JFP **28**, e21, 14 pages, 2018. © Cambridge University Press 2018
doi:10.1017/S0956796818000199

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, 2018), 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 unraveling 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, 2018), 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:

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

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

These properties imply, for example, that minimum, or meet, and maximum, or join, are monotonic. The ordering $\leqslant$ is lifted pointwise to sequences of the same length, $x \leqslant y \Leftrightarrow \mathit{and}\,(\mathit{zip}\,(\leqslant)\,x\,y)$. Here $\mathit{zip}\,f$ combines sequences of equal length with f while the function

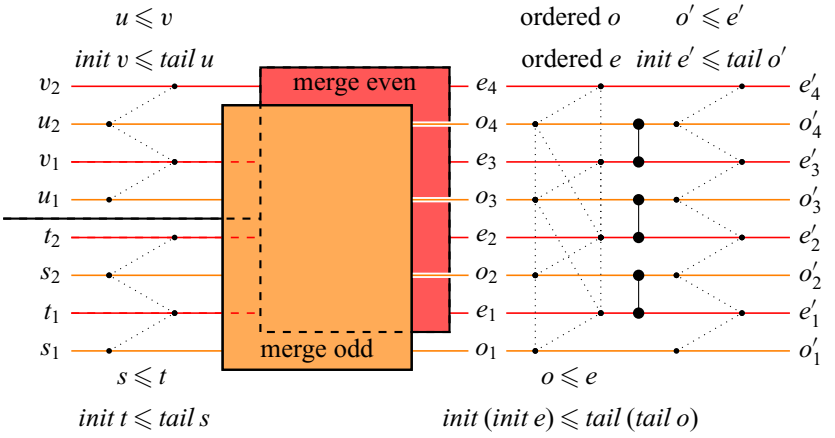


Fig. 1. Batcher's odd-even merging network.

and conjoins the result. Binary operators $\oplus$, such as minimum and maximum, are lifted similarly: $x \oplus y = zip\ (\oplus)\ x\ y$. Consequently, concatenation $(\cdot)$ is an order-embedding:

$$w \cdot x \leqslant y \cdot z \iff w \leqslant y \ \wedge \ x \leqslant z \tag{2}$$

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

$$(w \cdot x) \oplus (y \cdot z) = (w \oplus y) \cdot (x \oplus z) \tag{3}$$

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 $\between$ is merge):

$$\begin{aligned} &sort : A^n \to A^n \ \textbf{where}\ n = 2^k \\ &sort\ \langle a \rangle \quad\ = \langle a \rangle \\ &sort\ (x \parallel y) = sort\ x \between sort\ y \end{aligned}$$

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 ($\curlyvee$ is interleaving, so for example, $\langle 1,\ 2 \rangle \curlyvee \langle 3,\ 4 \rangle = \langle 1,\ 3,\ 2,\ 4 \rangle$, and $\wr$ is the *cleaner*, so-called because it performs the final "clean-up" phase of the merger):

$$\begin{aligned} &\langle a \rangle \between \langle b \rangle \qquad\quad = \langle a \downarrow b,\ a \uparrow b \rangle \\ &(s \curlyvee t) \between (u \curlyvee v) = (s \between u) \wr (t \between v) \\ &(\langle a \rangle \cdot x) \wr (y \cdot \langle b \rangle) = (\langle a \rangle \cdot (x \uparrow y)) \curlyvee ((x \downarrow y) \cdot \langle b \rangle) \end{aligned}$$

This recursive decomposition of the merger is illustrated in Figure 1, which merits careful study. As a reminder, the data flow from left to right; the perpendicular comparators ($\vdots$) sort the values on their two input wires into vertically ascending order. The figure

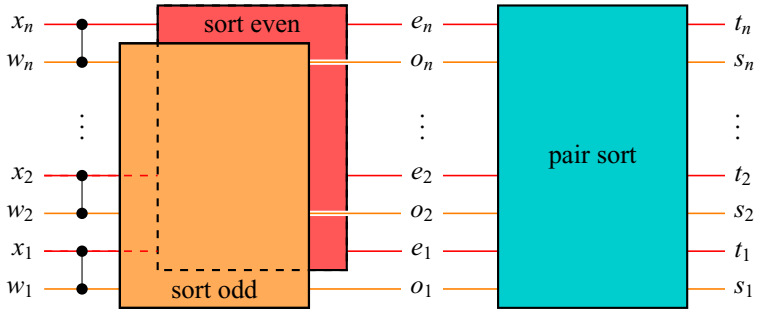
https://doi.org/10.1017/S0956796818000199 Published online by Cambridge University Press

