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 programs to control the granularity of the computation and 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 networks with one process. The agglomeration law in this paper pursues a different strategy: The input data is grouped and the algorithm is generalized 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 and thereby combining 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 this law.

1 Introduction

With the increased presence of parallel hardware the demand for parallel algorithms increases accordingly. Of course this demand includes sorting algorithms as one of the major disciplines in computer science. A particularly interesting class of sorting algorithms for parallelization is the class of *oblivious* algorithms whereupon we will call a parallel algorithm oblivious "iff its communication structure and its communication scheme are the same for all inputs the same size" [14].

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 [17]. 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.

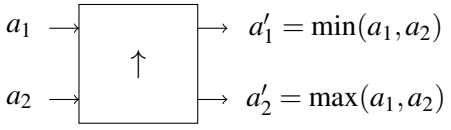


Figure 1: Comparison element (ascending).

Submitted to:
WFLP 2016

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:

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

with $i \neq j$. In a specific step a_i and a_j are sorted with a comparison element, 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 easy to predict behavior and easy to detect parallelism.

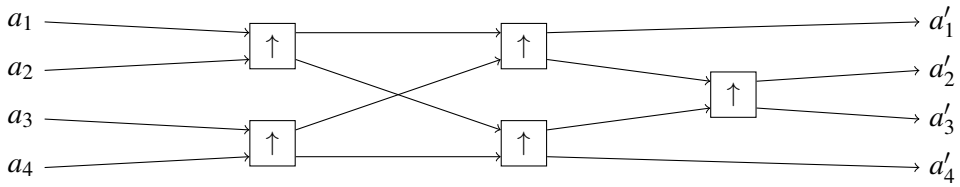


Figure 2: Simple sorting network with comparison elements. Source: [10]

Some well-known sorting algorithms can be described as sorting networks, for example Bubble Sort [10]. Especially in the case of systematically 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 [15].

In practice straightforward implementations of these algorithms often struggle with a too fine granularity of computation and therefore do not scale well. Agglomerating parts of the algorithm is a common step to deal with this problem when designing parallel programs (compare Foster's PCAM method [6]). For recursive algorithms for example it is a usual technique to agglomerate branches of the recursive tree by parallelizing only until a specific depth of recursion. With a coarser granularity the computation to communication ratio improves. A usual agglomeration for sorting networks is to place blocks or rows of comparison elements in one parallel process.

In this paper a different approach is discussed. We will agglomerate the input data and alter the comparison element to work on blocks of data. This approach is not based upon the specific structure of a specific sorting network and can therefore be applied to any sorting network. On the other hand we will see that the limited nature of sorting networks is necessary for this law to be correct. The application of this law 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 the 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 now take a partition of the 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.

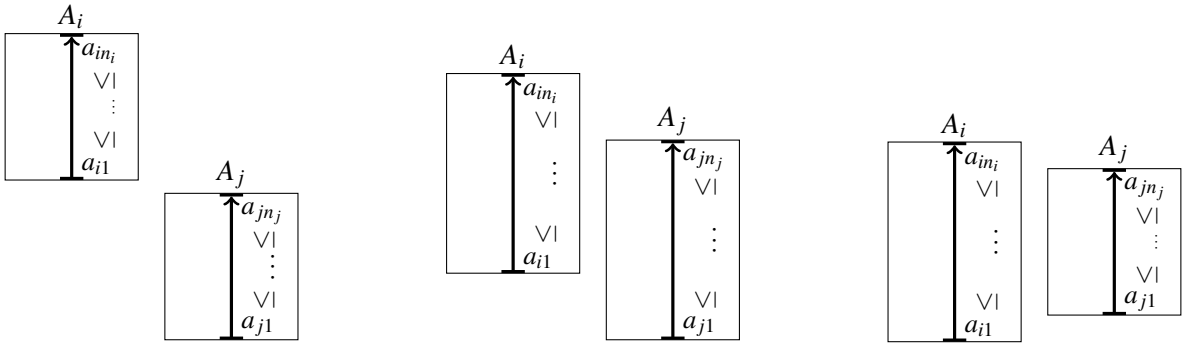
Theorem 1 (Agglomeration Law for Sorting Networks)**.** *Let* $\mathtt{A} = [\mathtt{a_1}, \ldots, \mathtt{a_n}] :: \mathtt{Ord} \Rightarrow [\mathtt{a}]$ *be a sequence with an associated total order "*$\leq$*",* $\mathtt{c} :: (\mathtt{a}, \mathtt{a}) \to (\mathtt{a}, \mathtt{a})$ *a comparison element as described before and*

```
sN :: ((a,a) → (a,a)) → [a] → [a]
```

