Journal of Universal Computer Science, vol. 23, no. 1 (2017), 21-41
submitted: 29/6/16, accepted: 3/12/16, appeared: 28/1/17 © J.UCS

A Logical Reconstruction of Batcher's Mergers *Or:* Bitonicity is a Red Herring

Ralf Hinze
(Institute for Computing and Information Sciences
Radboud University, 6525EC Nijmegen, The Netherlands
ralf@cs.ru.nl)

Clare Martin
(Department of Computing and Communication Technologies
Oxford Brookes University, Wheatley, Oxford, OX33 1HX, England
cemartin@brookes.ac.uk)

Abstract: Almost half a century after Batcher wrote his seminal paper on sorting networks, we revisit the key algorithmic design decisions for oblivious merging to rediscover his schemes in a disciplined way. The design space of sorting networks is explored, resulting in a systematic reconstruction of schemes that appear in the literature in various guises.
Key Words: Hardware design, functional programming, parallel algorithms
Category: D3.2, D3.3

1 Introduction

To iterate is human, to recurse divine.
L Peter Deutsch

In 1968, K.E. Batcher introduced two related schemes for merging networks: the bitonic merger and the merge exchange network. The former method requires more comparisons but appears to be the more popular of the two [Batcher 1968]. Both are instances of comparison networks, which can only perform comparisons between single pairs of inputs, but these operations can be executed in parallel. The concept was originally devised for describing hardware implementations in the 1950's [Knuth 1998, OConnor and Nelson 1962] but is equally applicable to parallel algorithms on multiple processors. The model has a rich underlying theory and has become an active topic of research since its introduction. Procedures presented in this model are *oblivious* in the sense that the comparisons they perform are independent of the input data.

The aim of this paper is to explore the design space for oblivious merging systematically, starting from minimal assumptions. So, whilst most other formal treatments of these algorithms have focused on correctness proofs, the emphasis here is on derivation from first principles, with simultaneous verification. The result is a comprehensive overview of various schemes and presentations, with the connections between them made precise.

In particular, our contribution is to show explicitly how the two mergers of Batcher are derived by similar calculations from identical assumptions, differing only in *one* initial design decision. The effect of using different strategies to divide the input are also explored and used to relate various rearrangements of Batcher's methods. Correctness proofs are often expressed in terms of the *zero-one principle* [Knuth 1998]. The style adopted here favours the fundamental monotonicity property of comparison networks instead, as the resulting proofs and derivations are more enlightening. On a related note, we also find Batcher's notion of bitonicity dispensable—both mergers can be derived using monotonicity alone. In summary, we show how the remarkable intuition of Batcher can be explained in a methodical way through the judicious use of algebraic manipulation.

Batcher's mergers have been presented in a variety of sometimes competing styles. These can be roughly categorized along the following dimensions: algebraic versus diagrammatic, imperative versus functional, recursive versus iterative, and sequential versus parallel. For this enterprise—deriving the mergers from first principles—we found that an algebraic or functional style using recursion equations as implementable specifications of the algorithms worked best. This practice, now so routine for functional programmers, is part of the abundant legacy of David Turner [Turner 1982]. His language designs, SASL, KRC, and Miranda, emphasized elegance and clarity. We hope that our derivations are in his spirit, conveying the beauty of the equational approach to algorithm design.

We begin with some preliminary notation and background in Section 2. Then we explore the design space for oblivious merging in Sections 3 and 4, deriving two mergers in the process. We claim that the first is Batcher's bitonic merger, and Sections 5 and 6 are devoted to proving this claim. The second merger is instantly recognizable from previous work [Hinze and Martin 2017a] as the merge exchange scheme. Building on these mergers, Section 7 explores the design space for oblivious sorting and relates the resulting schemes to the literature. Section 8 continues to review related work and, finally, Section 9 concludes.

2 Preliminaries

2.1 Lattices and order

Sorting networks work well if the underlying structure is a distributive lattice instead of a total order. The output is not necessarily a permutation of the input in this setting, but it is still an ordered, generalized permutation [Bove and Coquand 2006]. We assume only the existence of a fixed partial order $\leqslant$

and lattice operators meet $\downarrow$ and join $\uparrow$, which distribute over each other and are defined by:

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

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

If a and b are comparable, then these operators are simply the minimum and maximum. It is also useful to postulate that our lattices are bounded by a bottom element $\bot$ and a top element $\top$:

$$\bot \leqslant a \wedge a \leqslant \top \tag{2}$$

Algebraically, bottom is the unit of join and, dually, top is the unit of meet.

2.2 Sequences

We write sequences using angle brackets, e.g. $\langle 1 \rangle$ is a singleton sequence and $\langle 1, 3, 1, 2, 6, 4 \rangle$ is a sequence of length 6. For readability, we shall omit the angle brackets on the singleton sequences $\langle \bot \rangle$ and $\langle \top \rangle$. We construct sequences in two ways: using concatenation and interleaving. In the former case, we have:

$$\langle x_0, \ldots, x_{m-1} \rangle \cdot \langle y_0, \ldots, y_{n-1} \rangle = \langle x_0, \ldots, x_{m-1}, y_0, \ldots, y_{n-1} \rangle$$

For emphasis, we write $x \parallel y$ for $x \cdot y$ if both arguments have the same length (or, are implicitly required to have the same length). Interleaving is defined for equal-length arguments only:

$$\langle x_0, \ldots, x_{n-1} \rangle \curlyvee \langle y_0, \ldots, y_{n-1} \rangle = \langle x_0, y_0, \ldots, x_{n-1}, y_{n-1} \rangle$$

Together, these operators satisfy the following interchange law:

$$(u \parallel v) \curlyvee (x \parallel y) = (u \curlyvee x) \parallel (v \curlyvee y) \tag{3}$$

We also use both $\parallel$ and $\curlyvee$ as patterns on the left-hand side of definitions: $x \parallel y$ is halving, dividing an input sequence into a lower half x and an upper half y, while $x \curlyvee y$ is uninterleaving, unzipping the input into the sub-sequence x of elements at even positions and the sub-sequence y of elements at odd positions (assuming sequence indexing starts from 0 not 1).

The first element of a sequence is denoted by *head*, and dually for the *last* element, with *tail* and *init* the remaining parts respectively:

$$\langle \mathit{head}\, x \rangle \cdot \mathit{tail}\, x = x = \mathit{init}\, x \cdot \langle \mathit{last}\, x \rangle$$

For convenience, the lifting of binary relations, such as $\leqslant$, and functions, like $\downarrow$ and $\uparrow$, to sequences, is denoted by the same symbol as the original:

$$\langle x_0, \ldots, x_{n-1} \rangle \mathrel{R} \langle y_0, \ldots, y_{n-1} \rangle \iff x_0 \mathrel{R} y_0 \wedge \cdots \wedge x_{n-1} \mathrel{R} y_{n-1}$$
$$\langle x_0, \ldots, x_{n-1} \rangle \oplus \langle y_0, \ldots, y_{n-1} \rangle = \langle x_0 \oplus y_0, \ldots, x_{n-1} \oplus y_{n-1} \rangle$$

Liftings interact with concatenation according to another interchange law:

$$(u \oplus v) \cdot (x \oplus y) = (u \cdot x) \oplus (v \cdot y) \tag{4}$$

provided that u has the same length as v, and x has the same length as y.