additionally integrates Hasse diagrams (depicted by small discs • and dotted lines ⋰) 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 ↕, 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 disenchanted 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, 2018). 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 two 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 Batcher'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:

$$
\begin{aligned}
&\mathit{sort}_n : A^n \to A^n \ \textbf{where}\ n = 2^k \\
&\mathit{sort}_1 \quad\ = \mathit{id} \\
&\mathit{sort}_{2\times n} = \mathit{isort}_n \,;\, \mathit{esort}_n \,;\, \mathit{pair\text{-}sort}_n \\
&\mathit{isort}_n \ (w \curlyvee x) = (w \downarrow x) \curlyvee (w \uparrow x) \\
&\mathit{esort}_n \ (s \curlyvee t) \ = \mathit{sort}_n \ s \curlyvee \mathit{sort}_n \ t
\end{aligned}
$$

https://doi.org/10.1017/S0956796818000199 Published online by Cambridge University Press

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, 2018), which ensures that

$$y \leqslant z \implies sort_n\, y \leqslant sort_n\, z$$

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

$$o \text{ ordered} \quad \wedge \quad o \leqslant e \quad \wedge \quad e \text{ ordered} \tag{4}$$

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

$$init\,(init\,e) \leqslant tail\,(tail\,o) \tag{5}$$

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, *take i x*, and a negative to suffixes, *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 $\nearrow$ for *drop* and $\searrow$ for its dual:

$$\begin{array}{ll} (\nearrow) : (i : \{0 \;..\; n\}) \times A^n \to A^{n-i} & (\searrow) : A^n \times (i : \{0 \;..\; n\}) \to A^{n-i} \\ 0 \nearrow x \quad\quad\quad\quad = x & x \searrow 0 \quad\quad\quad\quad = x \\ (i+1) \nearrow (\langle a \rangle \cdot x) = i \nearrow x & (x \cdot \langle a \rangle) \searrow (i+1) = x \searrow i \end{array}$$

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

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

$$(x \searrow i) \searrow j = x \searrow (i+j) \tag{6a}$$

$$i \nearrow (j \nearrow x) = (i+j) \nearrow x \tag{6b}$$

$$(i \nearrow x) \searrow j = i \nearrow (x \searrow j) \tag{6c}$$

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

Both $(\searrow i)$ and $(i \nearrow)$ are *monotonic*:

$$x \leqslant y \implies x \searrow i \leqslant y \searrow i \tag{7a}$$

$$x \leqslant y \implies i \nearrow x \leqslant i \nearrow y \tag{7b}$$

https://doi.org/10.1017/S0956796818000199 Published online by Cambridge University Press

and they *distribute* over lifted operations:

$$(x \oplus y) \setminus i = (x \setminus i) \oplus (y \setminus i) \tag{8a}$$

$$i \mathbin{/} (x \oplus y) = (i \mathbin{/} x) \oplus (i \mathbin{/} y) \tag{8b}$$

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

$$i + j = n \iff x \setminus i \cdot j \mathbin{/} x = x \tag{9}$$

where x has length n and $0 \leqslant i, j \leqslant n$. 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]:

$$x \text{ ordered} \iff \forall k \,.\, x \setminus k \leqslant k \mathbin{/} x \tag{10}$$

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

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

$$x \setminus i \text{ ordered} \impliedby x \text{ ordered} \tag{11a}$$

$$i \mathbin{/} x \text{ ordered} \impliedby x \text{ ordered} \tag{11b}$$

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

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

Minimum and maximum also preserve ordered sequences:

$$x \downarrow y \text{ ordered} \impliedby x \text{ ordered} \;\wedge\; y \text{ ordered} \tag{12a}$$

$$x \uparrow y \text{ ordered} \impliedby x \text{ ordered} \;\wedge\; y \text{ ordered} \tag{12b}$$

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

$$x \cdot y \text{ ordered} \iff x \text{ ordered} \;\wedge\; \mathit{last}\, x \leqslant \mathit{head}\, y \;\wedge\; y \text{ ordered} \tag{13}$$

while non-empty interleavings satisfy the *zig-zag property* (Hinze & Martin, 2018), named after the corresponding Hasse diagram, as shown in Figure 1:

$$x \curlyvee y \text{ ordered} \iff x \leqslant y \;\wedge\; \mathit{init}\, y \leqslant \mathit{tail}\, x \tag{14}$$

