on `PE1` and transferred via `PE0` to `PE2`, the actual result is transferred directly.

If we call the `bSort` function with the original comparison element and the organization of the communication via Remote Data we receive a correct implementation of the Bitonic Sorter:

Listing 4: Parallel variant of the original Bitonic Sorter

```
88  bitonicSort "simple" = unwrap ∘ bSort sCompElem Up ∘ wrap where
89      wrap = releaseAll
90      unwrap = fetchAll
91      sCompElem :: (Trans a, Ord a) ⇒ Direction → [RD a] → [RD a]
92      sCompElem d = releaseAll ∘ compElem d ∘ fetchAll
```

To apply the agglomeration law we can change the comparison element to a suitable comparison element for blocks. In Listing 5 a simple implementation is given:

Listing 5: A simple MergeSplit function working as a comparison element for blocks of data

```
70  simpleMergeSplit :: Ord a ⇒ [a] → [a] → ([a],[a])
71  simpleMergeSplit [] a2 = ([], a2)
72  simpleMergeSplit a1 [] = (a1, [])
73  simpleMergeSplit a1 a2 = (s,b) where
74      ag = Ordered.merge a1 a2 -- merge from Data.List.Ordered
75      lb = max (minimum a1) (minimum a2)
76      ub = min (maximum a1) (maximum a2)
77      l = [x | x ← ag, x < lb]
78      m = [x | x ← ag, x ≥ lb, x ≤ ub]
79      u = [x | x ← ag, x > ub]
80      (m1,m2) = balancingSplit (length u - length l) m
81      s = l ++ m1
82      b = m2 ++ u
83  balancingSplit :: Int → [a] → ([a],[a])
84  balancingSplit d xs = splitAt lh xs where
85      lh = div ((length xs)+d) 2
```

In Listing 6 an optimised implementation is given. It uses unboxed vectors to optimise transmissions. For reasons of comparability, we use the list variant of the altered comparison element and the merge sort from `Data.List`.

Listing 6: Combination of mergesort with the Bitonic Sorter

```
100 bitonicSort "block" = unwrap ∘ bSort sCompElem Up ∘ preSort ∘ wrap where
101   wrap = releaseAll ∘ map V.fromList ∘ splitIntoN p
102   unwrap = concat ∘ map V.toList ∘ fetchAll
103   p = noPe * 2 -- two input lists per row, one row for every PE
104   places = 1 : 1 : map (1+) places
105
106   preSort :: (V.Unbox a, Ord a) ⇒ [RD (V.Vector a)] → [RD (V.Vector a)]
107   preSort = parMapAt places sSort where
108       sSort = release ∘ V.fromList ∘ sort ∘ V.toList ∘ fetch
109
110   sCompElem :: (V.Unbox a, Trans a, Ord a)
111               ⇒ Direction → [RD (V.Vector a)] → [RD (V.Vector a)]
112   sCompElem d = releaseAll ∘ map V.fromList ∘ compElemB d ∘ map V.toList ∘ fetchAll
113
114   compElemB :: Ord a ⇒ Direction → [[a]] → [[a]]
115   compElemB Up [xs,ys] = (λ(x,y) → [x,y]) $ simpleMergeSplit xs ys
116   compElemB Down xss = reverse $ compElemB Up xss
```

This simple adaption results in a hybrid sorting algorithm parallelising merge sort with the Bitonic Sorter.

3.1 Runtime and Speedup Evaluation

We tested the algorithms on a multicore computer called Hex and on the Beowulf Cluster[2] in order to compare two different implementations of Eden: with MPI [6] as a middleware on the Beowulf cluster and an optimised implementation on the multicore computer. First we will compare the above-mentioned parallelization of merge sort using the Bitonic Sorter to another parallelization of the same merge sort using the disDC divide-and-conquer skeleton from Eden's skeleton library. Both variants are implemented in Eden and equipped with similar improvements. We will work on lists in particular since they are the common choice of data structure in Haskell but use unboxed vectors for transmissions. In Figure 12 the runtime graphs of the parallel disDC merge sort and the Bitonic Sorter are depicted.