2.3 Ordered sequences

We can use the lifted order to capture that a sequence is *ordered*:

$$x \text{ ordered} \iff \bot \cdot x \leqslant x \cdot \top \tag{5}$$

Notice that if x is non-empty, then condition (5) is equivalent to $\mathit{init}\, x \leqslant \mathit{tail}\, x$.
Concatenations of non-empty sequences enjoy the *split property*:

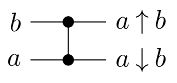
$$x \cdot y \text{ ordered} \iff x \text{ ordered} \wedge \mathit{last}\, x \leqslant \mathit{head}\, y \wedge y \text{ ordered} \tag{6}$$

while interleavings enjoy the *zig-zag property*, so-called because of its pictorial representation, see [Hinze and Martin 2017a]:

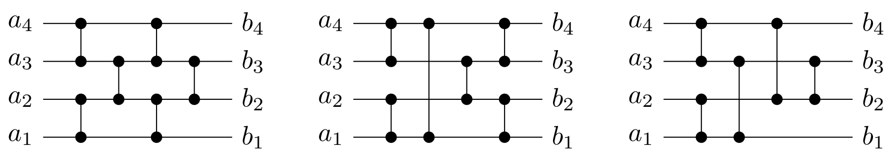
$$x \curlyvee y \text{ ordered} \iff x \leqslant y \wedge \bot \cdot y \leqslant x \cdot \top \tag{7}$$

2.4 Comparison Networks

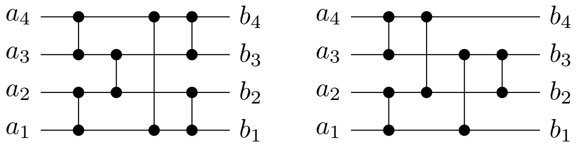
Merging and sorting networks are *data oblivious* algorithms in the sense that the comparisons used to merge or sort a sequence are the same regardless of the input data. Both are instances of comparison networks. The primary component of such a network is a comparator, depicted as a box by Batcher, but more commonly as a so-called Knuth diagram like that below, where horizontal lines represent wires, and comparators are vertical connections. Data flows from left to right along the wires. Each comparator outputs the smaller of its two input values on the lower wire, and the larger value on the upper one:



In general, a comparison network consists of a number of horizontal wires, which are vertically connected using comparators. Below are some examples of small sorting networks for four inputs $\langle a_1, a_2, a_3, a_4\rangle$:

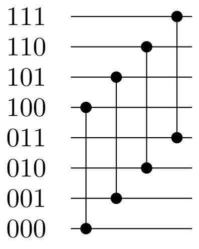


Note that many of the comparators can operate in parallel. Consider the second diagram. From the layout we can easily infer that the first two and the last two comparators can act in parallel. Perhaps less obviously, the same holds for the two comparators in the middle. They are only drawn beside each other to avoid overlapping wires. The diagrams below are alternative drawings of the second and third network.



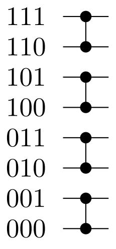
We identify two diagrams if one can be transformed into the other by sliding comparators horizontally, without moving connectors (depicted by filled circles) past each other.

To illustrate the use of the combinators introduced in Section 2.2, we define two fundamental building blocks of merging and sorting networks: low-high cleaners (also known as half-cleaners) and even-odd cleaners.



$$\textit{low-high–clean}\,(x \parallel y) = (x \downarrow y) \parallel (x \uparrow y)$$

$$\textit{even-odd-clean}\,(x \curlyvee y) = (x \downarrow y) \curlyvee (x \uparrow y)$$



Notice that the two cleaners are in a sense dual to each other: numbering the inputs in binary (most significant digit first), the low-high cleaner connects inputs $0w$ and $1w$, while the even-odd cleaner connects inputs $w0$ and $w1$.

Comparator networks enjoy a fundamental monotonicity property [Knuth 1998, Hinze and Martin 2017a] that is central to all of our correctness proofs: every network nw transforms greater inputs to greater outputs.

$$x \leqslant y \implies nw\,x \leqslant nw\,y \tag{8}$$

3 Deriving Batcher's Bitonic Merger

Given these prerequisites, let us now explore the design space for oblivious merging, starting from first principles. A merging network $\mathbin{/\!\!\wedge\!\!\backslash}$ takes two ordered inputs to an ordered output, as formalized by the specification:

$$x \text{ ordered} \wedge y \text{ ordered} \implies x \mathbin{/\!\!\wedge\!\!\backslash} y \text{ ordered} \tag{9}$$

Although this does not state explicitly that the output is a permutation of the two inputs, any network consisting solely of comparators does indeed produce a permutation in a generalized sense [Bove and Coquand 2006].

Now, applying an inductive approach to problem solving, the first question is how to reduce a given merging problem to simpler sub-problems. For simplicity, we restrict attention to input sequences with length an exact power of two. There are many ways to divide the input, including two principled approaches: each sequence can either be halved or uninterleaved. We choose the latter option, as the former seems to go nowhere; roughly speaking, the zig-zag property (7) is more pleasant to use than the split property (6). This may also explain the observation [Misra 1994] that most discussions of the principles of parallel sorting prefer interleaving to halving. So, the first assumption is that the inputs to be merged are treated as interleaved sequences, say $s \curlyvee t$ and $u \curlyvee v$.

The next choice is how to combine the sub-sequences in the recursive calls. There are two options: given the inputs $s \curlyvee t$ and $u \curlyvee v$, we can either recursively combine s with u or with v, partnering t with the remainder. We pursue the first route in Section 4; the second is explored below. We obtain as an initial sketch:

$$\begin{aligned}\langle a\rangle \mathbin{\wedge\!\!\wedge} \langle b\rangle &= \langle a \downarrow b, a \uparrow b\rangle\\ (s \curlyvee t) \mathbin{\wedge\!\!\wedge} (u \curlyvee v) &= (s \mathbin{\wedge\!\!\wedge} v) \mathbin{?} (t \mathbin{\wedge\!\!\wedge} u)\end{aligned}$$

The base case is uncontentious: if both input sequences are singletons, then the network is a single comparator. For the inductive step, we observe that the merge of interleavings is not quite the same as the interleaving of the merges, for example, $(\langle 3\rangle \curlyvee \langle 4\rangle) \mathbin{\wedge\!\!\wedge} (\langle 1\rangle \curlyvee \langle 2\rangle) \neq (\langle 3\rangle \mathbin{\wedge\!\!\wedge} \langle 2\rangle) \curlyvee (\langle 4\rangle \mathbin{\wedge\!\!\wedge} \langle 1\rangle)$. In other words, the placeholder '?' above is not just $\curlyvee$. But perhaps we can clean things up using a few additional comparators? The goal of the subsequent calculations is to derive the additional circuitry denoted by '?'.

We can assume that the recursive invocations are proper mergers. In particular, we make use of the fact that their output is ordered (10b), and that bottom and top elements are propagated to the front and to the rear respectively (10c). For reference, the pre-condition of (9) is also reiterated in (10a):

$$s \curlyvee t \text{ ordered} \;\wedge\; u \curlyvee v \text{ ordered} \tag{10a}$$

$$s \mathbin{\wedge\!\!\wedge} v \text{ ordered} \;\wedge\; t \mathbin{\wedge\!\!\wedge} u \text{ ordered} \tag{10b}$$

$$(\bot \cdot w) \mathbin{\wedge\!\!\wedge} z = \bot \cdot (w \mathbin{\wedge\!\!\wedge} z) \;\wedge\; (w \cdot \top) \mathbin{\wedge\!\!\wedge} z = (w \mathbin{\wedge\!\!\wedge} z) \cdot \top \tag{10c}$$

for all ordered sequences w and z. In case you find the last property unmotivated, recall that the definition of 'x ordered' (5) involves both bottom $\bot$ and top $\top$. Property (10c) makes precise how merge interacts with them.