[1] The previous pearl (Hinze & Martin, 2018) featured an alternative definition of "orderedness". The proof that the two definitions are equivalent is left as an instructive exercise to the reader.

https://doi.org/10.1017/S0956796818000199 Published online by Cambridge University Press

5 Deriving the pair sorter

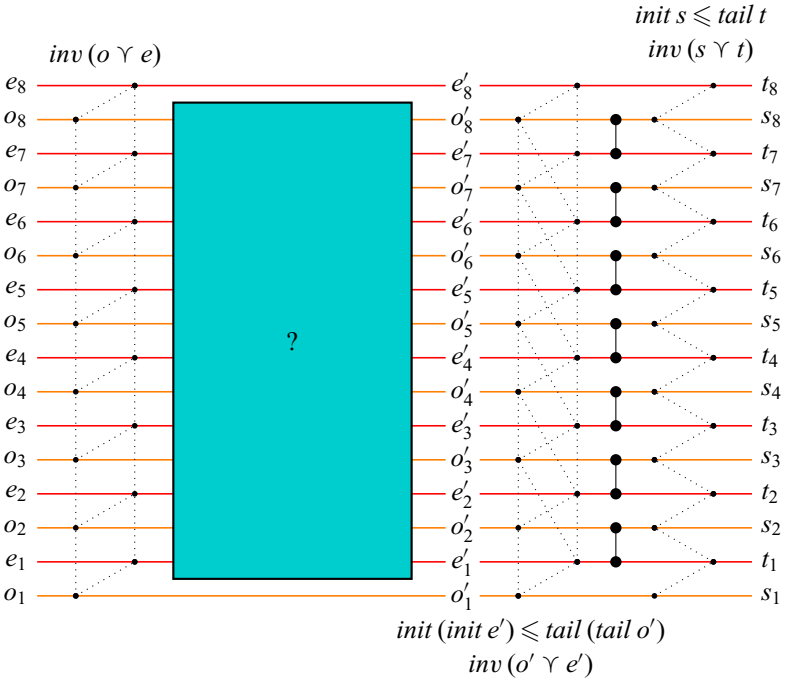
Equipped with this machinery we are now ready to derive the third component of Parberry's sorter. We shall maintain the post-condition of the first two phases (4) as an invariant:

$$inv\,(o \curlyvee e) \iff o \text{ ordered} \;\wedge\; o \leqslant e \;\wedge\; e \text{ ordered} \tag{15}$$

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 Figure 1. The invariant (15) is only missing the conjunct (5). This suggests that we might try to reuse 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 $init\,t \leqslant tail\,s$ (the zag) for the output $s \curlyvee t$ to be ordered, as the invariant guarantees that $s \leqslant t$ (the zig).



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

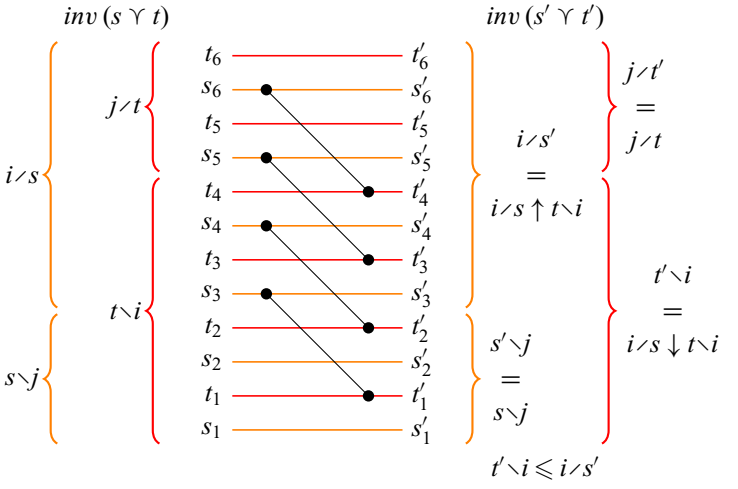
$$post_i\,(s' \curlyvee t') \iff t' \backslash i \leqslant i / s' \tag{16}$$

https://doi.org/10.1017/S0956796818000199 Published online by Cambridge University Press

where $1 \leqslant i \leqslant n$ 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