a correct sorting network, meaning $\mathtt{sN\ c\ A} = \mathtt{A'}$ *with* $\mathtt{A'} = [\mathtt{a'_1}, \ldots, \mathtt{a'_n}]$ *and* $\mathtt{a'_1} \leq \ldots \leq \mathtt{a'_n}$ *where* $\mathtt{a'_1}, \ldots, \mathtt{a'_n}$ *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. And let* $\mathfrak{A} = [A_1, \ldots, A_n]$ *with* $A_i = [a_{i1}, \ldots, a_{in_i}]$*. Then there exists a comparison element* $c' :: (([a], [a]) \to ([a], [a]))$ *with* $\mathrm{S}\ \mathtt{c'}\ \mathfrak{A} = \mathfrak{A}'$*,* $\mathfrak{A}' = [\mathtt{A'_1}, \ldots, \mathtt{A'_n}]$ *and* $\mathtt{A'_1} \preceq \ldots \preceq \mathtt{A'_n}$*. Where* $A \preceq B$ *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 $\mathtt{A'_1}, \ldots, \mathtt{A'_n}$ need not 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 specialize 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) $\max A_j \leq \min A_i$ or $\max A_i \leq \min A_j$

(b) $\max A_j \leq \max A_i$ and $\min A_j \leq \min A_i$ or vice versa

(c) $\max A_j \leq \max A_i$ 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 they do not have an order between one another (meaning neither $\preceq$ nor $\succeq$ holds).

So if for example the input lists do overlap (e.g. $\mathtt{c'}\ ([1,2,3,4],[3,4,5,6])$) a simple swap would not fulfill the requirements. We would rather expect a result like $([1,2,3,3],[4,4,5,6])$ 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 $\mathtt{A'_1}, \ldots, \mathtt{A'_n}$.

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. At first 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:

$$a_1^1 \leq a_1 \leq a_1^2 \qquad \text{and} \qquad a_2^1 \leq a_2 \leq a_2^2$$

If we do sort a_1 and a_2 with an comparison element $(a_1, a_2) \mapsto (a_1', a_2')$ we receive new relations (e.g. $a_1^1 \leq a_1 \Rightarrow a_1^1 \leq a_2'$). We will distinguish between *direct relations* and *conditional relations*. Hereby direct relations refer to all direct resulting relations which are valid in any case 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 the application of the comparison element. Here the direct relations are:

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

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

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

Lemma 1. *Let* $\mathtt{A_1}, \mathtt{A_2} :: (\mathtt{Ord\ a}) \Rightarrow [\mathtt{a}]$ *And let* $\mathtt{c'} :: ([\mathtt{a}], [\mathtt{a}]) \to ([\mathtt{a}], [\mathtt{a}])$ *be a comparison element with* $\mathtt{c'}(\mathtt{A_1}, \mathtt{A_2}) = (\mathtt{A_1'}, \mathtt{A_2'})$ *and* $A_1' \preceq A_2'$ *where 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 and Appendix A).*

Then the direct relations *analogous to the case of the original comparison element do hold.*

Proof.
1. $\mathtt{A_1'} \preceq \mathtt{A_2'}$ is included in the definition.
2. $\max \mathtt{A_1'} \leq \mathtt{ub} \leq \min \mathtt{A_i^2} \Rightarrow \mathtt{A_1'} \preceq \mathtt{A_i^2},\ \mathtt{i} \in \{1,2\}$
3. $\max \mathtt{A_i^1} \leq \mathtt{lb} \leq \min \mathtt{A_2'} \Rightarrow \mathtt{A_i^1} \preceq \mathtt{A_2'},\ \mathtt{i} \in \{1,2\}$
4. $\mathtt{A_i^1} \preceq \mathtt{A_1} \wedge \mathtt{A_i^1} \preceq \mathtt{A_2},\ \mathtt{i} \in \{1,2\} \Rightarrow \mathtt{A_i^1} \preceq [\min(\min \mathtt{A_1}\ \min \mathtt{A_2})] \preceq \mathtt{A_1'} \Rightarrow \mathtt{A_i^1} \preceq \mathtt{A_1'}$
5. $\mathtt{A_1} \preceq \mathtt{A_i^2} \wedge \mathtt{A_2} \preceq \mathtt{A_i^2},\ \mathtt{i} \in \{1,2\} \Rightarrow \mathtt{A_2'} \preceq [\max(\max \mathtt{A_1}\ \max \mathtt{A_2})] \preceq \mathtt{A_i^2} \Rightarrow \mathtt{A_2'} \preceq \mathtt{A_i^2}$
6. $\mathtt{A_i^1} \preceq \mathtt{A_i} \preceq \mathtt{A_j} \Rightarrow \mathtt{A_i^1} \preceq \mathtt{A_1'}$

