

FUNCTIONAL PEARLS

Parberry's Pairwise Sorting Network Revealed

RALF HINZE

Fachbereich Informatik, Technische Universität Kaiserslautern
67653 Kaiserslautern, Germany
(e-mail: ralf-hinze@cs.uni-kl.de)

CLARE MARTIN

Department of Computing and Communication Technologies,
Oxford Brookes University, Wheatley, Oxford, OX33 1HX, England
(e-mail: cemartin@brookes.ac.uk)

1 Introduction

Batcher's "merge exchange" sorting network, discussed in a previous pearl (Hinze & Martin, 2016), remains one of the best practical algorithms for oblivious sorting, even almost half a century after its inception. So it is surprising that an algorithm with exactly the same level of performance, devised two decades later by Parberry (1992), has been relatively overlooked. Perhaps a reason for its lack of celebrity is that Parberry's design is not immediately recognizable, whereas the Batcher method has a familiar ring, as a hard-wired implementation of merge sort. Here we hope to rectify this imbalance by unravelling Parberry's algorithm and uncoupling its close relationship to Batcher's.

Interestingly, Parberry *derives* his network using the *zero-one principle* (Knuth, 1998). We abandon this traditional method, in favour of a feature of comparison networks that we consider to be more fundamental: monotonicity. We shall see that this property, used before to demystify Batcher's merger (Hinze & Martin, 2016), also helps to shed some light on Parberry's design. To keep the pearl reasonably self-contained we start with a quick recap of the notation and Batcher's construction.

2 Recap

Sorting networks work nicely if the underlying structure is a distributive lattice instead of a total order (Bove & Coquand, 2006). We make the same assumption here, in particular:

$$x \leq a \wedge x \leq b \iff x \leq a \downarrow b \tag{1a}$$

$$a \uparrow b \leq x \iff a \leq x \wedge b \leq x \tag{1b}$$

These properties imply, for example, that minimum, or meet, and maximum, or join, are monotonic. The ordering \leq is lifted pointwise to sequences of the same length, $x \leq y \iff \text{and}(\text{zip } f(\leq) x y)$. Here $\text{zip } f$ combines sequences of equal length with f while the function and conjoins the result. Binary operators \oplus , such as minimum and maximum, are lifted

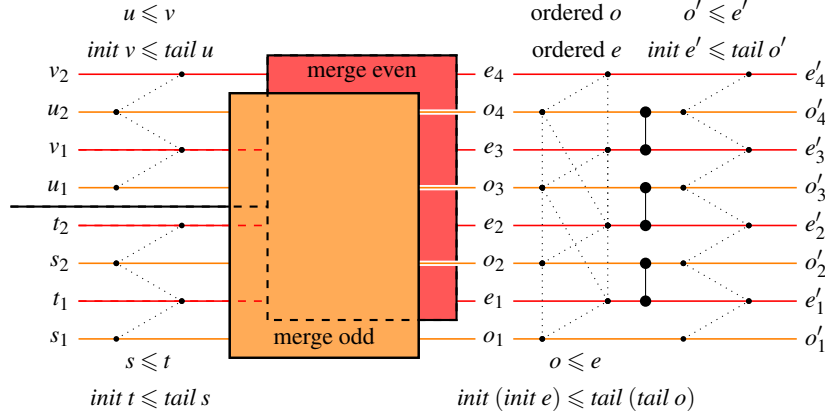


Fig. 1. Batcher's odd-even merging network.

similarly: $x \oplus y = \text{zip}(\oplus) x y$. Consequently, concatenation (\cdot) is an order-embedding:

$$w \cdot x \leq y \cdot z \iff w \leq y \wedge x \leq z \quad (2)$$

and it interacts with lifted operations in an interesting way (middle-two interchange law):

$$(w \cdot x) \oplus (y \cdot z) = (w \oplus y) \cdot (x \oplus z) \quad (3)$$

Now, Batcher's "merge exchange" sorting network uses a standard divide-and-conquer construction: the first and last halves of an input sequence are sorted independently and then a merger is applied to the result (the pattern $x \parallel y$ splits a sequence into two *equal* halves and \bowtie is merge):

$$\begin{aligned} \text{sort} : A^n &\rightarrow A^n \text{ where } n = 2^k \\ \text{sort} \langle a \rangle &= \langle a \rangle \\ \text{sort} (x \parallel y) &= \text{sort } x \bowtie \text{sort } y \end{aligned}$$

Throughout we assume for simplicity that the length of the input to the sorter is an exact power of two. The interesting aspect of Batcher's design is his merger, which also relies on divide-and-conquer: the input sequences are divided into odd and even sub-sequences that are merged in parallel by recursively defined sub-networks (Υ is interleaving, so for example $\langle 1, 2 \rangle \Upsilon \langle 3, 4 \rangle = \langle 1, 3, 2, 4 \rangle$, and \downarrow is the *cleaner*, so-called because it performs the final 'clean-up' phase of the merger):

$$\begin{aligned} \langle a \rangle \bowtie \langle b \rangle &= \langle a \downarrow b, a \uparrow b \rangle \\ (s \Upsilon t) \bowtie (u \Upsilon v) &= (s \bowtie u) \downarrow (t \bowtie v) \\ (\langle a \rangle \cdot x) \downarrow (y \cdot \langle b \rangle) &= (\langle a \rangle \cdot (x \uparrow y)) \Upsilon ((x \downarrow y) \cdot \langle b \rangle) \end{aligned}$$