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

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, $clean_2\,(s \curlyvee t)$, we have $s' = s{\upharpoonleft}4 \cdot (2{\nearrow}s \uparrow t{\upharpoonleft}2)$ and $t' = (2{\nearrow}s \downarrow t{\upharpoonleft}2) \cdot 4{\nearrow}t$. This suggests the following generalized circuit:

$$\begin{aligned}&clean_i : A^{2\times n} \to A^{2\times n} \textbf{ where } n \geqslant 1\\ &clean_i\,(s \curlyvee t) = (s{\upharpoonleft}j \cdot (i{\nearrow}s \uparrow t{\upharpoonleft}i)) \curlyvee ((i{\nearrow}s \downarrow t{\upharpoonleft}i) \cdot j{\nearrow}t) \textbf{ where } i + j = n\end{aligned}$$

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

To summarize, the circuit $clean_i\,(s \curlyvee t)$ has been constructed to achieve the post-condition $t'{\upharpoonleft}i \leqslant i{\nearrow}s'$, 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 $(s' \curlyvee t')$ of the circuit $clean_i\,(s \curlyvee t)$ needs to satisfy the invariant $inv\,(s' \curlyvee t')$, defined in (15):

$$inv\,(s' \curlyvee t') \iff s' \text{ ordered} \;\wedge\; s' \leqslant t' \;\wedge\; t' \text{ ordered}$$

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

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

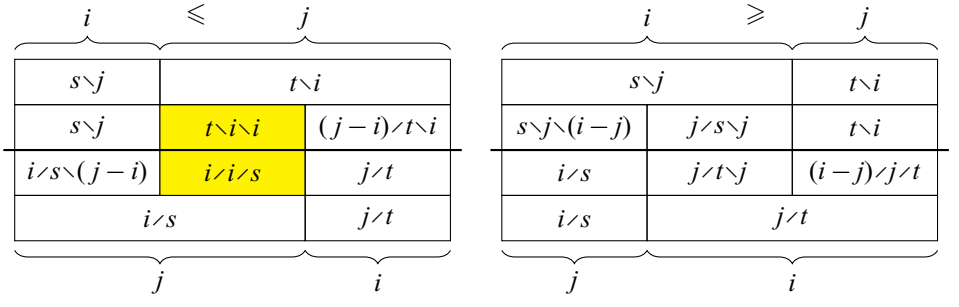
https://doi.org/10.1017/S0956796818000199 Published online by Cambridge University Press

$$
\begin{array}{ll}
\Longleftrightarrow & \{\ s \text{ and } t \text{ are ordered, } (k\swarrow),\ (\searrow k), \text{ and } \uparrow \text{ preserve orderedness} \\
& \quad [(11a)\text{–}(12b)]\} \\
& \mathit{last}\,(s\searrow j) \leqslant \mathit{head}\,(i\swarrow s \uparrow t\searrow i) \\
\Longleftrightarrow & \{\ \mathit{head} \text{ distributes over } \uparrow\ (8a)\ \} \\
& \mathit{last}\,(s\searrow j) \leqslant \mathit{head}\,(i\swarrow s) \uparrow \mathit{head}\,(t\searrow i) \\
\Longleftarrow & \{\ \text{transitivity and } a \leqslant a \uparrow b\ \} \\
& \mathit{last}\,(s\searrow j) \leqslant \mathit{head}\,(i\swarrow s) \\
\Longleftarrow & \{\ \text{link property } (13)\ \} \\
& s\searrow j \cdot i\swarrow s \text{ ordered} \\
\Longleftrightarrow & \{\ i+j=n, \text{ so } s = s\searrow j \cdot i\swarrow s\ \} \\
& s \text{ ordered}
\end{array}
$$