□

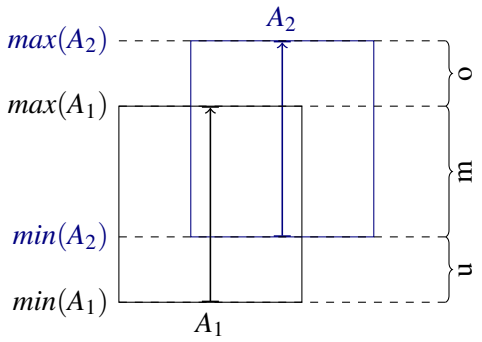


Figure 4: Sections of the comparison element for blocks of data. Elements from u must be in the lesser result (A'_1), elements from o must be in the greater result (A'_2) and elements from m can be in both results as long as $A'_1 \preceq A'_2$ holds.

All other producible information are about *conditional relations* which depend on a condition. 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. In this case every input element has – analogous to the original comparison element – a direct descendant. In which 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 $\mathtt{A}$. Therefore the boundaries of each block can at the most come closer to each other when applying the comparison element and all relations are preserved. An example is given in Figure 5. In Figure 5b we can see A'_1 as the descendant of A_2 and A'_2 as the descendant of A_1.

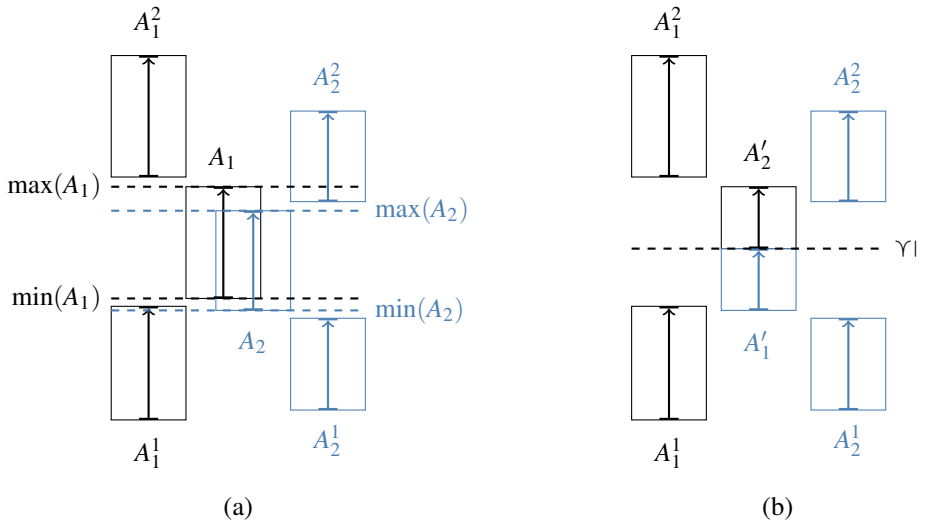


Figure 5: Split of overlapping blocks. In this case the minimal (maximal) element of A_2 is smaller than the minimal (maximal) element of A_1. Thereby A_2 "shrinks" from above, meaning that the maximum element of A'_1 is smaller than $\max A_2$) but this does not give any information about the number of elements in A'_1. A_1 "shrinks" from below. All relevant 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).