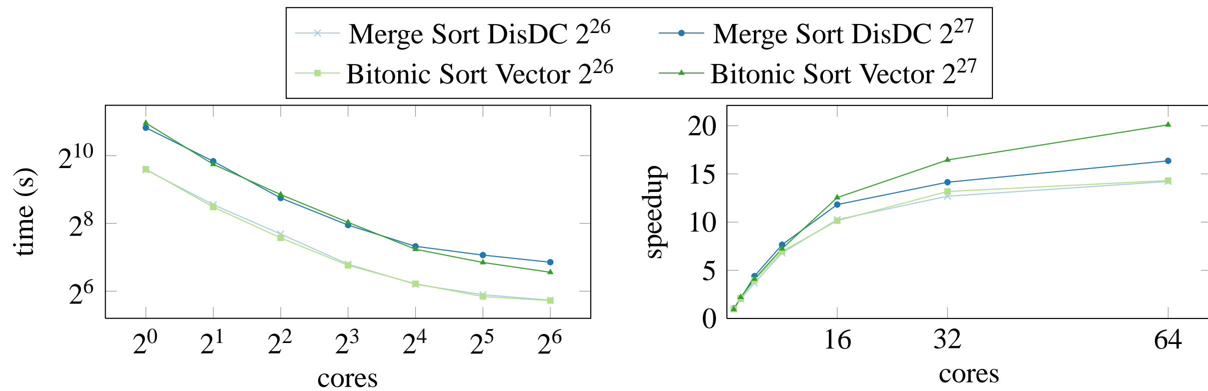


Figure 12: Runtime and Speedup of the Bitonic Sorter and merge sort on Hex with 2^{26} and 2^{27} elements.

The graphs indicate that although the respective runtimes are fairly similar, the Bitonic Sorter variant scales better for larger inputs. The assumption can be hardened by the examination of the corresponding (absolute) speedups. The better scalability of the Bitonic Sorter can partly be explained by the merging, consisting of many small steps with comparison elements. This concept of merging can benefit from a great number of `PEs`. A discovery that can also be made on the Beowulf Cluster though it is notable that here the perceived characteristics are even more pronounced (cf. Figure 13).

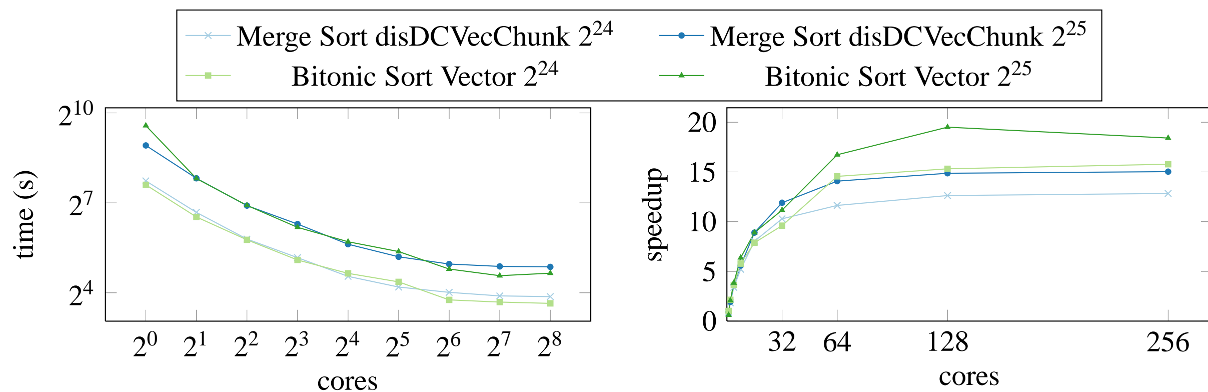


Figure 13: Bitonic Sorter / merge sort hybrid compared to a traditional merge sort on the Beowulf Cluster.

[2]Hex is equipped with an `AMD Opteron CPU 6378` (64 cores) and 64 GB memory, the Beowulf cluster at the Heriot-Watt University Edinburgh consists of 32 nodes, each one equipped with an `Intel Xeon E5504` CPU (8 Cores) and 12 GB memory.

On the Beowulf Cluster the communication between different PEs located on the same computer is cheap while intercommunication between computers is comparatively slow. The local communication structure of the bitonic sorting network is well-suited to this setting. The fixed communication structure of the Bitonic Sorter allows for an accurate process placement where the structure of the Bitonic Sorter is aligned to the structure of the cluster.

Another remarkable property of the bitonic sorting network is the potential of working with distributed input and output. The algorithm can work with distributed data without the need to aggregate the data. This is particularly interesting for very large sets of data. We will therefore compare the bitonic sorter to the PSRS algorithm [12], a parallel variant of quicksort with an elaborated pivot selection which guarantees a well-balanced distribution of the resulting lists. A comparison to PSRS is well-suited because the algorithmic structures are rather similar. In Figure 14 the runtime graphs of the PSRS algorithm and the Bitonic Sorter are depicted. The algorithms are modified to work with distributed data, only the sorting time without data distribution and collection is measured.