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

$$
\begin{array}{ll}
& s' \leqslant t' \\
\Longleftrightarrow & \{\ \text{definitions of } s' \text{ and } t'\ \} \\
& s\searrow j \cdot (i\swarrow s \uparrow t\searrow i) \leqslant (i\swarrow s \downarrow t\searrow i) \cdot j\swarrow t \\
\Longleftrightarrow & \{\ \downarrow \text{ and } \uparrow \text{ are idempotent}\ \} \\
& (s\searrow j \uparrow s\searrow j) \cdot (i\swarrow s \uparrow t\searrow i) \leqslant (i\swarrow s \downarrow t\searrow i) \cdot (j\swarrow t \downarrow j\swarrow t) \\
\Longleftrightarrow & \{\ \text{middle-two interchange law } (3)\ \} \\
& (s\searrow j \cdot i\swarrow s) \uparrow (s\searrow j \cdot t\searrow i) \leqslant (i\swarrow s \cdot j\swarrow t) \downarrow (t\searrow i \cdot j\swarrow t) \\
\Longleftrightarrow & \{\ \text{split property } (9) \text{ twice}\ \} \\
& s \uparrow (s\searrow j \cdot t\searrow i) \leqslant (i\swarrow s \cdot j\swarrow t) \downarrow t \\
\Longleftrightarrow & \{\ \text{minimum } (1a) \text{ and maximum } (1b)\ \} \\
& s \leqslant i\swarrow s \cdot j\swarrow t \ \wedge\ s \leqslant t \ \wedge\ s\searrow j \cdot t\searrow i \leqslant i\swarrow s \cdot j\swarrow t \ \wedge\ s\searrow j \cdot t\searrow i \leqslant t
\end{array}
$$

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 $s\searrow j \cdot t\searrow i \leqslant i\swarrow s \cdot j\swarrow t$ 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:



https://doi.org/10.1017/S0956796818000199 Published online by Cambridge University Press

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 $i \geqslant j$ **(**$\Longleftrightarrow 2 \times i \geqslant n$**):**

$$
\begin{array}{ll}
 & s \backslash j \cdot t \backslash i \leqslant i /\!\!^{\prime} s \cdot j /\!\!^{\prime} t \\
\Longleftrightarrow & \quad \{ \text{ split property (9), see diagram above, (6a) and (6b) } \} \\
 & s \backslash i \cdot j /\!\!^{\prime} s \backslash j \cdot t \backslash i \leqslant i /\!\!^{\prime} s \cdot j /\!\!^{\prime} t \backslash j \cdot i /\!\!^{\prime} t \\
\Longleftrightarrow & \quad \{ \text{ pointwise ordering (2) twice } \} \\
 & s \backslash i \leqslant i /\!\!^{\prime} s \ \wedge \ j /\!\!^{\prime} s \backslash j \leqslant j /\!\!^{\prime} t \backslash j \ \wedge \ t \backslash i \leqslant i /\!\!^{\prime} t \\
\Longleftarrow & \quad \{ \text{ definition ordered (10), } (k /\!\!^{\prime}) \text{ and } (\backslash k) \text{ are monotonic (7a) and (7b) } \} \\
 & s \text{ ordered } \ \wedge \ s \leqslant t \ \wedge \ t \text{ ordered}
\end{array}
$$

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

Case $i \leqslant j$ **(**$\Longleftrightarrow 2 \times i \leqslant n$**):** we reason

$$
\begin{array}{ll}
 & s \backslash j \cdot t \backslash i \leqslant i /\!\!^{\prime} s \cdot j /\!\!^{\prime} t \\
\Longleftrightarrow & \quad \{ \text{ split property (9), see diagram above } \} \\
 & s \backslash j \cdot t \backslash i \backslash i \cdot (j - i) /\!\!^{\prime} t \backslash i \leqslant i /\!\!^{\prime} s \backslash (j - i) \cdot i /\!\!^{\prime} i /\!\!^{\prime} s \cdot j /\!\!^{\prime} t \\
\Longleftrightarrow & \quad \{ \text{ (6a) and (6b) } \} \\
 & s \backslash j \cdot t \backslash (2 \times i) \cdot (j - i) /\!\!^{\prime} t \backslash i \leqslant i /\!\!^{\prime} s \backslash (j - i) \cdot (2 \times i) /\!\!^{\prime} s \cdot j /\!\!^{\prime} t \\
\Longleftrightarrow & \quad \{ \text{ pointwise ordering (2) } \} \\
 & s \backslash j \leqslant i /\!\!^{\prime} s \backslash (j - i) \ \wedge \ t \backslash (2 \times i) \leqslant (2 \times i) /\!\!^{\prime} s \ \wedge \ (j - i) /\!\!^{\prime} t \backslash i \leqslant j /\!\!^{\prime} t
\end{array}
$$

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:

$$pre_i\,(s \curlyvee t) \iff t \backslash (2 \times i) \leqslant (2 \times i) /\!\!^{\prime} s \tag{17}$$