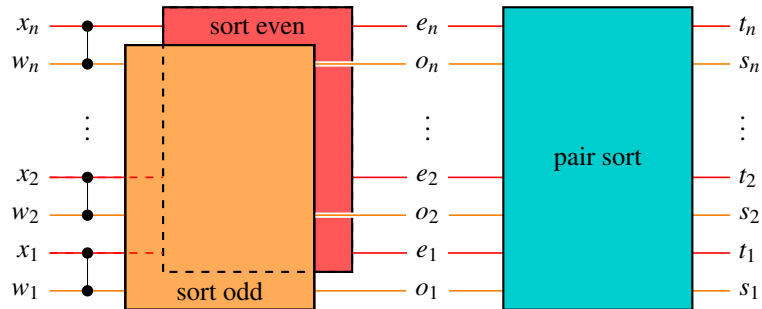
This recursive decomposition of the merger is illustrated in Fig. 1, which merits careful study. As a reminder, the data flows from left to right; the perpendicular comparators (\bullet) sort the values on their two input wires into vertically ascending order. The figure additionally integrates Hasse diagrams (depicted by small discs \bullet and dotted lines \cdots) to visualize the pre- and post-conditions of each phase, as indicated also by the formulae

above and below the diagrams. The sub-networks *merge odd* and *merge even* each fuse two ordered sequences into one. The outputs are then fed into a final column of comparators, representing the cleaner \Downarrow , which produces the desired ordered output.

At first it seems rather magical that it is sufficient for the cleaner to compare only the values on the inner adjacent wires. The construction can be disenchanting using the monotonicity property of comparator networks: the pre-conditions of the two sub-mergers imply the pre-condition of the cleaner. For the proof we refer to the previous pearl (Hinze & Martin, 2016). Interestingly, we shall see that the correctness of the cleaner also arises as a special case of Parberry’s construction, but we are leaping ahead.

3 Parberry’s Pairwise Sorting Network

The idea of Parberry’s design is to treat the $2 \times n$ inputs as pairs. Sorting proceeds in three steps. First, the pairs are sorted “internally” using n parallel 2-sorters, shown in the first column of the diagram below. The first component of each pair of wires is then at most the corresponding second component. Second, the pairs are sorted “externally” using 2 parallel n -sorters (*sort odd* and *even* below), so that consecutive pairs are related by the pointwise ordering. The third phase (*pair sort*), which is “slightly more complicated” (Parberry, 1992), then establishes the desired overall ordering.



The 2-sorters are, of course, just comparators; the n -sorters are given by “recursive invocations” of Parberry’s scheme. So, like Batchier’s sorter, Parberry’s is based on divide-and-conquer. The sub-problems are constructed differently however: the former halves the inputs, while the latter uninterleaves them. There is one further difference: Parberry does not use a merger. Instead, some work is done before the recursive invocations, and some work is done afterwards.

The diagram translates into the following definition:

$$\begin{aligned}
 \text{sort}_n &: A^n \rightarrow A^n \text{ where } n = 2^k \\
 \text{sort}_1 &= id \\
 \text{sort}_{2 \times n} &= \text{isort}_n ; \text{esort}_n ; \text{pair-sort}_n \\
 \text{isort}_n (w \Upsilon x) &= (w \Downarrow x) \Upsilon (w \Uparrow x) \\
 \text{esort}_n (s \Upsilon t) &= \text{sort}_n s \Upsilon \text{sort}_n t
 \end{aligned}$$

where ‘;’ is forward functional composition, *isort* and *esort* are the internal and external sorters, and *pair-sort* denotes the third and final phase, the implementation of which we seek to derive.

It is worth recording that the ordering initially established by the internal sort is preserved by the subsequent external sort. This is again due to the monotonicity property of comparator networks (Hinze & Martin, 2016), which ensures that:

$$y \leq z \implies \text{sort}_n y \leq \text{sort}_n z$$

Since $w \downarrow x \leq w \uparrow x$, we have $\text{sort}_n (w \downarrow x) \leq \text{sort}_n (w \uparrow x)$. Thus, the input of pair-sort_n is an interleaving $o \curlyvee e$ with

$$o \text{ ordered} \wedge o \leq e \wedge e \text{ ordered} \quad (4)$$

Notice the resemblance to the pre-condition of Batchner's cleaner in Fig. 1, which was slightly stronger: the inputs additionally satisfied

$$\text{init} (\text{init } e) \leq \text{tail} (\text{tail } o). \quad (5)$$

The nested invocations of init and tail are actually slightly awkward to work with. For the derivation of pair-sort it will be prudent to introduce some notation for suffixes and prefixes, generalizing tail and init . This is what we do next.

4 A Little Theory of Segments

Surprisingly, there seems to be no established notation for prefixes and suffixes. For example, the language Haskell takes a positive approach to prefixes, $\text{take } i x$, and a negative to suffixes, $\text{drop } i x$. With a positive mindset i specifies the number of elements retained, rather than the number of elements discarded. We adopt a uniform approach, using the symbol \diagdown for drop , and \diagup for its dual:

$$\begin{array}{ll} (\diagdown) : (i : \{0 \dots n\}) \times A^n \rightarrow A^{n-i} & (\diagup) : A^n \times (i : \{0 \dots n\}) \rightarrow A^{n-i} \\ 0 \diagdown x = x & x \diagup 0 = x \\ (i+1) \diagdown (\langle a \rangle \cdot x) = i \diagdown x & (x \cdot \langle a \rangle) \diagup (i+1) = x \diagup i \end{array}$$

Uniform, but negative: if x has length n , then the length of both $x \diagup i$ and $i \diagdown x$ is $n-i$, so $i \diagdown x$ is $\text{take } (n-i) x$. We stipulate that \diagdown and \diagup bind more tightly than the other operators. Both init and tail arise as special cases: $\text{tail } x = 1 \diagdown x$ and $\text{init } x = x \diagup 1$.