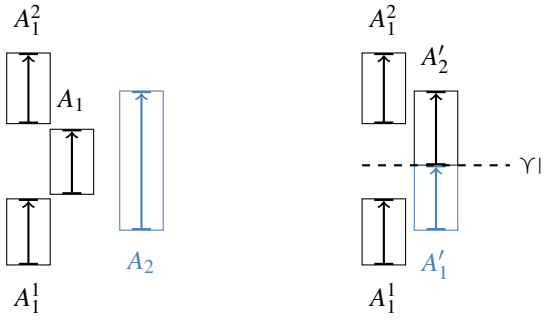


Figure 6: Split of encased blocks. There are no direct descendants. $A_1' \preceq A_1^2$ and $A_1^1 \preceq A_2'$ but neither A_1' nor A_2' is between A_1^1 and A_1^2.

This means that the 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 but with sorting networks because the comparison element does not compare one fixed element with another element but does rather return 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 previously mentioned *conditional* information. Some of these conditional information can no longer be guaranteed to hold but can not be used in a sorting network in any case because of the limited operations of sorting networks. The relations of concern are

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

resulting from $a_i \leq a_j \Rightarrow (a_i^1 \leq a_1' \wedge a_2' \leq a_j^2),\ \{i,j\} = \{1,2\}$ and $a_1 \leq a_2 \vee a_2 \leq a_1$.

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. All information generated by the sorting network are of the form

1. $a_1 \leq a_j$ (direct relations)
2. $a_i \leq a_j \Rightarrow a_k \leq a_l$ (conditional relations I)
3. $a_i \leq a_j \vee a_k \leq a_l$ (conditional relations II).

In particular the information that $a_i \not\leq a_j$ can not be produced for any i and j. Furthermore it is not possible to equalize 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. The conditional relation 7 does only exists if we do not know wether or not $a_1 \leq a_2$ or vice versa. Otherwise it is the case of Equation 6, a direct relation. We can not test wether one side of Equation 7 is false or if both sides are equal and therefore the relation can not be used.

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 every 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 for Haskell lists can be found in Appendix A. 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 parallelized algorithm, for example if the parallelization is done with a computer cluster. By preserving the inner sorting of the blocks, the result 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 parallelize arbitrary sorting algorithms and work as the merging stage of the newly combined (parallel) algorithm. A concept that will prove its 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 systematically 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).

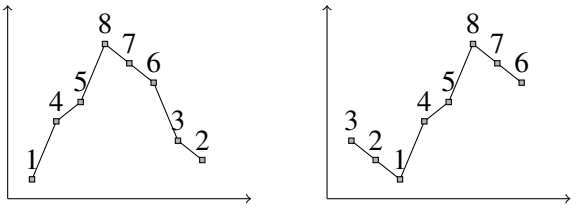


Figure 7: Examples of bitonic sequences.

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.

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.

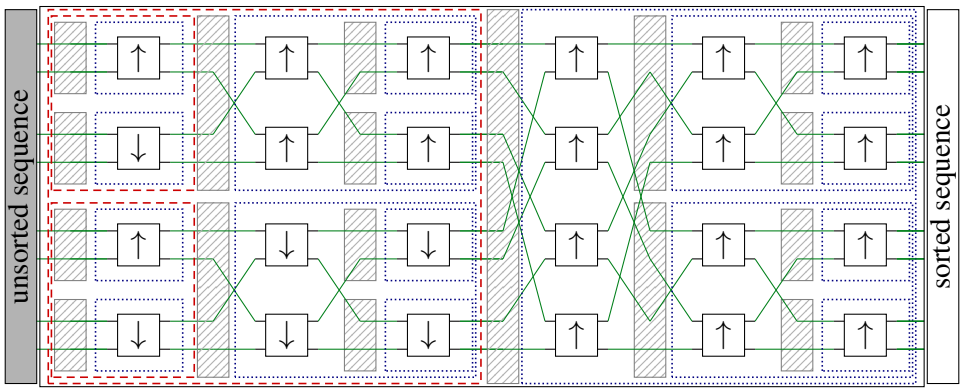


Figure 8: Bitonic Sorter of order 8. The function `prodBList` is represented by a red dashed rectangle, the function `bMerge` by a blue dotted one. Bitonic Sequences are represented by shaded rectangles.

We will define the actual algorithm using the Eden[13, 12] programming language which extends Haskell with the concept of parallel *processes* with an implicit communication as well as a Remote Data [4] 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).