Since the arguments of merge are given as interleavings, we begin by applying the zig-zag property (7) to the pre-condition of merge (10a):

$$s \curlyvee t \text{ ordered} \;\wedge\; u \curlyvee v \text{ ordered}$$

$$
\begin{array}{ll}
\iff & \{\ \text{zig-zag property (twice) (7)}\ \} \\
 & s \leqslant t \ \wedge\ \bot \cdot v \leqslant u \cdot \top \ \wedge\ \bot \cdot t \leqslant s \cdot \top \ \wedge\ u \leqslant v \\
\Longrightarrow & \{\ \text{monotonicity of comparison networks (twice) (8)}\ \} \\
 & s \mathbin{/\!\!\!\backslash\!\!\backslash} (\bot \cdot v) \leqslant t \mathbin{/\!\!\!\backslash\!\!\backslash} (u \cdot \top) \ \wedge\ (\bot \cdot t) \mathbin{/\!\!\!\backslash\!\!\backslash} u \leqslant (s \cdot \top) \mathbin{/\!\!\!\backslash\!\!\backslash} v \\
\iff & \{\ \text{property of merge (10c)}\ \} \\
 & \bot \cdot (s \mathbin{/\!\!\!\backslash\!\!\backslash} v) \leqslant (t \mathbin{/\!\!\!\backslash\!\!\backslash} u) \cdot \top \ \wedge\ \bot \cdot (t \mathbin{/\!\!\!\backslash\!\!\backslash} u) \leqslant (s \mathbin{/\!\!\!\backslash\!\!\backslash} v) \cdot \top
\end{array}
$$

Abbreviating $s \mathbin{/\!\!\!\backslash\!\!\backslash} v$ by x and $t \mathbin{/\!\!\!\backslash\!\!\backslash} u$ by y, our goal is to establish '$x \mathbin{?} y$ ordered', deriving the placeholder '?' in the process. It stands to reason that the unknown circuitry is some interleaving; so we work towards a situation where we can apply the zig-zag property (7). Introducing the induction assumption (10b), we continue:

$$
\begin{array}{ll}
 & x \text{ ordered} \ \wedge\ \bot \cdot x \leqslant y \cdot \top \ \wedge\ \bot \cdot y \leqslant x \cdot \top \ \wedge\ y \text{ ordered} \\
\iff & \{\ \text{definition of ordered (twice) (5)}\ \} \\
 & \bot \cdot x \leqslant x \cdot \top \ \wedge\ \bot \cdot x \leqslant y \cdot \top \ \wedge\ \bot \cdot y \leqslant x \cdot \top \ \wedge\ \bot \cdot y \leqslant y \cdot \top \\
\iff & \{\ \text{characterization of minimum (1a) and maximum (1b)}\ \} \\
 & (\bot \cdot x) \uparrow (\bot \cdot y) \leqslant (x \cdot \top) \downarrow (y \cdot \top) \\
\iff & \{\ \text{interchange law (4)}\ \} \\
 & (\bot \uparrow \bot) \cdot (x \uparrow y) \leqslant (x \downarrow y) \cdot (\top \downarrow \top) \\
\iff & \{\ \text{idempotence: } a \downarrow a = a \text{ and } a \uparrow a = a\ \} \\
 & \bot \cdot (x \uparrow y) \leqslant (x \downarrow y) \cdot \top \\
\iff & \{\ \text{minimum is smaller than maximum: } a \downarrow b \leqslant a \uparrow b\ \} \\
 & x \downarrow y \leqslant x \uparrow y \ \wedge\ \bot \cdot (x \uparrow y) \leqslant (x \downarrow y) \cdot \top \\
\iff & \{\ \text{zig-zag property (7)}\ \} \\
 & (x \downarrow y) \curlyvee (x \uparrow y) \text{ ordered}
\end{array}
$$

So the additional circuitry is just a single column of comparators: the *even-odd cleaner* of Section 2.4. It is convenient to introduce an infix operator for the circuitry: $x \updownarrow y = \textit{even-odd-clean}\,(x \curlyvee y)$. We can then substitute this new operator for the placeholder in our original sketch of the merger and by the preceding calculation we guarantee that the output is sorted.

$$
\begin{aligned}
\langle a \rangle \mathbin{/\!\!\!\backslash\!\!\backslash} \langle b \rangle &= \langle a \downarrow b, a \uparrow b \rangle && \text{(11a)} \\
(s \curlyvee t) \mathbin{/\!\!\!\backslash\!\!\backslash} (u \curlyvee v) &= (s \mathbin{/\!\!\!\backslash\!\!\backslash} v) \updownarrow (t \mathbin{/\!\!\!\backslash\!\!\backslash} u) && \text{(11b)} \\
x \updownarrow y &= (x \downarrow y) \curlyvee (x \uparrow y)
\end{aligned}
$$

We claim that this is Batcher's *bitonic merger.* This assertion is verified in Sections 5 and 6, but first let us explore the second design option mentioned at the start of this section. Batcher called his second arrangement the "odd-even merger", but it is also known as the merge exchange network [Knuth 1998].

4 Deriving Batcher's Merge Exchange Network

The bitonic merger "mixes" even and odd sub-sequences in the recursive calls. The only other option is to recurse on the even sequences and on the odd ones:

$$\begin{aligned}
\langle a\rangle \mathbin{/\!\!\backslash\!\!\backslash} \langle b\rangle &= \langle a \downarrow b, a \uparrow b\rangle \\
(s \curlyvee t) \mathbin{/\!\!\backslash\!\!\backslash} (u \curlyvee v) &= (s \mathbin{/\!\!\backslash\!\!\backslash} u) \mathbin{?} (t \mathbin{/\!\!\backslash\!\!\backslash} v)
\end{aligned}$$

We aim to derive a circuit to substitute for the placeholder in this specification in a similar way as we did in Section 3. Now that s is combined with u instead of v, the inductive assumption (10b) becomes:

$$s \mathbin{/\!\!\backslash\!\!\backslash} u \text{ ordered} \quad\wedge\quad t \mathbin{/\!\!\backslash\!\!\backslash} v \text{ ordered} \tag{12}$$

The derivation then proceeds as before. First we massage the pre-condition of merge (10a). A similar calculation to that in Section 3 produces:

$$s \mathbin{/\!\!\backslash\!\!\backslash} u \leqslant t \mathbin{/\!\!\backslash\!\!\backslash} v \;\wedge\; \bot \cdot \bot \cdot (t \mathbin{/\!\!\backslash\!\!\backslash} v) \leqslant (s \mathbin{/\!\!\backslash\!\!\backslash} u) \cdot \top \cdot \top$$

Let us again introduce some shortcuts for the results of the recursive calls. This time we choose to replace $s \mathbin{/\!\!\backslash\!\!\backslash} u$ by $\langle a\rangle \cdot x$ and $t \mathbin{/\!\!\backslash\!\!\backslash} v$ by $y \cdot \langle b\rangle$. The reason for picking this generalization instead of simply x and y as in Section 3 is that *head* $(s \mathbin{/\!\!\backslash\!\!\backslash} u)$ is the overall minimum and, dually, *last* $(t \mathbin{/\!\!\backslash\!\!\backslash} v)$ the overall maximum. This fact actually drops out naturally from the final calculation. Using these shortcuts the pre-condition can be further simplified.