Turning to the properties, the operators enjoy *pseudo-associative* laws:

$$(x \diagup i) \diagdown j = x \diagup (i+j) \quad (6a)$$

$$i \diagdown (j \diagdown x) = (i+j) \diagdown x \quad (6b)$$

$$(i \diagdown x) \diagup j = i \diagdown (x \diagup j) \quad (6c)$$

The last law expresses that a prefix of a suffix is the same as a suffix of a prefix, allowing us to write $i \diagdown x \diagup j$ unambiguously without any parentheses.

Both $(\diagup i)$ and $(i \diagdown)$ are *monotonic*:

$$x \leq y \implies x \diagup i \leq y \diagup i \quad (7a)$$

$$x \leq y \implies i \diagdown x \leq i \diagdown y \quad (7b)$$

and they *distribute* over lifted operations:

$$(x \oplus y) \diagup i = (x \diagup i) \oplus (y \diagup i) \quad (8a)$$

$$i \diagdown (x \oplus y) = (i \diagdown x) \oplus (i \diagdown y) \quad (8b)$$

A law that we will use repeatedly is the *split property*:

$$i + j = n \iff x \setminus i \cdot j / x = x \quad (9)$$

where x has length n and $0 \leq i, j \leq n$. The lemma is particularly useful in conjunction with the order-embedding (2) to simplify inequalities between sequences.

Using prefixes and suffixes we can capture that a sequence is *ordered*:¹

$$x \text{ ordered} \iff \forall k . x \setminus k \leq k / x \quad (10)$$

We assume that variables implicitly range over the domain of the operations involved, here: $0 \leq k \leq n$ where n is the length of x . Note that the right-hand side is trivially true if x is the empty sequence, written $\langle \rangle$.

Prefixes and suffixes of an ordered sequence are clearly ordered, as well:

$$x \setminus i \text{ ordered} \iff x \text{ ordered} \quad (11a)$$

$$i / x \text{ ordered} \iff x \text{ ordered} \quad (11b)$$

Let us actually prove (11a) to illustrate some of the previous definitions and laws:

$$\begin{aligned} & x \setminus i \text{ ordered} \\ \iff & \{ \text{definition ordered (10)} \} \\ & \forall k . x \setminus i \setminus k \leq k / x \setminus i \\ \iff & \{ (6a) \text{ twice and commutativity of } + \} \\ & \forall k . x \setminus k \setminus i \leq k / x \setminus i \\ \iff & \{ (\setminus i) \text{ monotonic (7a)} \} \\ & \forall k . x \setminus k \leq k / x \\ \iff & \{ \text{definition ordered (10)} \} \\ & x \text{ ordered} \end{aligned}$$

Minimum and maximum also preserve ordered sequences:

$$x \downarrow y \text{ ordered} \iff x \text{ ordered} \wedge y \text{ ordered} \quad (12a)$$

$$x \uparrow y \text{ ordered} \iff x \text{ ordered} \wedge y \text{ ordered} \quad (12b)$$

Finally, concatenations of non-empty sequences enjoy the *link property*:

$$x \cdot y \text{ ordered} \iff x \text{ ordered} \wedge \text{last } x \leq \text{head } y \wedge y \text{ ordered} \quad (13)$$

while non-empty interleavings satisfy the *zig-zag property* (Hinze & Martin, 2016), named after the corresponding Hasse diagram, as shown in Figure 1:

$$x \curlywedge y \text{ ordered} \iff x \leq y \wedge \text{init } y \leq \text{tail } x \quad (14)$$

¹ The previous pearl (Hinze & Martin, 2016) featured an alternative definition of “orderedness”. The proof that the two definitions are equivalent is left as an instructive exercise to the reader.

5 Deriving the Pair Sorter

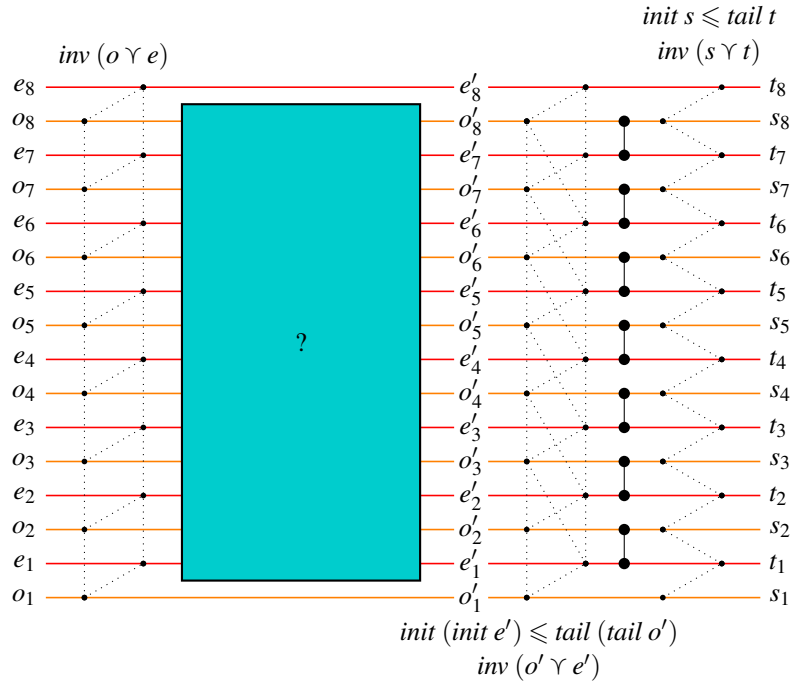
Equipped with this machinery we are now ready to derive the third component of Parberry's sorter. We shall maintain the post-condition of the first two phases (4) as an invariant:

$$\text{inv}(o \curlywedge e) \iff o \text{ ordered} \wedge o \leq e \wedge e \text{ ordered} \quad (15)$$

This invariant can be seen as the essence of "pairwise sorting". It states that both odd and even sub-sequences are ordered (external ordering of pairs), and the former is at most the latter (internal ordering of pairs).