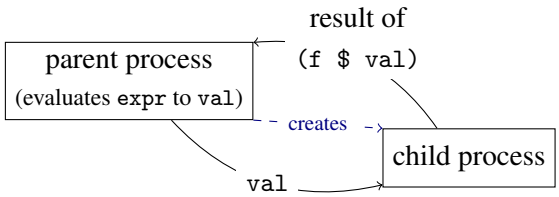


Figure 9: The scheme for process instantiation. Source: [12]

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 realized 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

```
37 bSort :: Trans a
38      ⇒ (Direction → [a] → [a]) -- ^specialized comparison element
39      → Direction               -- ^sorting direction
40      → [a] → [a]               -- ^input and ^output
41 bSort _          _ [ ] = [ ]
42 bSort _          _ [x] = [x]
43 bSort sCompElem d xss = (bMerge sCompElem d) ∘ prodBList $ xss where
44     prodBList = unSplit ∘ pMap bSort' ∘ zip [Up, Down] ∘ splitHalf
45     bSort' =  uncurry (bSort sCompElem)
46     pMap = parMapAt [selfPe, selfPe+hcc]
47     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 ascending or descending and an input list. The main part of the algorithm is a composition of the `prodBList` and the `bMerge` function (cf. Line 43 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 44). 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 (row wise) placement is calculated depending on the width of the sorting network (cf. Line 47). 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 than the `bSort` function but the input list must be a bitonic list for the function to work correct:

Listing 3: Parallel bMerge

```
50 bMerge :: Trans a
51      ⇒ (Direction → [a] → [a]) -- ^specialized comparison element
52      → Direction               -- ^sorting direction
53      → [a] → [a]               -- ^input and ^output
54 bMerge sCompElem d xss@[x,y] = sCompElem d xss
55 bMerge sCompElem d xss = unSplit ∘ pMap (bMerge sCompElem d) ∘ bSplit $ xss where
56     bSplit = splitHalf ∘ shuffle ∘ pMap' (sCompElem d) ∘ perfectShuffle
57     pMap = parMapAt [selfPe, selfPe+hcc]
58     hcc = (length xss) `div` 4 {- half comparator count -}
59     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 [20]. With this communication scheme the element i and $i + \frac{p}{2}$ are compared resulting in a split depicted in Figure 10.

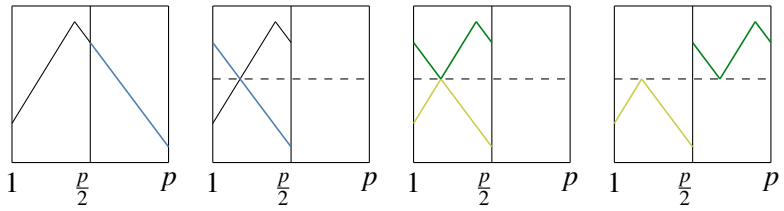


Figure 10: Concept of splitting a bitonic sequence.

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

```
-- Round robin distribution - inverse to shuffle
unshuffle :: Int → [a] → [[a]]
-- Simple shuffling - inverse to round robin distribution:
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:

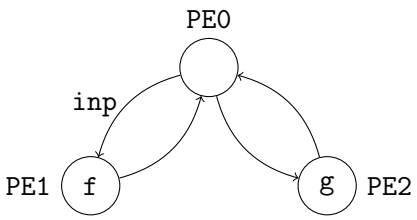
```
perfectShuffle :: [a] → [[a]]
perfectShuffle xs = unshuffle halfSize xs
                        where halfSize = (length xs) `div` 2
```

A direct communication between consecutive comparison elements can be realized 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. This can be done by the basic operations `fetch` and `release`:

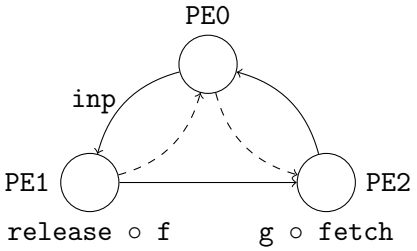
```
type RD a -- remote data

-- converts local data into corresponding remote data.
release :: Trans a ⇒ a → RD a
-- convert remote data into local data
fetch :: Trans a ⇒ RD a → a