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 $clean_i\,(s \curlyvee t)$ both maintains the invariant (15) and establishes the post-condition (16). Moreover, it is clear that $pre_i\,(s \curlyvee t) \iff post_{2\times i}\,(s \curlyvee t)$ for $i \leqslant n \,/\, 2$, so there is only one logical way to connect the cleaners:

$$
\begin{array}{l}
pair\text{-}sort_n : A^{2\times n} \to A^{2\times n} \textbf{ where } n = 2^k \\
pair\text{-}sort_1 \ \ = id \\
pair\text{-}sort_{2\times n} = clean_n \,;\, pair\text{-}sort_n
\end{array}
$$

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

https://doi.org/10.1017/S0956796818000199 Published online by Cambridge University Press

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

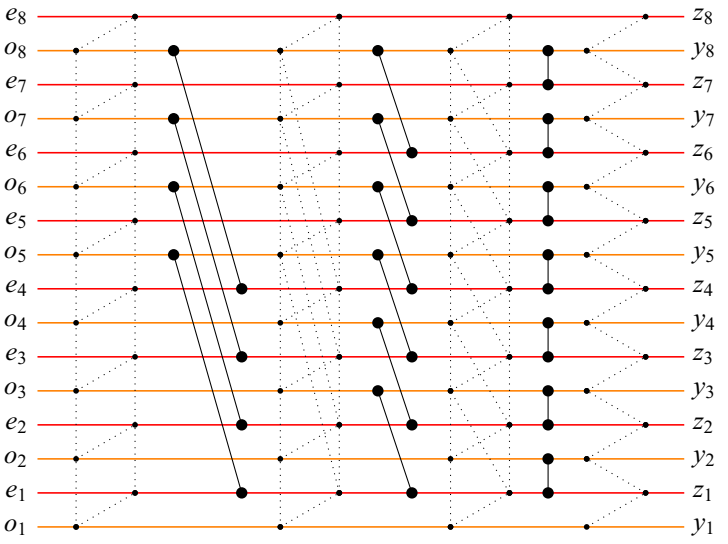


Figure 2 displays Parberry's pairwise sorting network for $n = 32$. You are invited to identify the three phases of the pairwise sorter. Interestingly, the sorter contains tree-shaped sub-circuits to determine the overall minimum (on the top) and the overall maximum (scattered in the middle) in $\Theta(\log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

6 The hidden merger

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

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

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, $bmerge\,(s \cdot t) = s \mathbin{/\!\!\backslash} t$, unfolds to:

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

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

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

https://doi.org/10.1017/S0956796818000199 Published online by Cambridge University Press