5.1 Back to Batcher

Recall the similarity of the invariant to the pre-condition of Batcher's cleaner column, shown in Fig. 1. The invariant (15) is only missing the conjunct (5). This suggests that we might try to re-use the cleaner as part of the current design. We illustrate this in the following sketch for the pair sorter. The diagram below again integrates Hasse diagrams, annotated with formulae, to visualize the required or guaranteed order of elements. By the zig-zag property (14), it suffices to show that $\text{init } t \leq \text{tail } s$ (the zag) for the output $s \curlywedge t$ to be ordered, as the invariant guarantees that $s \leq t$ (the zig).



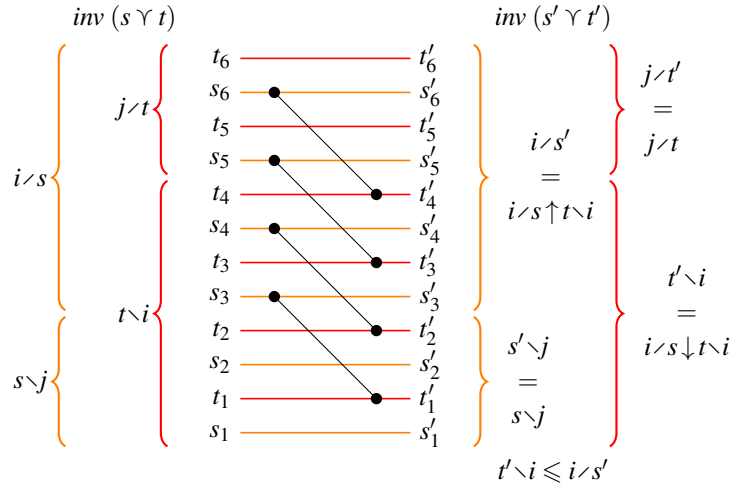
Our next task is to derive the unknown circuit in the box. We begin by expressing its post-condition $\text{init}(\text{init } e') \leq \text{tail}(\text{tail } o')$ using our notation for suffixes and prefixes as $e' \setminus 2 \leq 2 \setminus o'$. We then adopt the classic technique of replacing the constant 2 by a variable, i , to give an intermediate post-condition:

$$\text{post}_i(s' \curlywedge t') \iff t' \setminus i \leq i \setminus s' \quad (16)$$

where $1 \leq i \leq n$ and n is the length of output variables s' and t' . A circuit that establishes this condition is not hard to construct: we enforce the inequality by installing suitable comparators. For example, if $i = 2$ and $n = 6$, the post-condition becomes:

$$post_2 (s' \Upsilon t') \iff \langle t'_1, t'_2, t'_3, t'_4 \rangle \leq \langle s'_3, s'_4, s'_5, s'_6 \rangle.$$

The diagram below shows four comparators that achieve this ordering; for instance t_1 is compared with s_3 . The braces indicate the prefixes and suffixes involved in this case.



In this example, $clean_2 (s \Upsilon t)$, we have $s' = s \setminus 4 \cdot (2 \setminus s \uparrow t \setminus 2)$ and $t' = (2 \setminus s \downarrow t \setminus 2) \cdot 4 \setminus t$. This suggests the following generalized circuit:

$$\begin{aligned} clean_i : A^{2 \times n} &\rightarrow A^{2 \times n} \text{ where } n \geq 1 \\ clean_i (s \Upsilon t) &= (s \setminus j \cdot (i \setminus s \uparrow t \setminus i)) \Upsilon ((i \setminus s \downarrow t \setminus i) \cdot j \setminus t) \text{ where } i + j = n \end{aligned}$$

The output $s' \Upsilon t'$ then satisfies $t' \setminus i = i \setminus s \downarrow t \setminus i$ and $i \setminus s' = i \setminus s \uparrow t \setminus i$ so the post-condition (16) holds by construction. Notice that Batcher's cleaner \downarrow falls out as a special case of this one: it is simply $clean_1$.

To summarize, the circuit $clean_i (s \Upsilon t)$ has been constructed to achieve the post-condition $t' \setminus i \leq i \setminus s'$, but we also require that it maintains the invariant. So now we need to derive the pre-condition that provides this guarantee.

5.2 Deriving the Pre-condition

To recap, the output $(s' \Upsilon t')$ of the circuit $clean_i (s \Upsilon t)$ needs to satisfy the invariant $inv (s' \Upsilon t')$, defined in (15):

$$inv (s' \Upsilon t') \iff s' \text{ ordered} \wedge s' \leq t' \wedge t' \text{ ordered}$$

First we show that s' is ordered; the proof for t' is similar.