-- list variants
releaseAll :: Trans a ⇒ [a] → [RD a]
fetchAll :: Trans a ⇒ [RD a] → [a]
```

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

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



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



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

Figure 11: Remote Data scheme. Source: [12]. The process computing the result of the function `f` is placed on one PE, the second process computing the result of the function `g` is placed on another PE. Without RD, the result of `f` is transferred via the parental process (placed on `PE0`). With RD a handle is generated on `PE1` and transferred via `PE0` to `PE2`. The actual result is transferred directly from `PE1` to `PE2`. With more intermediate steps involved, the benefits of this concept are getting more effective.

If we call the `bSort` function with the original comparison element and the needed 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

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

To apply the agglomeration law we can change the comparison element to the previously discussed comparison element from Appendix A. In Listing 5 an optimized implementation is given which uses unboxed vectors to optimize transmissions. For reasons of comparability, the list variant of the altered comparison element and the merge sort from `Data.Lists` is used.

Listing 5: Combination of Mergesort with the Bitonic Sorter

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

This simple adaption results in a hybrid sorting algorithm parallelizing merge sort with the Bitonic Sorter. We tested the algorithms on the multicore computer Hex and on the Beowulf Cluster[2] in order to compare the different implementations of Eden: with MPI[5] as a middleware and an implementation optimized for multicore computers. First we will compare the above parallelization of merge sort using the Bitonic Sorter, and another parallelization of the same merge sort using the disDC divide-and-conquer skeleton from Eden's skeleton library. Therefore both variants are implemented in Eden and equipped with similar improvements. We will work on lists in particular since they are the usual 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.

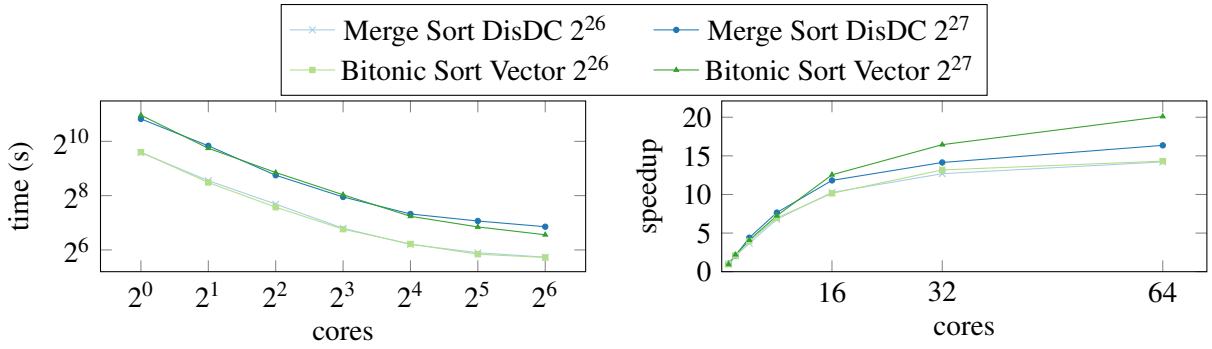


Figure 12: Runtime and Speedup of the Bitonic Sorter and merge sort on Hex with 2^{26} and 2^{27} elements.

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 that consists 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)

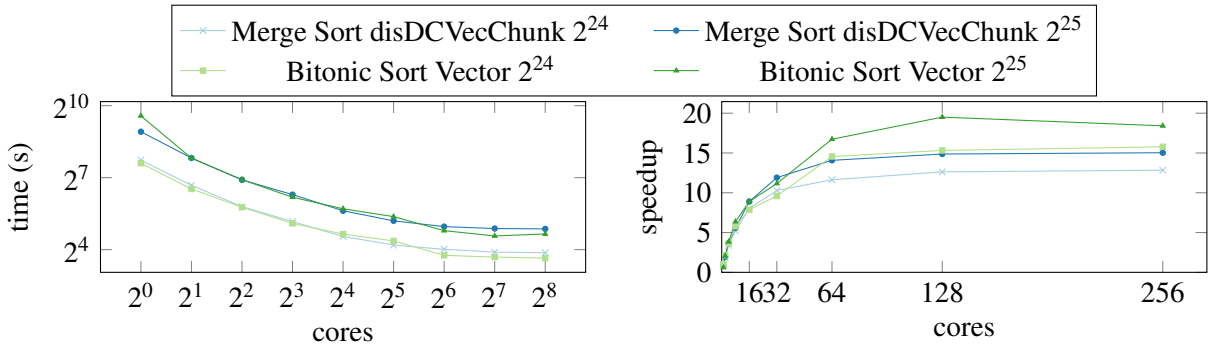


Figure 13: Runtime and Speedup of merge sort with the Bitonic Sorter as merge stage compared to a traditional merge sort on the Beowulf Cluster with 2^{24} elements.

On the Beowulf Cluster the communication between different `PEs` located on the same computer is cheap while intercommunication between computers is proportionally slow. In this setting the local

[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.

communication structure of the bitonic sorting network is well suited. 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 has 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 [11], 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.

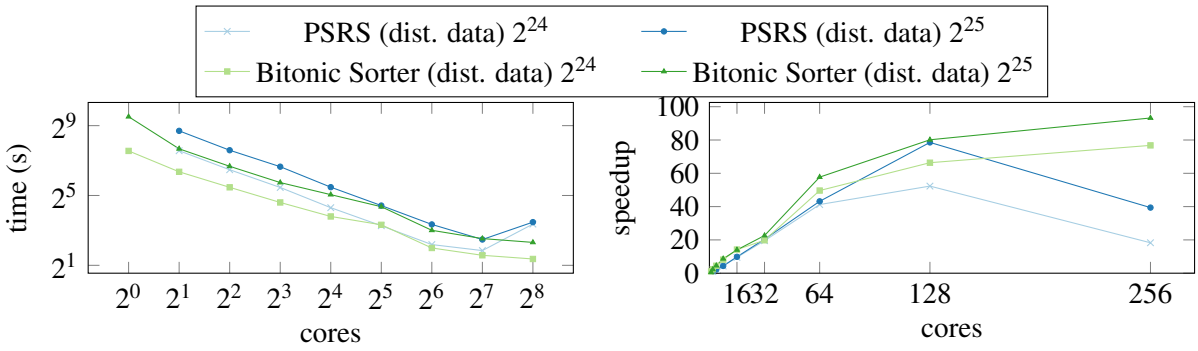