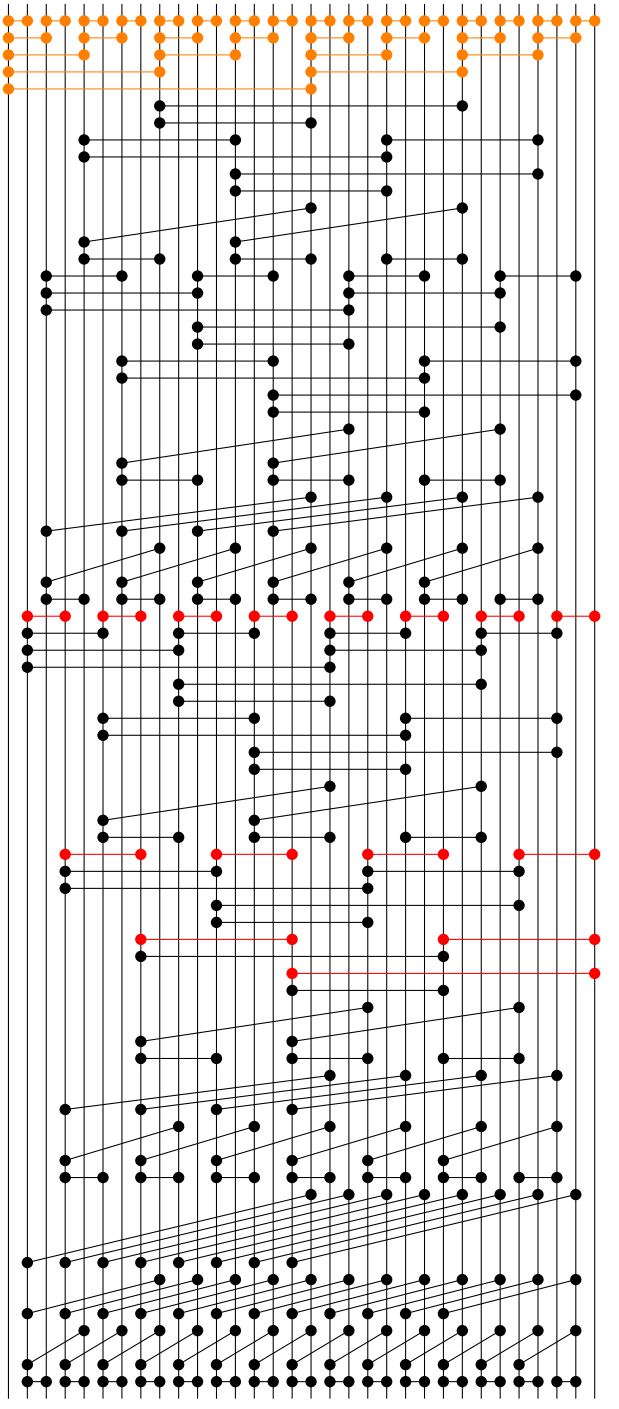


Fig. 2. Parberry's pairwise sorting network for $n = 32$.

https://doi.org/10.1017/S0956796818000199 Published online by Cambridge University Press

$$
\begin{aligned}
&sort_1 = id\\
&sort_{2\times n} = esort_n \,;\, isort_n \,;\, pair\text{-}sort_n\\
&isort_n = (odd \downarrow even) \curlyvee (odd \uparrow even)\\
&esort_n = sort_n \star sort_n\\
&pair\text{-}sort_1 = id\\
&pair\text{-}sort_{2\times n} = clean_n \,;\, pair\text{-}sort_n
\end{aligned}
\qquad
\begin{aligned}
&sort_1 = id\\
&sort_{2\times n} = esort_n \,;\, pmerge_n\\
&esort_n = sort_n \star sort_n\\
&pmerge_1 = isort_1\\
&pmerge_{2\times n} = ((odd \star odd) \,;\, pmerge_n)\\
&\qquad\qquad \curlyvee ((even \star even) \,;\, pmerge_n)
\end{aligned}
$$

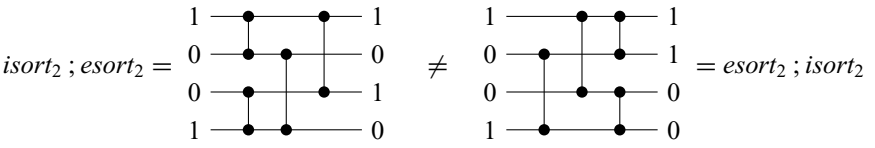
Fig. 3. Iterative and recursive implementation of Parberry's "hidden" sorting network.

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

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

So, Parberry's pairwise sorting network is obtained from Batcher'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:

$$
\begin{aligned}
&(f \curlyvee g)\, x = f\, x \curlyvee g\, x\\
&(f \wr g)\; x = f\, x \wr g\, x
\end{aligned}
$$

For example, we have $clean_1 = odd \wr even$. We shall also need two projection functions as a substitute for the use of interleavings in patterns:

$$
\begin{aligned}
&odd\;\; (x \curlyvee y) = x\\
&even\, (x \curlyvee y) = y
\end{aligned}
$$

You may recognize the similarity to categorical products: $\curlyvee$ 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:

$$f \star g = (odd \,;f) \curlyvee (even \,;g)$$

The function f is applied to the odd sub-sequence and g is applied to the even sub-sequence, for example, $esort_n = sort_n \star sort_n$. 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 $n = 2^k$.

https://doi.org/10.1017/S0956796818000199 Published online by Cambridge University Press

Base $k = 0$**:** the proposition holds trivially as $\mathit{pair\text{-}sort}_1 = \mathit{id}$.
Step $k > 1$**:** we reason