$$\begin{aligned} &s' \text{ ordered} \\ \iff &\{ \text{link property (13) and definition of } s' \} \\ &s \setminus j \text{ ordered} \wedge last (s \setminus j) \leq head (i \setminus s \uparrow t \setminus i) \wedge i \setminus s \uparrow t \setminus i \text{ ordered} \end{aligned}$$

$$\begin{aligned}
&\iff \{ s \text{ and } t \text{ are ordered, } (k\prime), (\setminus k), \text{ and } \uparrow \text{ preserve orderedness (11a)–(12b)} \} \\
&\quad \text{last } (s\setminus j) \leq \text{head } (i\prime s \uparrow t\setminus i) \\
&\iff \{ \text{head distributes over } \uparrow \text{ (8a)} \} \\
&\quad \text{last } (s\setminus j) \leq \text{head } (i\prime s) \uparrow \text{head } (t\setminus i) \\
&\iff \{ \text{transitivity and } a \leq a \uparrow b \} \\
&\quad \text{last } (s\setminus j) \leq \text{head } (i\prime s) \\
&\iff \{ \text{link property (13)} \} \\
&\quad s\setminus j \cdot i\prime s \text{ ordered} \\
&\iff \{ i + j = n, \text{ so } s = s\setminus j \cdot i\prime s \} \\
&\quad s \text{ ordered}
\end{aligned}$$

To establish the remaining part of $\text{inv } (s' \vee t')$, namely, $s' \leq t'$, we work towards a situation where we can apply the defining properties of minimum (1a) and maximum (1b):

$$\begin{aligned}
&s' \leq t' \\
&\iff \{ \text{definitions of } s' \text{ and } t' \} \\
&\quad s\setminus j \cdot (i\prime s \uparrow t\setminus i) \leq (i\prime s \downarrow t\setminus i) \cdot j\prime t \\
&\iff \{ \downarrow \text{ and } \uparrow \text{ are idempotent} \} \\
&\quad (s\setminus j \uparrow s\setminus j) \cdot (i\prime s \uparrow t\setminus i) \leq (i\prime s \downarrow t\setminus i) \cdot (j\prime t \downarrow j\prime t) \\
&\iff \{ \text{middle-two interchange law (3)} \} \\
&\quad (s\setminus j \cdot i\prime s) \uparrow (s\setminus j \cdot t\setminus i) \leq (i\prime s \cdot j\prime t) \downarrow (t\setminus i \cdot j\prime t) \\
&\iff \{ \text{split property (9) twice} \} \\
&\quad s \uparrow (s\setminus j \cdot t\setminus i) \leq (i\prime s \cdot j\prime t) \downarrow t \\
&\iff \{ \text{minimum (1a) and maximum (1b)} \} \\
&\quad s \leq i\prime s \cdot j\prime t \wedge s \leq t \wedge s\setminus j \cdot t\setminus i \leq i\prime s \cdot j\prime t \wedge s\setminus j \cdot t\setminus i \leq t
\end{aligned}$$

The second conjunct is immediate from the invariant (15) and the first and fourth are direct consequences of it. The third conjunct is more challenging. The basic idea for establishing the inequality $s\setminus j \cdot t\setminus i \leq i\prime s \cdot j\prime t$ is to align suitable segments of the left- and right-hand side, so that we can apply equivalence (2). As the segmentation depends on the relative order of i and j , we have to make a case distinction:

i	\leq	j		i	\geq	j
$s\setminus j$		$t\setminus i$		$s\setminus j$		$t\setminus i$
$s\setminus j$	$t\setminus i\setminus i$	$(j-i)\prime t\setminus i$		$s\setminus j\setminus (i-j)$	$j\prime s\setminus j$	$t\setminus i$
$i\prime s\setminus (j-i)$	$i\prime i\prime s$	$j\prime t$		$i\prime s$	$j\prime t\setminus j$	$(i-j)\prime j\prime t$
$i\prime s$		$j\prime t$		$i\prime s$	$j\prime t$	
j		i		j	i	

As an aside, this step uses a central property of free monoids: *equidivisibility*: if $u \cdot v = x \cdot y$, then there exists a sequence w such that either $u \cdot w = x$ and $v = w \cdot y$ or $u = x \cdot w$ and $w \cdot v = y$.

Case $i \geq j$ ($\iff 2 \times i \geq n$):

$$\begin{aligned}
& s \setminus j \cdot t \setminus i \leq i \setminus s \cdot j \setminus t \\
\iff & \{ \text{split property (9), see diagram above, (6a) and (6b)} \} \\
& s \setminus i \cdot j \setminus s \setminus j \cdot t \setminus i \leq i \setminus s \cdot j \setminus t \setminus j \cdot i \setminus t \\
\iff & \{ \text{pointwise ordering (2) twice} \} \\
& s \setminus i \leq i \setminus s \wedge j \setminus s \setminus j \leq j \setminus t \setminus j \wedge t \setminus i \leq i \setminus t \\
\iff & \{ \text{definition ordered (10), } (k \setminus) \text{ and } (\setminus k) \text{ are monotonic (7a)–(7b)} \} \\
& s \text{ ordered} \wedge s \leq t \wedge t \text{ ordered}
\end{aligned}$$

So, once again, the invariant (15) provides the guarantee that we seek.

Case $i \leq j$ ($\iff 2 \times i \leq n$): we reason:

$$\begin{aligned}
& s \setminus j \cdot t \setminus i \leq i \setminus s \cdot j \setminus t \\
\iff & \{ \text{split property (9), see diagram above} \} \\
& s \setminus j \cdot t \setminus i \setminus i \cdot (j-i) \setminus t \setminus i \leq i \setminus s \setminus (j-i) \cdot i \setminus i \setminus s \cdot j \setminus t \\
\iff & \{ \text{(6a) and (6b)} \} \\
& s \setminus j \cdot t \setminus (2 \times i) \cdot (j-i) \setminus t \setminus i \leq i \setminus s \setminus (j-i) \cdot (2 \times i) \setminus s \cdot j \setminus t \\
\iff & \{ \text{pointwise ordering (2)} \} \\
& s \setminus j \leq i \setminus s \setminus (j-i) \wedge t \setminus (2 \times i) \leq (2 \times i) \setminus s \wedge (j-i) \setminus t \setminus i \leq j \setminus t
\end{aligned}$$