$$\begin{aligned}
& \langle a\rangle \cdot x \leqslant y \cdot \langle b\rangle \;\wedge\; \bot \cdot \bot \cdot y \cdot \langle b\rangle \leqslant \langle a\rangle \cdot x \cdot \top \cdot \top \\
\iff & \quad \{ \text{ pointwise ordering, bottom and top (2) } \} \\
& \langle a\rangle \cdot x \leqslant y \cdot \langle b\rangle \;\wedge\; \bot \cdot y \leqslant x \cdot \top \\
\iff & \quad \{ \text{ pointwise ordering } \} \\
& a \leqslant \mathit{head}\, y \;\wedge\; \mathit{init}\, x \leqslant \mathit{tail}\, y \;\wedge\; \mathit{last}\, x \leqslant b \;\wedge\; \bot \cdot y \leqslant x \cdot \top \\
\iff & \quad \{ \; \mathit{init}\, x \leqslant \mathit{tail}\, y \iff \bot \cdot x \leqslant y \cdot \top \; \} \\
& a \leqslant \mathit{head}\, y \;\wedge\; \bot \cdot x \leqslant y \cdot \top \;\wedge\; \mathit{last}\, x \leqslant b \;\wedge\; \bot \cdot y \leqslant x \cdot \top
\end{aligned}$$

Next we introduce the induction assumption (12).

$$\begin{aligned}
& \langle a\rangle \cdot x \text{ ordered} \;\wedge\; a \leqslant \mathit{head}\, y \;\wedge\; \bot \cdot x \leqslant y \cdot \top \;\wedge\; \mathit{last}\, x \leqslant b \\
& \quad \wedge\; \bot \cdot y \leqslant x \cdot \top \;\wedge\; y \cdot \langle b\rangle \text{ ordered} \\
\iff & \quad \{ \text{ split property (twice) (6) } \} \\
& a \leqslant \mathit{head}\, x \;\wedge\; x \text{ ordered} \;\wedge\; a \leqslant \mathit{head}\, y \;\wedge\; \bot \cdot x \leqslant y \cdot \top \\
& \quad \wedge\; \mathit{last}\, x \leqslant b \;\wedge\; \bot \cdot y \leqslant x \cdot \top \;\wedge\; y \text{ ordered} \;\wedge\; \mathit{last}\, y \leqslant b
\end{aligned}$$

$$
\begin{array}{ll}
\iff & \{ \text{ see last calculation in Section 3 } \} \\
 & a \leqslant \mathit{head}\, x \;\wedge\; a \leqslant \mathit{head}\, y \;\wedge\; x \updownarrow y \text{ ordered} \\
 & \quad \wedge\; \mathit{last}\, x \leqslant b \;\wedge\; \mathit{last}\, y \leqslant b \\
\iff & \{ \text{ characterization of minimum (1a) and maximum (1b) } \} \\
 & a \leqslant \mathit{head}\, x \downarrow \mathit{head}\, y \;\wedge\; x \updownarrow y \text{ ordered} \;\wedge\; \mathit{last}\, x \uparrow \mathit{last}\, y \leqslant b \\
\iff & \{ \; \mathit{head} \text{ and } \mathit{tail} \text{ distribute over lifted operations } \} \\
 & a \leqslant \mathit{head}\,(x \downarrow y) \;\wedge\; x \updownarrow y \text{ ordered} \;\wedge\; \mathit{last}\,(x \uparrow y) \leqslant b \\
\iff & \{ \; \mathit{head}\,(x \updownarrow y) = \mathit{head}\,(x \downarrow y) \text{ and } \mathit{last}\,(x \updownarrow y) = \mathit{last}\,(x \uparrow y) \; \} \\
 & a \leqslant \mathit{head}\,(x \updownarrow y) \;\wedge\; x \updownarrow y \text{ ordered} \;\wedge\; \mathit{last}\,(x \updownarrow y) \leqslant b \\
\iff & \{ \text{ split property (6) } \} \\
 & \langle a \rangle \cdot (x \updownarrow y) \cdot \langle b \rangle \text{ ordered}
\end{array}
$$

Again, a single column of comparators does the job. However, one less comparator is used than in Section 3 as a is the overall minimum and b the overall maximum. The resulting circuit is called an *odd-even cleaner*, denoted by $\upharpoonleft\!\downharpoonright$. Substituting this operator for the placeholder in our original sketch, we obtain:

$$
\begin{aligned}
\langle a \rangle \mathbin{/\!\!\wedge\!\!\backslash} \langle b \rangle &= \langle a \downarrow b, a \uparrow b \rangle \\
(s \curlyvee t) \mathbin{/\!\!\wedge\!\!\backslash} (u \curlyvee v) &= (s \mathbin{/\!\!\wedge\!\!\backslash} u) \mathbin{\upharpoonleft\!\downharpoonright} (t \mathbin{/\!\!\wedge\!\!\backslash} v) \\
(\langle a \rangle \cdot x) \mathbin{\upharpoonleft\!\downharpoonright} (y \cdot \langle b \rangle) &= \langle a \rangle \cdot (x \updownarrow y) \cdot \langle b \rangle
\end{aligned}
$$

This is identical to the definition of Batcher's *exchange merger* from previous work [Hinze and Martin 2017a]. Quite amazingly, by systematically exploring the design space, we claim to have obtained both of Batcher's mergers. It is also crystal clear that the exchange merger requires fewer comparators than the bitonic merger as it builds on the odd-even cleaner $\upharpoonleft\!\downharpoonright$ rather than the even-odd cleaner $\updownarrow$. It now remains to confirm that the bitonic merger really is the same as that invented by Batcher. This is the subject of the next two sections.

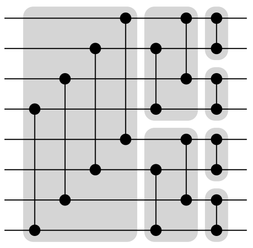
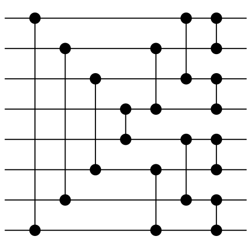
5 Deriving the Bitonic Sorter

Batcher did not introduce a bitonic merger in his seminal paper [Batcher 1968], but what he called a *bitonic sorter*. This is actually something of a misnomer, since such networks do not sort arbitrary inputs. Instead they sort only *bitonic sequences*: ones that first increase then decrease, or can be circularly shifted to such.

We will now show how the definition of the merger calculated in Section 3 is related to Batcher's bitonic sorter, taking two steps to do so. We start with a recap of Batcher's construction in Section 5.1, as presented, for example, in [Cormen et al. 2001]. Next in Section 5.2 we derive an algebraic definition of the bitonic sorter from our merger, further simplifying the design in the process. Finally in Section 6, we show that the two definitions do indeed coincide.

5.1 Cormen et al.'s diagrammatic presentation

Of Batcher's two designs, the bitonic merger seems to be the more popular one, perhaps because it enjoys an attractive diagrammatic decomposition, see diagram on the left below.

The circuit is composed of several stages, each corresponding to a *low-high-cleaner*, as defined in Section 2.4. For example, the leftmost 4 comparators in the left-hand diagram above constitute the cleaner for 8 inputs. When applied to a bitonic input sequence, the cleaner produces output with smaller numbers in the bottom half and larger ones in the top, and both halves bitonic. (The original name for the low-high-cleaner, *half-cleaner*, stems from the fact that if the input contains only zeros and ones, at least one half of the output will be *clean*: consisting solely of either zeros or ones.)

The shading in the diagram on the left above indicates how low-high-cleaners are combined recursively to create a bitonic sorter:

$$\begin{aligned} \mathit{bisort}\,\langle a\rangle &= \langle a\rangle \\ \mathit{bisort}\,(x \parallel y) &= \mathit{bisort}\,(x \downarrow y) \parallel \mathit{bisort}\,(x \uparrow y) \end{aligned}$$

The merger is then defined in terms of the bitonic sorter. The result is illustrated in the network on the right above, formed by modifying that on the left. It is based on the intuition that two ordered sequences x and y can be merged by applying a bitonic sorter to the first concatenated to the reverse of the second:

$$x \mathbin{/\!\!\!\wedge\!\!\!\backslash} y = \mathit{bisort}\,(x \parallel \mathit{rev}\ y) \tag{14}$$

This reversal of the second half of the input can be performed implicitly, hence the rearrangement of the leftmost 4 comparators in the diagram on the right.