$$
\begin{aligned}
& \mathit{pmerge}_{2\times n} \\
= \quad & \{ \text{ definition of } \mathit{pmerge}_{2\times n} \} \\
& ((\mathit{odd} \star \mathit{odd}) \,;\, \mathit{pmerge}_n) \wr ((\mathit{even} \star \mathit{even}) \,;\, \mathit{pmerge}_n) \\
= \quad & \{ \text{ induction assumption (18) and definition of } \mathit{pair\text{-}sort} \} \\
& ((\mathit{odd} \star \mathit{odd}) \,;\, \mathit{isort}_n \,;\, \mathit{clean}_{n/2} \,;\, \ldots \,;\, \mathit{clean}_1) \wr ((\mathit{even} \star \mathit{even}) \,;\, \mathit{isort}_n \,;\, \mathit{clean}_{n/2} \,;\, \ldots \,;\, \mathit{clean}_1) \\
= \quad & \{ \text{ odd and even lemmata (19a)–(19d), see below } \} \\
& (\mathit{isort}_{2\times n} \,;\, \mathit{clean}_n \,;\, \ldots \,;\, \mathit{clean}_2 \,;\, (\mathit{odd} \star \mathit{odd})) \wr (\mathit{isort}_{2\times n} \,;\, \mathit{clean}_n \,;\, \ldots \,;\, \mathit{clean}_2 \,;\, (\mathit{even} \star \mathit{even})) \\
= \quad & \{ \text{ fusion: } h \,;\, (f \wr g) = (h \,;\, f) \wr (h \,;\, g) \} \\
& \mathit{isort}_{2\times n} \,;\, \mathit{clean}_n \,;\, \ldots \,;\, \mathit{clean}_2 \,;\, ((\mathit{odd} \star \mathit{odd}) \wr (\mathit{even} \star \mathit{even})) \\
= \quad & \{ \text{ crossing lemma (20), see below } \} \\
& \mathit{isort}_{2\times n} \,;\, \mathit{clean}_n \,;\, \ldots \,;\, \mathit{clean}_2 \,;\, (\mathit{odd} \wr \mathit{even}) \\
= \quad & \{ \text{ definition of } \mathit{clean}_1 \} \\
& \mathit{isort}_{2\times n} \,;\, \mathit{clean}_n \,;\, \ldots \,;\, \mathit{clean}_2 \,;\, \mathit{clean}_1 \\
= \quad & \{ \text{ definition of } \mathit{pair\text{-}sort} \} \\
& \mathit{isort}_{2\times n} \,;\, \mathit{pair\text{-}sort}_{2\times n}
\end{aligned}
$$

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

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

$$(\mathit{odd} \star \mathit{odd}) \,;\, \mathit{isort}_n = \mathit{isort}_{2\times n} \,;\, (\mathit{odd} \star \mathit{odd}) \tag{19a}$$

$$(\mathit{even} \star \mathit{even}) \,;\, \mathit{isort}_n = \mathit{isort}_{2\times n} \,;\, (\mathit{even} \star \mathit{even}) \tag{19b}$$

$$(\mathit{odd} \star \mathit{odd}) \,;\, \mathit{clean}_i = \mathit{clean}_{2\times i} \,;\, (\mathit{odd} \star \mathit{odd}) \tag{19c}$$

$$(\mathit{even} \star \mathit{even}) \,;\, \mathit{clean}_i = \mathit{clean}_{2\times i} \,;\, (\mathit{even} \star \mathit{even}) \tag{19d}$$

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*:

$$\mathit{odd} \wr \mathit{even} = (\mathit{odd} \star \mathit{odd}) \wr (\mathit{even} \star \mathit{even}) \tag{20}$$

Notice that the isomorphism $(\mathit{odd} \star \mathit{odd}) \curlyvee (\mathit{even} \star \mathit{even})$ 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 (2018) 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

https://doi.org/10.1017/S0956796818000199 Published online by Cambridge University Press

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 & 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, unpublished data)] corresponding to Parberry's method are actually superior to those of Batcher. 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. & Coquand, T. (2006) Formalising bitonic sort in type theory. In *Types for Proofs and Programs*, Filliâtre, J.-C., Paulin-Mohring, C. & Werner, B. (eds), Lecture Notes in Computer Science, vol. 3839. Berlin, Heidelberg: Springer, pp. 82–97.

Codish, M. & Zazon-Ivry, M. (2010) Pairwise cardinality networks. In Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2010. Berlin, Heidelberg: Springer-Verlag, pp. 154–172.

Hinze, R. (2000) Functional Pearl: Perfect trees and bit-reversal permutations. *J. Funct. Program.* **10**(3), 305–317.

Hinze, R. & Martin, C. (2018) Functional Pearl: Batcher's odd-even merging network revealed. *J. Funct. Program.* **28**(e14), 1–13.

Knuth, D. E. (1998) *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd edn. Reading, MA: Addison-Wesley Publishing Company.

Parberry, I. (1992) The pairwise sorting network. *Parallel Process. Lett.* **2**(2&3), 205–211.