The first and third conjunct are consequences of the fact that the input sequences s and t are ordered. The second conjunct, however, cannot be discharged. In other words, we have reached the pre-condition that we sought to derive. The circuit $clean_i$ maintains the invariant provided that its input satisfies the following pre-condition:

$$pre_i(s \setminus t) \iff t \setminus (2 \times i) \leq (2 \times i) \setminus s \quad (17)$$

5.3 Cleaning Up

We are now in a position to define the pair sorter. We have shown that the pre-condition (17) guarantees that the circuit $clean_i(s \setminus t)$ both maintains the invariant (15) and establishes the post-condition (16). Moreover, it is clear that $pre_i(s \setminus t) \iff post_{2 \times i}(s \setminus t)$ for $i \leq n/2$, so there is only one logical way to connect the cleaners:

$$\begin{aligned}
& pair\text{-}sort_n : A^{2 \times n} \rightarrow A^{2 \times n} \text{ where } n = 2^k \\
& pair\text{-}sort_1 = id \\
& pair\text{-}sort_{2 \times n} = clean_n ; pair\text{-}sort_n
\end{aligned}$$

The circuit $pair\text{-}sort_{2 \times n}$ maintains the invariant. Its vacuous pre-condition $pre_n(s \setminus t)$ ensures that it establishes the post-condition $post_1(s' \setminus t') \iff init\ t' \leq tail\ s'$. Hence the output $(s' \setminus t')$ is ordered, by the zig-zag property (14), as noted in Section 5.1.

The diagram below illustrates the resulting circuit for $n = 8$, including pre- and post-conditions as Hasse diagrams.

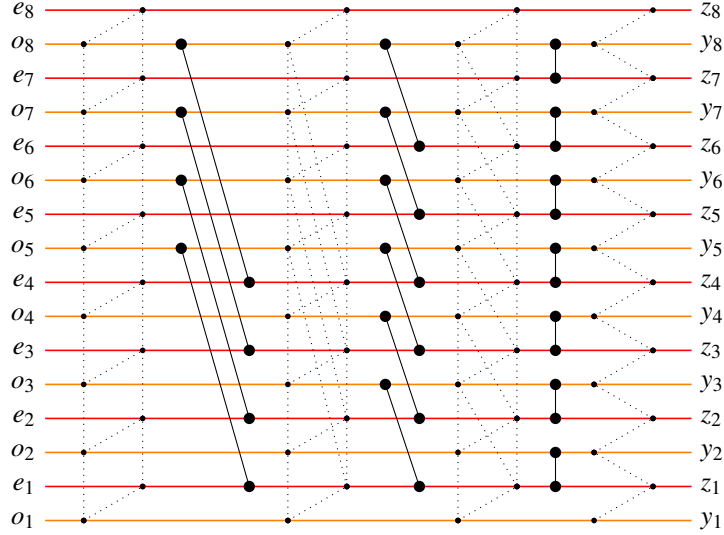


Fig. 2 displays Parberry’s pairwise sorting network for $n = 32$. You are invited to identify the three phases of the pairwise sorter. Interestingly, the sorter contains tree-shaped sub-circuits to determine the overall minimum (on the top) and the overall maximum (scattered in the middle) in $\Theta(\log n)$ time.

6 The Hidden Merger

Parberry’s construction places the internal sorter in front of the external sorter: $sort_{2 \times n} = isort_n ; esort_n ; pair-sort_n$. This arrangement is, however, not cast in stone. We can also swap the two phases without compromising the correctness of the sorter:

$$sort_{2 \times n} = esort_n ; isort_n ; pair-sort_n$$

Thus, the internal sorter jointly with the pair sorter implements a merger! Quite amazingly, the resulting circuit is almost identical to Batcher’s merger. The main difference is structural and concerns the “supply” of the two arguments: they are concatenated for Batcher and interleaved for Parberry. To make this concrete, consider the following “one-argument” version of Batcher’s merger, the specification of which, $bmerge(s \cdot t) = s \bowtie t$, unfolds to:

$$\begin{aligned} bmerge_1(\langle a \rangle \parallel \langle b \rangle) &= \langle a \downarrow b, a \uparrow b \rangle \\ bmerge_{2 \times n}((s \Upsilon t) \parallel (u \Upsilon v)) &= bmerge_n(s \parallel u) \downarrow bmerge_n(t \parallel v) \end{aligned}$$

Contrast this with the recursive counterpart of Parberry’s merger:

$$\begin{aligned} pmerge_1(\langle a \rangle \Upsilon \langle b \rangle) &= \langle a \downarrow b, a \uparrow b \rangle \\ pmerge_{2 \times n}((s \Upsilon t) \Upsilon (u \Upsilon v)) &= pmerge_n(s \Upsilon u) \downarrow pmerge_n(t \Upsilon v) \end{aligned}$$

which we claim is identical to the iterative implementation derived in Section 5:

$$pmerge_n = isort_n ; pair-sort_n \tag{18}$$

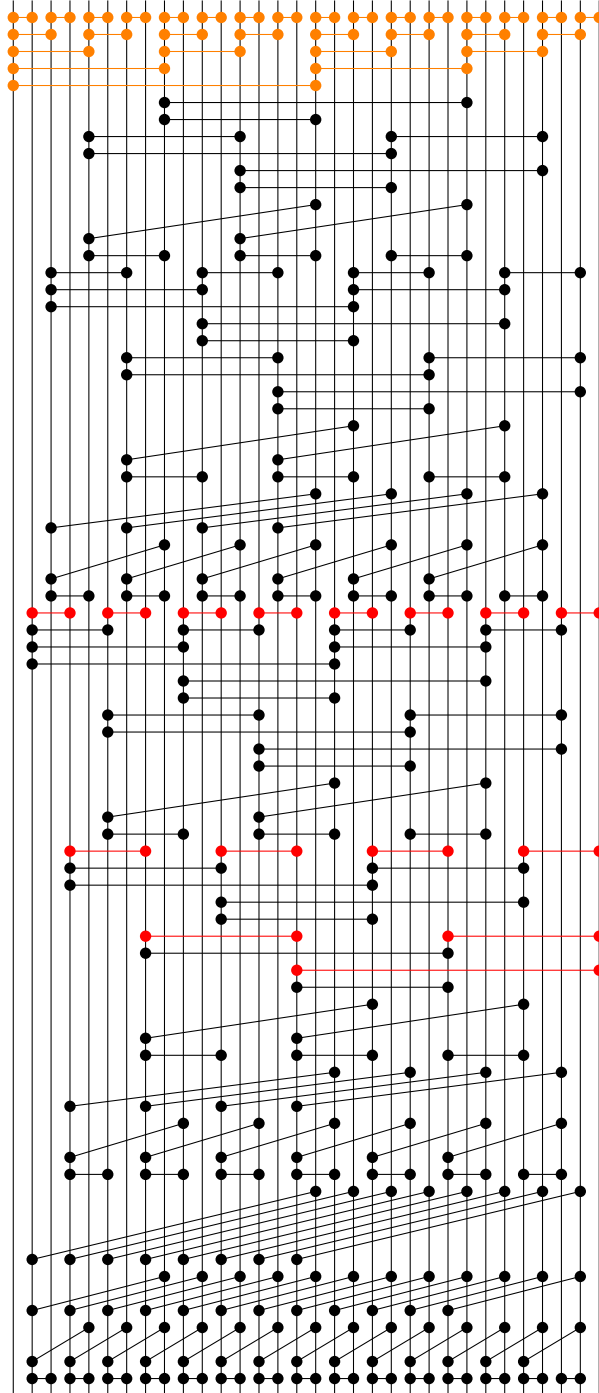


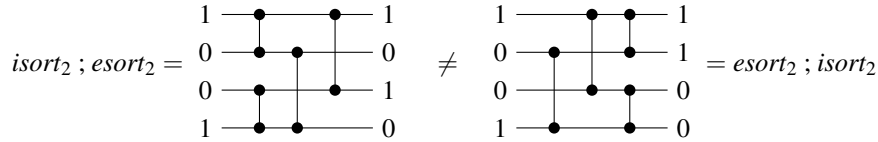
Fig. 2. Parberry's pairwise sorting network for $n = 32$.

$$\begin{array}{ll}
\text{sort}_1 = id & \text{sort}_1 = id \\
\text{sort}_{2 \times n} = \text{esort}_n ; \text{isort}_n ; \text{pair-sort}_n & \text{sort}_{2 \times n} = \text{esort}_n ; \text{pmerge}_n \\
\text{isort}_n = (\text{odd} \downarrow \text{even}) \Upsilon (\text{odd} \uparrow \text{even}) & \text{esort}_n = \text{sort}_n \star \text{sort}_n \\
\text{esort}_n = \text{sort}_n \star \text{sort}_n & \text{pmerge}_1 = \text{isort}_1 \\
\text{pair-sort}_1 = id & \text{pmerge}_{2 \times n} = ((\text{odd} \star \text{odd}) ; \text{pmerge}_n) \\
\text{pair-sort}_{2 \times n} = \text{clean}_n ; \text{pair-sort}_n & \quad \downarrow ((\text{even} \star \text{even}) ; \text{pmerge}_n)
\end{array}$$

Fig. 3. Iterative and recursive implementation of Parberry’s “hidden” sorting network.

So Parberry’s pairwise sorting network is obtained from Batchner’s “merge exchange” sorting network by swapping some components and suitably rearranging the inputs and outputs, using so-called bit-reversal permutations (Hinze, 2000).

Just in case you wonder, it is not the case that the two arrangements of internal and external sorter are equivalent. The following diagram provides a simple counterexample:



The two arrangements only establish the same post-condition, our invariant (15).

Up to this point we have exclusively conducted pointwise calculations. However, experience shows that for *structural* proofs a point-free argument is preferable. To this end we “functionalize” the operators we have seen before:

$$\begin{aligned}
(f \Upsilon g) x &= f x \Upsilon g x \\
(f \downarrow g) x &= f x \downarrow g x
\end{aligned}$$

For example, we have $\text{clean}_1 = \text{odd} \downarrow \text{even}$. We shall also need two projection functions as a substitute for the use of interleavings in patterns:

$$\begin{aligned}
\text{odd} (x \Upsilon y) &= x \\
\text{even} (x \Upsilon y) &= y
\end{aligned}$$

You may recognize the similarity to categorical products: Υ is pairing and odd and even are the projection functions. Indeed, using the ingredients above we can define the arrow part of a categorical product:

$$f \star g = (\text{odd} ; f) \Upsilon (\text{even} ; g)$$

The function f is applied to the odd sub-sequence and g is applied to the even sub-sequence, for example, $\text{esort}_n = \text{sort}_n \star \text{sort}_n$. For reference, Figure 3 lists the iterative and the recursive versions of Parberry’s “hidden” sorter in a point-free style.

The proof of (18) proceeds by induction on k , where $n = 2^k$.