5.2 Misra's algebraic presentation

Misra uses relationship (14) to apply a correctness proof of the bitonic sorter to the merger [Misra 1994]. Here we do the opposite. We derive the bitonic sorter from the bitonic merger, simultaneously establishing its correctness. This order reversal leads us to question the necessity for the very existence of the misleadingly named bitonic sorter or the associated notion of bitonicity.

Rearranging (14), the bitonic sorter is specified as a one-argument version of the bitonic merger, with the twist that the second argument is reversed:

$$\mathit{bisort}\,(x \parallel y) = x \mathbin{/\!\!\backslash\!\!\backslash} \mathit{rev}\; y \tag{15}$$

In the following derivation we distinguish between a base case, two wires, and an inductive case, more than two wires. For reasons that become clear from the calculation, the point of departure is in both cases an interleaving:

$$\begin{array}{ll}
 & \mathit{bisort}\,(\langle a\rangle \curlyvee \langle b\rangle) \\
= & \{\ \text{singletons}\ \} \\
 & \mathit{bisort}\,(\langle a\rangle \parallel \langle b\rangle) \\
= & \{\ \text{specification (15)}\ \} \\
 & \langle a\rangle \mathbin{/\!\!\backslash\!\!\backslash} \mathit{rev}\,\langle b\rangle \\
= & \{\ \text{definition of reverse}\ \} \\
 & \langle a\rangle \mathbin{/\!\!\backslash\!\!\backslash} \langle b\rangle \\
= & \{\ \text{definition of merge (11a)}\ \} \\
 & \langle a \downarrow b, a \uparrow b\rangle \\
= & \{\ \text{definition of even-odd cleaner}\ \} \\
 & \langle a\rangle \updownarrow \langle b\rangle
\end{array}
\qquad
\begin{array}{ll}
 & \mathit{bisort}\,((s \parallel t) \curlyvee (u \parallel v)) \\
= & \{\ \text{interchange law (3)}\ \} \\
 & \mathit{bisort}\,((s \curlyvee u) \parallel (t \curlyvee v)) \\
= & \{\ \text{specification (15)}\ \} \\
 & (s \curlyvee u) \mathbin{/\!\!\backslash\!\!\backslash} \mathit{rev}\,(t \curlyvee v) \\
= & \{\ \text{reverse and interleaving}\ \} \\
 & (s \curlyvee u) \mathbin{/\!\!\backslash\!\!\backslash} (\mathit{rev}\; v \curlyvee \mathit{rev}\; t) \\
= & \{\ \text{definition of merge (11b)}\ \} \\
 & (s \mathbin{/\!\!\backslash\!\!\backslash} \mathit{rev}\; t) \updownarrow (u \mathbin{/\!\!\backslash\!\!\backslash} \mathit{rev}\; v) \\
= & \{\ \text{specification (twice) (15)}\ \} \\
 & \mathit{bisort}\,(s \parallel t) \updownarrow \mathit{bisort}\,(u \parallel v)
\end{array}$$

We could use the resulting equalities as defining equations. A moment's reflection, however, reveals that we can streamline the definition:

$$\begin{array}{ll}
\mathit{bisort}\,\langle a\rangle & = \langle a\rangle \\
\mathit{bisort}\,(x \curlyvee y) & = \mathit{bisort}\, x \updownarrow \mathit{bisort}\, y
\end{array}$$

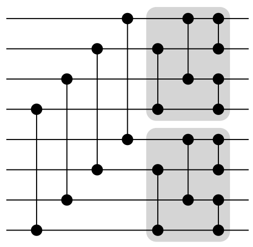
We have introduced a base case for singletons and generalized the inductive case.

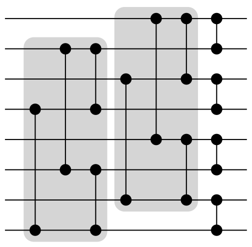
6 Relating Diagrams and Algebra

We are finally in a position to reconcile art with mathematics, diagrams with algebra. Reconciliation is asked for as the recursive decomposition of the bitonic sorter is quite different in the two approaches, the diagrammatic presentation

[Cormen et al. 2001] and the algebraic presentation [Misra 1994].

diagrammatic presentation **algebraic presentation**





$$bisort\,\langle a\rangle = \langle a\rangle$$
$$bisort\,(x \parallel y) = bisort\,(x \downarrow y) \parallel bisort\,(x \uparrow y)$$

$$bisort\,\langle a\rangle = \langle a\rangle$$
$$bisort\,(x \curlyvee y) = bisort\,x \updownarrow bisort\,y$$

In the definition on the left, the divide step requires additional circuitry (the low-high-cleaner), while the conquer step is free. On the right-hand side, the divide step is free, while the conquer step requires additional circuitry (the even-odd-cleaner). These differences are, however, illusory: the structure of the circuits is identical. The diagram on the left is obtained from the diagram on the right by shifting the comparators that span five wires as far as possible to the left. Moreover, the correspondence does not depend on the specifics of the underlying comparator circuit—it is purely structural; it holds for any gate with two inputs and two outputs. What follows is an algebraic proof of this claim.

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

$$(f \parallel g)\,x = f\,x \parallel g\,x \qquad (f \curlyvee g)\,x = f\,x \curlyvee g\,x$$

We also require projection functions as a substitute for the use of concatenation and interleaving in patterns:

$$low\ \ (x \parallel y) = x \qquad even\,(x \curlyvee y) = x$$
$$high\,(x \parallel y) = y \qquad odd\ \ (x \curlyvee y) = y$$

Constructors and projections are related by the following universal properties:

$$f = low \circ g\ \wedge\ g = high \circ h \iff f \parallel g = h$$
$$f = even \circ g\ \wedge\ g = odd \circ h \iff f \curlyvee g = h$$

where $\circ$ is function composition. You may recognize the similarity to categorical products. Indeed, we can use the ingredients above to define the product arrow:

$$f \times g = f \circ \mathit{low} \parallel g \circ \mathit{high} \qquad\qquad f \star g = f \circ \mathit{even} \curlyvee g \circ \mathit{odd}$$

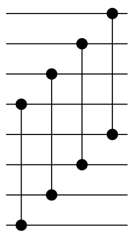
The universal properties imply the following functor laws:

$$id \times id = id \tag{16a}$$
$$(f \circ g) \times (h \circ k) = (f \times h) \circ (g \times k) \tag{16b}$$
$$id \star id = id \tag{16c}$$
$$(f \circ g) \star (h \circ k) = (f \star h) \circ (g \star k) \tag{16d}$$

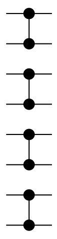
Finally, the interchange laws below make precise how the two products interact:

$$(f \curlyvee g) \parallel (h \curlyvee k) = (f \parallel h) \curlyvee (g \parallel k) \tag{17a}$$
$$(f \star g) \times (h \star k) = (f \times h) \star (g \times k) \tag{17b}$$

We illustrate the use of these point-free combinators by redefining the low-high and even-odd cleaners which were defined pointwise in Section 2.4. They are obtained by repeatedly "squaring" the underlying comparator circuit, for example:



$$\begin{array}{c} (\mathit{cmp} \star \mathit{cmp}) \\ \star \\ (\mathit{cmp} \star \mathit{cmp}) \end{array} \qquad \begin{array}{c} (\mathit{cmp} \times \mathit{cmp}) \\ \times \\ (\mathit{cmp} \times \mathit{cmp}) \end{array}$$