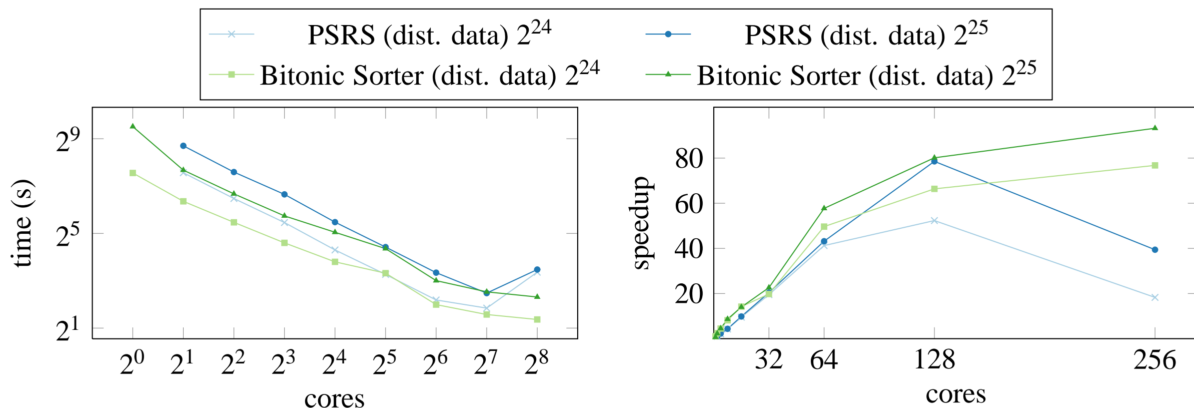


Figure 14: Runtime and Speedup of merge sort with the Bitonic Sorter as merge stage compared to a traditional merge sort on the Beowulf Cluster.

Again, the bitonic sorter scales well in comparison to the PSRS algorithm. With an increasing number of PEs the all-to-all communication of the PSRS algorithm becomes more expensive.

4 Related Work

There have been some newer approaches to sorting networks often in combination with hardware accelerators like FPGAs [17] or GPUs [9]. In particular GPGPU programming has led to a little renaissance of sorting networks, especially with different implementations of the Bitonic Sorter [19, 8, 10] achieving good results. However these approaches usually either implement the bitonic sorter in the original way as presented by Batcher or sometimes implement the Adaptive Bitonic Sorter [2] instead. The latter is a data dependent variant of the Bitonic Sorter and therefore not a sorting network. Consequently the work presented in this paper is closer to the different approaches of hybrid sorting algorithms. There are numerous examples for the benefit of hybrid sorting algorithms, for example in [20] a hybridization of Bucketsort and merge sort yields good results. Some ideas of this work were motivated by Dieterle's [4] work on skeleton composition.

5 Conclusion and Future Work

We have presented a different approach of agglomeration for sorting networks. This technique equips us with the possibility of using sorting networks as a parallel merging stage for arbitrary sorting algorithms, which is a versatile, easily adaptable and very promising approach. We are convinced that further improvements to the given example application are possible. We will investigate different possibilities of constructing combinations of arbitrary sorting algorithms with sorting networks. Therefore we will consider possible connections to embedded languages that allow for GPGPU programming from Haskell such as Accelerate [3] or Obsidian [22] or the possibility of combining the concise and easy to maintain functional implementation of sorting networks with efficient sorting algorithms for example via Haskell's Foreign Function Interface. Furthermore, most of the findings of this paper are applicable to other sorting networks such as Batcher's Odd-Even-Mergesort. All further investigations could benefit from a cost model that enables for better runtime predictions.

Acknowledgments

The author thanks Rita Loogen and the anonymous reviewers for their helpful comments on a previous version of this paper and Wolfgang Loidl, Prabhat Totoo and Phil Trinder for giving us access to their Beowulf Cluster.

References

[1] K. E. Batcher (1968): *Sorting Networks and their Applications*. In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), ACM, New York, NY, USA, pp. 307–314, doi:10.1145/1468075.1468121.

[2] Gianfranco Bilardi & Alexandru Nicolau (1989): *Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared-memory Machines*. *SIAM J. Comput.* 18(2), pp. 216–228, doi:10.1137/0218014.

[3] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell & Vinod Grover (2011): *Accelerating Haskell Array Codes with Multicore GPUs*. In: *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, ACM, New York, NY, USA, pp. 3–14, doi:10.1145/1926354.1926358.