Base $k = 0$: the proposition holds trivially as $\text{pair-sort}_1 = id$.

Step $k > 1$: we reason:

$$\text{pmerge}_{2 \times n}$$

$$\begin{aligned}
&= \{ \text{definition of } pmerge_{2 \times n} \} \\
&\quad ((odd \star odd) ; pmerge_n) \downarrow ((even \star even) ; pmerge_n) \\
&= \{ \text{induction assumption (18) and definition of } pair\text{-sort} \} \\
&\quad ((odd \star odd) ; isort_n ; clean_{n/2} ; \dots ; clean_1) \downarrow ((even \star even) ; isort_n ; clean_{n/2} ; \dots ; clean_1) \\
&= \{ \text{odd and even lemmata (19a)–(19d), see below} \} \\
&\quad (isort_{2 \times n} ; clean_n ; \dots ; clean_2 ; (odd \star odd)) \downarrow (isort_{2 \times n} ; clean_n ; \dots ; clean_2 ; (even \star even)) \\
&= \{ \text{fusion: } h ; (f \downarrow g) = (h ; f) \downarrow (h ; g) \} \\
&\quad isort_{2 \times n} ; clean_n ; \dots ; clean_2 ; ((odd \star odd) \downarrow (even \star even)) \\
&= \{ \text{crossing lemma (20), see below} \} \\
&\quad isort_{2 \times n} ; clean_n ; \dots ; clean_2 ; (odd \downarrow even) \\
&= \{ \text{definition of } clean_1 \} \\
&\quad isort_{2 \times n} ; clean_n ; \dots ; clean_2 ; clean_1 \\
&= \{ \text{definition of } pair\text{-sort} \} \\
&\quad isort_{2 \times n} ; pair\text{-sort}_{2 \times n}
\end{aligned}$$

The proof includes one or perhaps two non-obvious rewrites.

In the third step, we move $odd \star odd$ and $even \star even$ across the internal sorter and the pair sorter, making use of the following equalities:

$$(odd \star odd) ; isort_n = isort_{2 \times n} ; (odd \star odd) \quad (19a)$$

$$(even \star even) ; isort_n = isort_{2 \times n} ; (even \star even) \quad (19b)$$

$$(odd \star odd) ; clean_i = clean_{2 \times i} ; (odd \star odd) \quad (19c)$$

$$(even \star even) ; clean_i = clean_{2 \times i} ; (even \star even) \quad (19d)$$

The proofs of the *odd and even lemmata* are all fairly straightforward and left as exercises.

There is a tiny structural mismatch between the recursive and the iterative version of the hidden merger. In the second but last step, we have to counter for this, making use of the fact that minimum and maximum are commutative, captured in the *crossing lemma*:

$$odd \downarrow even = (odd \star odd) \downarrow (even \star even) \quad (20)$$

Notice that the isomorphism $(odd \star odd) \curlywedge (even \star even)$ swaps the values on the inner adjacent wires. Again, we leave the proof as an exercise to the reader.

7 Conclusion

This concludes our homage to the neglected algorithm of Parberry. It has been satisfying to reuse the idea from (Hinze & Martin, 2016) of motivating an algebraic derivation from the visual intuition of pre- and post-conditions: once again the diagrams and algebra work hand in hand. We have introduced a simple calculus of prefixes and suffixes to avoid the need for offset calculations, which we found fun to use. We hope to see further applications in the future.

The connection between the iterative and recursive mergers was originally observed by Codish and Zazon-Ivry (2010), where a relational style is used for the definitions.

We consider this to be an unnecessary level of complexity for a deterministic algorithm. The functional presentation used here shows instantly how the merger arises by simply swapping the first two phases. Another difference is that our point-free proof is much more succinct than the arguments given in (Codish & Zazon-Ivry, 2010) which rely heavily on the use of indexed sequences. However, the authors make the further point that both the cardinality networks (Codish & Zazon-Ivry, 2010) and the selection networks (Zazon-Ivry & Codish, 2012) corresponding to Parberry’s method are actually superior to those of Batchier. Further justification, if any is needed, for this unusual yet ultimately simple design.

The description of the pairwise sorter suggests an obvious generalization, where we have m parallel n -sorters followed by n parallel m -sorters. Can you devise a suitable n -tuple sorter to finish off the sort?

Acknowledgements

Many thanks are due to Roland Backhouse and Ian Bayley for suggesting numerous improvements regarding structure and presentation. In particular, Ian proposed some valuable amendments to the narrative in Section 5.1 and Roland questioned the use of subscripts, proposing an alternative presentation via catamorphisms. This is an intriguing recommendation, but not adopted since it is beyond the scope of this paper.

References

- Bove, A. and Coquand, T. (2006) Formalising bitonic sort in type theory. Filliâtre, J.-C., Paulin-Mohring, C. and Werner, B. (eds), *Types for Proofs and Programs*. Lecture Notes in Computer Science 3839, pp. 82–97. Springer Berlin Heidelberg.
- Codish, M. and Zazon-Ivry, M. (2010) Pairwise cardinality networks. *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. LPAR’10, pp. 154–172. Springer-Verlag.
- Hinze, R. (2000) Functional Pearl: Perfect trees and bit-reversal permutations **10**(3):305–317.
- Hinze, R. and Martin, C. (2016) *Batchier’s Odd-Even Merging Network Revealed*. in submission.
- Knuth, D. E. (1998) *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2nd edn.
- Parberry, I. (1992) The pairwise sorting network. *Parallel Processing Letters* **2**(2&3):205–211.
- Zazon-Ivry, M. and Codish, M. (2012) Pairwise networks are superior for selection. *Unpublished manuscript*.