In general, the low-high cleaner is $\mathit{cmp}^{k\star}$, while the even-odd is $\mathit{cmp}^{k\times}$, where

$$f^{0\times} = f \qquad\qquad f^{0\star} = f$$
$$f^{(k+1)\times} = f^{k\times} \times f^{k\times} \qquad\qquad f^{(k+1)\star} = f^{k\star} \star f^{k\star}$$

We can use these combinators to express the algebraic definitions of the bitonic sorters in a point-free style:

$$\mathit{bisort}_0 = id \qquad\qquad \mathit{bisort}_0 = id$$
$$\mathit{bisort}_{k+1} = (\mathit{bisort}_k \times \mathit{bisort}_k) \circ \mathit{cmp}^{k\star} \qquad \mathit{bisort}_{k+1} = \mathit{cmp}^{k\times} \circ (\mathit{bisort}_k \star \mathit{bisort}_k)$$

This presentation very clearly exhibits the symmetry between the two definitions. In a sense they are dual: we obtain one from the other simply by exchanging $\times$ and $\star$. Turning to the equivalence proof, we first abstract away from the specifics of the application:

$$f_0 = id \qquad\qquad g_0 = id$$
$$f_{k+1} = (f_k \times f_k) \circ c^{k\star} \qquad g_{k+1} = c^{k\times} \circ (g_k \star g_k)$$

To establish $f = g$, we show that f also satisfies the recursion equation of g:

$$f_{k+1} = c^{k\times} \circ (f_k \star f_k) \tag{18}$$

The proof proceeds by induction over k:

$$\begin{aligned}
& f_1 \\
= {} & \{ \text{ definition of } f \ \} \\
& (f_0 \times f_0) \circ c^{0\star} \\
= {} & \{ \text{ definition of } f \ \} \\
& (id \times id) \circ c^{0\star} \\
= {} & \{ \text{ functor law (16a) } \} \\
& c^{0\star} \\
= {} & \{ \text{ definition of } c^{\star} \ \} \\
& c \\
= {} & \{ \text{ definition of } c^{\times} \ \} \\
& c^{0\times} \\
= {} & \{ \text{ functor law (16c) } \} \\
& c^{0\times} \circ (id \star id) \\
= {} & \{ \text{ definition of } f \ \} \\
& c^{0\times} \circ (f_0 \star f_0)
\end{aligned}$$

$$\begin{aligned}
& f_{k+2} \\
= {} & \{ \text{ definition of } f \ \} \\
& (f_{k+1} \times f_{k+1}) \circ c^{(k+1)\star} \\
= {} & \{ \text{ induction assumption (18) } \} \\
& ((c^{k\times} \circ (f_k \star f_k)) \times (c^{k\times} \circ (f_k \star f_k))) \circ c^{(k+1)\star} \\
= {} & \{ \text{ functor law (16d) } \} \\
& (c^{k\times} \times c^{k\times}) \circ ((f_k \star f_k) \times (f_k \star f_k)) \circ c^{(k+1)\star} \\
= {} & \{ \text{ definition of } c^{\times} \ \} \\
& c^{(k+1)\times} \circ ((f_k \star f_k) \times (f_k \star f_k)) \circ c^{(k+1)\star} \\
= {} & \{ \text{ interchange law (17b) } \} \\
& c^{(k+1)\times} \circ ((f_k \times f_k) \star (f_k \times f_k)) \circ c^{(k+1)\star} \\
= {} & \{ \text{ definition of } c^{\star} \ \} \\
& c^{(k+1)\times} \circ ((f_k \times f_k) \star (f_k \times f_k)) \circ (c^{k\star} \star c^{k\star}) \\
= {} & \{ \text{ functor law (16d) } \} \\
& c^{(k+1)\times} \circ ((f_k \times f_k) \circ c^{k\star}) \star ((f_k \times f_k) \circ c^{k\star}) \\
= {} & \{ \text{ definition of } f \ \} \\
& c^{(k+1)\times} \circ (f_{k+1} \star f_{k+1})
\end{aligned}$$

The base case on the left is entirely straightforward. The central manoeuvre in the inductive step on the right is the application of the interchange law (17b). (As an aside, observe that the terms surrounding the rewrite also exhibit a nice symmetric recursion structure, where some work is done before the recursive calls and some work is done afterwards.)

7 Relation to Sorting Networks

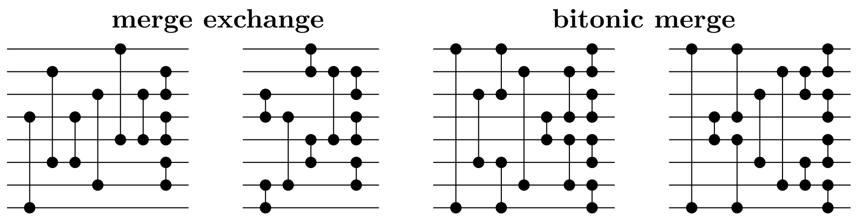
We have derived two mergers from first principles, using an inductive approach based on uninterleaving the input sequences. We then showed that the resulting schemes correspond to those proposed by Batcher. Let us now apply the same approach to sorters. Once again there are two disciplined ways to sub-divide the input: halving and uninterleaving. This time we will explore both strategies because examples of almost all of the resulting arrangements appear in the literature, in various guises.

7.1 The Sorters

The first step is to rewrite the binary merge operator as an equivalent function of one argument. The resulting two versions of the merger are denoted as follows:

$$merge_{\|}\ (x \parallel y) = x \bigwedge\!\!\!\bigwedge y \qquad\qquad merge_{\curlyvee}\ (x \curlyvee y) = x \bigwedge\!\!\!\bigwedge y$$

The associated Knuth diagrams for circuits with 8 inputs are presented below, where (a) and (b) depict the exchange merger, while (c) and (d) represent the bitonic one. In each case the diagram on the left uses halving and that on the right interleaving; the difference amounts to a bit reversal permutation of the input.



(a) $merge_{\|}$ (b) $merge_{\curlyvee}$ (c) $merge_{\|}$ (d) $merge_{\curlyvee}$

Each of these four merging networks gives rise to a sorter:

$$\begin{aligned} sort_{\oplus}\ \langle a\rangle &= \langle a\rangle \\ sort_{\oplus}\ (x \oplus y) &= merge_{\oplus}\ (sort_{\oplus}\ x \oplus sort_{\oplus}\ y) \end{aligned}$$

where $\oplus$ is instantiated either to $\|$ or $\curlyvee$. The sorters that use halving represent Batcher's original methods. The interleaved merge exchange sorter has appeared before [Codish and Zazon-Ivry 2010], as has the bitonic sorter [Misra 1994].