[4] Mischa Dieterle (2016): *Structured Parallelism by Composition*. Ph.D. thesis, Philipps-Universität Marburg, doi:10.17192/z2016.0107.

[5] Mischa Dieterle, Thomas Horstmeyer & Rita Loogen (2010): *Skeleton Composition Using Remote Data*. In Manuel Carro & Ricardo Peña, editors: *Practical Aspects of Declarative Languages*, *Lecture Notes in Computer Science* 5937, Springer Berlin Heidelberg, pp. 73–87, doi:10.1007/978-3-642-11503-5_8.

[6] The MPI Forum (2012): *MPI: A Message-Passing Interface Standard*. Technical Report, Knoxville, TN, USA. Available at https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[7] Ian Foster (1995): *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. Available at http://www.mcs.anl.gov/~itf/dbpp/text/book.html.

[8] Naga Govindaraju, Jim Gray, Ritesh Kumar & Dinesh Manocha (2006): *GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management*. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, ACM, New York, NY, USA, pp. 325–336, doi:10.1145/1142473.1142511.

[9] Naga K. Govindaraju, Nikunj Raghuvanshi & Dinesh Manocha (2005): *Fast and Approximate Stream Mining of Quantiles and Frequencies Using Graphics Processors*. In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, ACM, New York, NY, USA, pp. 611–622, doi:10.1145/1066157.1066227.

[10] Peter Kipfer & Rüdiger Westermann (2005): *Improved GPU Sorting*. In Matt Pharr, editor: *GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley, pp. 733–746. Available at https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter46.html.

[11] Donald E. Knuth (1998): *The Art of Computer Programming: Sorting and Searching*, 2 edition. *The Art of Computer Programming* 3, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[12] Xiaobo Li, Paul Lu, Jonathan Schaeffer, John Shillington, Pok Sze Wong & Hanmao Shi (1993): *On the Versatility of Parallel Sorting by Regular Sampling*. *Parallel Computing* 19(10), pp. 1079–1103, doi:10.1016/0167-8191(93)90019-H.

[13] Rita Loogen (2012): *Eden — Parallel Functional Programming with Haskell*. In Viktória Zsók, Zoltán Horváth & Rinus Plasmeijer, editors: *Proceedings of the 4th Summer School Conference on Central European Functional Programming School, CEFP 11*, *Lecture Notes in Computer Science* 7241, Springer-Verlag, Berlin, Heidelberg, pp. 142–206, doi:10.1007/978-3-642-32096-5_4.

[14] Rita Loogen, Yolanda Ortega-Mallén & Ricardo Peña-Marí (2005): *Parallel Functional Programming in Eden*. *J. Funct. Program.* 15(3), pp. 431–475, doi:10.1017/S0956796805005526.

[15] Welf Löwe (1995): *Optimization of PRAM-Programs with Input-Dependent Memory Access*. In: *Proceedings of the First International Euro-Par Conference on Parallel Processing, Euro-Par '95*, *LNCS* 966, Springer-Verlag, London, UK, UK, pp. 243–254, doi:10.1007/BFb0020469.

[16] Simon Marlow (2010): *Haskell 2010 Language Report*. Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.179.2870.

[17] Rene Mueller, Jens Teubner & Gustavo Alonso (2012): *Sorting networks on FPGAs*. *The VLDB Journal* 21(1), pp. 1–23, doi:10.1007/s00778-011-0232-z.

[18] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone & J. C. Phillips (2008): *GPU Computing*. In: *Proceedings of the IEEE 96*, 5, pp. 879–899, doi:10.1109/JPROC.2008.917757.

[19] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen & Pat Hanrahan (2005): *Photon Mapping on Programmable Graphics Hardware*. In: *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, ACM, New York, NY, USA, pp. 258–267, doi:10.1145/1198555.1198797.

[20] Erik Sintorn & Ulf Assarsson (2008): *Fast Parallel GPU-sorting Using a Hybrid Algorithm*. *J. Parallel Distrib. Comput.* 68(10), pp. 1381–1388, doi:10.1016/j.jpdc.2008.05.012.

[21] H. S. Stone (1971): *Parallel Processing with the Perfect Shuffle*. *IEEE Trans. Comput.* 20(2), pp. 153–161, doi:10.1109/T-C.1971.223205.

[22] Joel Svensson, Mary Sheeran & Koen Claessen (2008): *Obsidian: A Domain Specific Embedded Language for General-Purpose Parallel Programming of Graphics Processors*. In: *Proceedings of the 20th International Conference on Implementation and Application of Functional Languages, IFL '08*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, pp. 156–173, doi:10.1007/978-3-642-24452-0_9.