Figure 14: Runtime and Speedup of merge sort with the Bitonic Sorter as merge stage compared to a traditional merge sort on the Beowulf Cluster with 2^{24} elements.

4 Related Work

There have been some newer approaches to sorting networks often in combination with hardware accelerators like FPGAs [16] or GPUs [8]. In particular GPGPU programming has led to a little renaissance of sorting networks, especially with different implementations of the Bitonic Sorter [18, 7, 9] achieving good results. However these realizations 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 [19] a hybridization of Bucketsort and Mergesort yields good results. Some ideas of this work were motivated by Dieterle's [3] 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 to use sorting networks as a parallel merging stage for arbitrary sorting algorithms. A versatile, easy adaptable and very promising approach. We are convinced that further improvements

to the given example application are possible. We will further investigate different possibility of constructing different combinations of arbitrary sorting algorithms with sorting networks. Therefore we will investigate possible connections to embedded languages that allow for GPGPU programming from Haskell such as Accelerate[3] or Obsidian[4] or the possibility to combine 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, Prabjat 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] Mischa Dieterle (2016): *Structured Parallelism by Composition*. Ph.D. thesis, Philipps-Universität Marburg, doi:10.17192/z2016.0107.

[4] 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.

[5] 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`.

[6] 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`.

[7] 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.

[8] 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.

[9] 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 `http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter46.html`.

[3] url: `https://github.com/AccelerateHS/accelerate/`

[4] url: `https://github.com/svenssonjoel/Obsidian`

[10] 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.

[11] 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.

[12] Rita Loogen (2012): *Eden — Parallel Functional Programming with Haskell*. In: *Proceedings of the 4th Summer School Conference on Central European Functional Programming School*, CEFP'11, Springer-Verlag, Berlin, Heidelberg, pp. 142–206, doi:10.1007/978-3-642-32096-5_4. Available at `http://www.mathematik.uni-marburg.de/~eden`.

[13] 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.

[14] 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, Springer-Verlag, London, UK, UK, pp. 243–254, doi:10.1007/BFb0020469.

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

[16] 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.

[17] 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.

[18] 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, doi:10.1145/1198555.1198797.

[19] 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.

[20] 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.

Appendices

A A simple, balanced `mergeSplit` for comparison elements

Listing 6: A simple MergeSplit function working as a comparison element for blocks of data

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