7.2 The Twist

The mergers can be mechanically divided into two sub-components by considering the base case separately from the inductive one. This arrangement is particularly interesting for the exchange merger, which has an intriguing twist. In that case, the factorized definitions are as follows:

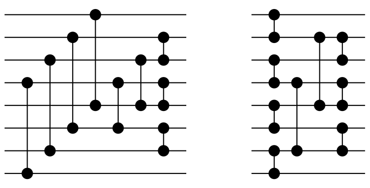
$$merge_{\|} = pair_{\|} \circ \textit{low-high–clean} \qquad\qquad merge_{\curlyvee} = pair_{\curlyvee} \circ \textit{even-odd-clean}$$

where $pair_{\oplus}\ (x \oplus y) = x \curlywedge y$ is merge with a trivial base case:

$$\begin{aligned} \langle a\rangle \curlywedge \langle b\rangle &= \langle a, b\rangle \\ (s \curlyvee t) \curlywedge (u \curlyvee v) &= (s \curlywedge u) \updownarrow (t \curlywedge v) \end{aligned}$$

The function $pair_{\curlyvee}$ is a "pair sorter", which sorts sequences of sorted pairs. Its halving analogue, $pair_{\|}$, behaves similarly. The Knuth diagrams for the mergers now have slightly different layouts from before:

twisted merge exchange



(a') $merge_{||}$ (b') $merge_{\curlyvee}$

The non-base cases of the corresponding sorters become:

$$\begin{aligned} sort_{||} &= pair_{||} \circ \mathit{low\text{-}high\text{–}clean} \circ (sort_{||} \times sort_{||}) \\ sort_{\curlyvee} &= pair_{\curlyvee} \circ \mathit{even\text{-}odd\text{-}clean} \circ (sort_{\curlyvee} \star sort_{\curlyvee}) \end{aligned}$$

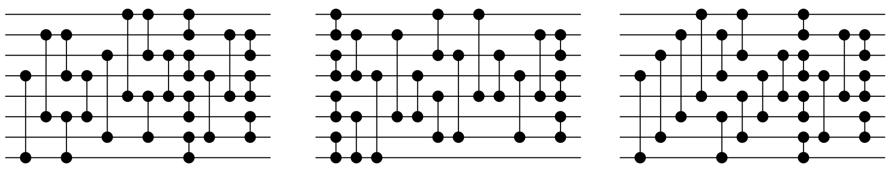
Now, here is the twist. Since each component preserves orderedness, the positions of the first two phases can be swapped:

$$\begin{aligned} sort_{||} &= pair_{||} \circ (sort_{||} \times sort_{||}) \circ \mathit{low\text{-}high\text{–}clean} \\ sort_{\curlyvee} &= pair_{\curlyvee} \circ (sort_{\curlyvee} \star sort_{\curlyvee}) \circ \mathit{even\text{-}odd\text{-}clean} \end{aligned}$$

The resulting interleaved sorter is a recursive expression of Parberry's pairwise sorting network [Parberry 1992, Hinze and Martin 2017b]. We have not seen the version using halving in the literature.

The Knuth diagrams below show the difference between various layouts of the interleaved exchange sorter: (e) is the original from Section 7.1 and (f) has the phases swapped, as in Parberry's adaptation. The third arrangement, (g), corresponds to two accounts of "Batcher's Baffler" with correctness proofs in a traditional imperative style [Gries 1986, Dijkstra 1987]. These latter two are the earliest expositions of the interleaved exchange sorter of which we are aware.

merge exchange sorter **Parberry's twist** **Batcher's Baffler**



(e) (f) (g)

The altered layout of (g) compared to (e) reflects the iterative nature of these presentations. Algebraically, it can be computed by unfolding the recursion then rearranging the components using the interchange law (16d), so for example:

$$\begin{aligned} & sort_{\curlyvee} \star sort_{\curlyvee} \\ &\quad = (pair_{\curlyvee} \star pair_{\curlyvee}) \circ (\mathit{even\text{-}odd\text{-}clean} \star \mathit{even\text{-}odd\text{-}clean}) \\ &\qquad \circ ((sort_{\curlyvee} \star sort_{\curlyvee}) \star (sort_{\curlyvee} \star sort_{\curlyvee})) \end{aligned}$$

It is natural to ask whether the phases can also be swapped for the bitonic mergers. They have a similar decomposition involving a pair-sorter:

$$\begin{aligned} \mathit{merge}_{\|} &= \mathit{pair}'_{\|} \circ \mathit{low\text{-}high\text{–}clean} \circ (\mathit{id} \times \mathit{rev}) \\ \mathit{merge}_{\curlyvee} &= \mathit{pair}'_{\curlyvee} \circ \mathit{even\text{-}odd\text{-}clean} \circ (\mathit{id} \star \mathit{rev}) \end{aligned}$$

where $\mathit{pair}'_{\oplus}$ has the same definition as $\mathit{pair}_{\oplus}$ except that the odd-even cleaner $\rightharpoonup\!\!\!\downharpoonleft$ is replaced by the even-odd one $\updownarrow$. Unfortunately the components are not as well-behaved as those for the exchange merger. Neither $\mathit{id} \times \mathit{rev}$ nor $\mathit{id} \star \mathit{rev}$ preserve orderedness, and so a similar interchange of sub-parts is not possible.

There are, of course, many other possible permutations of the input if we remove the restriction to halving and interleaving. For example, there are 315 different rearrangements of the 8-key merge exchange sorter alone [Al-Haj Baddar and Batcher 2011].

8 Related Work

Batcher's algorithms were the first systematic methods for designing sorting networks of arbitrary input size. For decades, they have attracted enormous interest from academia and industry alike; at the time of writing there are over 2500 citations of the original paper. Much of the related work concerns performance and bounds. Batcher's merge exchange sorter has been proven to use a minimal number of comparators for input sizes $n \leqslant 8$ [Knuth 1998]. Van Voorhis showed how greater economy of comparators is possible for larger input sizes by dividing the input into more than two groups [Van Voorhis 1971] and the results are still the best known for some values. The challenge of proving that a given number of comparators is minimal for sorting networks of input size $n > 8$ is still an open problem.

There is also a substantial body of literature on the design, modelling and verification of Batcher's networks. We therefore begin by categorizing a small selection of examples according to the approach taken, and then refer the reader to other work for more comprehensive reviews.

Imperative Both Gries and Dijkstra produced formal correctness proofs of the merge exchange sorter, lovingly called "Batcher's Baffler", in an imperative style [Gries 1986, Dijkstra 1987]. Neither used the zero-one principle but the proofs are quite lengthy in comparison to their functional counterparts.

Relational The bitonic merger was described and analysed in the relational language Ruby [Sheeran 1991] and shown to be related to the balanced merger. More recently, relations were used to identify the relationship between Parberry's pairwise network and Batcher's merge exchange sorter [Codish and Zazon-Ivry 2010]. The recursive, relational presentation is simpler and more straightforward to follow than Parberry's description.

Functional A functional description of bitonic sort appeared as an illustration of an algebraic model of divide-and-conquer algorithms for parallel computers [Mou and Hudak 1988] but this did not include any correctness proofs. An elegant and succinct correctness proof of bitonic sort was given subsequently using functions on the powerlist data structure [Misra 1994]. The idea of using parametricity in Haskell as an alternative to Knuth's zero-one principle was introduced later [Day et al. 1999]. Bitonic sort was included as a motivating example to demonstrate how a verification of correctness could be performed on Boolean input and then generalised to more complex types.

Machine-checked Formal Proofs One of the first formal proofs of correctness of bitonic sort using an automated reasoning system [Couturier 1998] was performed in the prototype verification system PVS [Owre et al. 1992]; the proof did not use the zero-one principle. Lava, a tool for hardware design and verification, was used later [Claessen et al. 2003] to verify both the bitonic and merge exchange algorithms using the zero-one principle. This account also mentioned the close relationship between the two mergers, in the sense that their definitions differ only in the choice of cleaner. The difference here is our derivation from first principles and precise relation to other expositions. Constructive type theory was used to verify bitonic sort in Agda, via the zero-one principle [Bove and Coquand 2006]. Agda was also used to verify bitonic sort using parametricity [Dybjer et al. 2004], in a method that combined testing, model checking and theorem proving. Braibant and Chlipala followed a similar approach to Bove and Coquand to conduct a formal proof in Coq, but with the added dimension of connecting to an actual hardware implementation in the Fesi language [Braibant and Chlipala 2013].

There is a further distinction within these various paradigms between recursive and iterative approaches. Sheeran comments that recursive descriptions can be more comprehensible, as they offer more insight into how the algorithm was designed [Sheeran 1989]. This is borne out by the contrast between the elegance of recursive butterfly circuits [Jones and Sheeran 1991] and the complexity of the derivations of an iterative merge exchange sort by Gries and Dijkstra. Parberry's method is an exception as the original design was iterative, but we argue that the recursive expression of the algorithm in Sections 7 is more transparent. An iterative approach to the bitonic sort is given in Obsidian [Claessen et al. 2012], which is quite easy to follow.

In the broader context, the interest in hardware design among the functional programming community that was sparked in the 1980's is still strong [Sheeran 2015]. This is partly because hardware design is essentially a form of parallel programming and recent growth in the use of parallel machines has driven a demand for related tools. As a consequence, the development of data-parallel functional programming languages has flourished [Blelloch 1993, Chakravarty et al. 2007]. Functional languages are well-suited for designing, analysing and validating parallel algorithms for many reasons. These include their high level of abstraction, use of higher-order functions to structure descriptions, associated formal transformation and verification methods, and ease of algebraic manipulation for equational reasoning. We refer the reader to [Gammie 2013] for excellent and comprehensive review of the long tradition of describing circuits using functional programming techniques. Gammie also includes a summary of other formal methods for hardware design including algebraic techniques, relational models and diagrammatic methods involving "boxes and wires". For more details of the early history see also [Knuth 1998] and [Sheeran 2005].

9 Conclusion

To conclude, the "bitonicity" property invented by Batcher is really a red herring. Yet he showed extraordinary insight to conceive both of his schemes using diagrams alone, without the benefit of calculation. The algebraic method permits a clean derivation of the mergers from minimal assumptions. It shows they are the two canonical choices, and the proofs rely solely on the monotonicity property of comparison networks. This rigorous approach also unmasks the resemblance between the two mergers. In both cases, all of the work is done by the conquer step, while the divide step is free. In contrast, Batcher's presentation of the circuits show a surprising inconsistency: the burden of the work is assumed by the divide step for one, and conquer for the other.

Batcher names two advantages of the bitonic sorter over the exchange sorter. First, they are flexible in the sense that one network can accommodate input lists of various lengths. Second, they are modular, so a network can be split up into several identical modules. We question whether these advantages are really valid? Both sorters appear to be equally flexible and have identical modularity, as evidenced by their recursive decomposition.

A final argument in favour of the symbolic presentation is the fresh perspective afforded by the point-free view. This high-level stance is useful for exposing duality between different presentations of the bitonic sorter, as well as making precise the connection between the exchange and pairwise sorters [Hinze and Martin 2017b].

Acknowledgements

We are grateful to the anonymous referees for constructive suggestions, including the addition of some historical perspective to the introduction and the removal of "rabbits" from the derivations. Thanks also to Jeremy Gibbons for pointing us

towards Dijkstra's derivation of "Batcher's Baffler" and David Gries for sharing an earlier manuscript.

References

[Al-Haj Baddar and Batcher 2011] Al-Haj Baddar, S. W., Batcher, K. E.: Designing Sorting Networks; Springer, 2011.

[Batcher 1968] Batcher, K. E.: "Sorting networks and their applications"; Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference; 307–314; ACM, 1968.

[Blelloch 1993] Blelloch, G. E.: "Nesl: A nested data-parallel language"; Technical report; CMU-CS-93-129, Carnegie Mellon University (1993).

[Bove and Coquand 2006] Bove, A., Coquand, T.: "Formalising bitonic sort in type theory"; J.-C. Fillitre, C. Paulin-Mohring, B. Werner, eds., Types for Proofs and Programs; volume 3839 of Lecture Notes in Computer Science; 82–97; Springer, 2006.

[Braibant and Chlipala 2013] Braibant, T., Chlipala, A.: "Formal verification of hardware synthesis"; Computer Aided Verification; volume 8044 of Lecture Notes in Computer Science; 213–228; Springer, 2013.

[Chakravarty et al. 2007] Chakravarty, M. M., Leshchinskiy, R., Peyton Jones, S., Keller, G., Marlow, S.: "Data parallel Haskell: a status report"; Proceedings of the 2007 workshop on Declarative Aspects of Multicore Programming; 10–18; ACM, 2007.

[Claessen et al. 2003] Claessen, K., Sheeran, M., Singh, S.: "Functional hardware description in Lava"; The Fun of Programming. Cornerstones of Computing; 151–176; Palgrave, 2003.

[Claessen et al. 2012] Claessen, K., Sheeran, M., Svensson, B. J.: "Expressive array constructs in an embedded GPU kernel programming language"; Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming; 21–30; ACM, 2012.

[Codish and Zazon-Ivry 2010] Codish, M., Zazon-Ivry, M.: "Pairwise cardinality networks"; Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning; volume 6355 of Lecture Notes in Computer Science; 154–172; Springer, 2010.

[Cormen et al. 2001] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.: Introduction to Algorithms; The MIT Press, Cambridge, Massachusetts, 2001; third edition.

[Couturier 1998] Couturier, R.: "Formal engineering of the bitonic sort using PVS"; 2nd Irish Workshop in Formal Methods; British Computer Society Electronic Workshops in Computing, Cork, Ireland, 1998.

[Day et al. 1999] Day, N. A., Launchbury, J., Lewis, J.: "Logical abstractions in Haskell"; Proceedings of the 1999 Haskell Workshop; Utrecht University Department of Computer Science, Technical Report UU-CS-1999-28, 1999.

[Dijkstra 1987] Dijkstra, E. W.: "A heuristic explanation of Batcher's Baffler"; Science of Computer Programming; volume 9; 213–220; Elsevier, 1987.

[Dybjer et al. 2004] Dybjer, P., Haiyan, Q., Takeyama, M.: "Verifying Haskell programs by combining testing, model checking and interactive theorem proving"; Information and Software Technology; 46 (2004), 15, 1011–1025.

[Gammie 2013] Gammie, P.: "Synchronous digital circuits as functional programs"; ACM Computing Surveys; 46 (2013), 2, 21.

[Gries 1986] Gries, D.: Unpublished manuscript; 1986.

[Hinze and Martin 2017a] Hinze, R., Martin, C.: "Functional Pearl: Batcher's odd-even merging network revealed"; Journal of Functional Programming; (2017a); . To appear.

[Hinze and Martin 2017b] Hinze, R., Martin, C.: "Functional Pearl: Parberrys pairwise sorting network revealed"; (2017b); in submission.

[Jones and Sheeran 1991] Jones, G., Sheeran, M.: "The study of butterflies"; Proceedings of the IVth Higher Order Workshop, Banff 1990; 54–65; Springer, 1991.

[Knuth 1998] Knuth, D. E.: The Art of Computer Programming, Volume 3: Sorting and Searching; Addison-Wesley, 1998; 2nd edition.

[Misra 1994] Misra, J.: "Powerlist: A structure for parallel recursion"; ACM Transactions on Programming Languages and Systems; 16 (1994), 6, 1737–1767.

[Mou and Hudak 1988] Mou, Z. G., Hudak, P.: "An algebraic model for divide-and-conquer and its parallelism"; The Journal of Supercomputing; 2 (1988), 3, 257–278.

[Owre et al. 1992] Owre, S., Rushby, J. M., Shankar, N.: "PVS: A prototype verification system"; D. Kapur, ed., Proceedings of the Eleventh International Conference on Automated Deduction (CADE); volume 607 of Lecture Notes in Artificial Intelligence; 748–752; Springer-Verlag, 1992.

[OConnor and Nelson 1962] OConnor, D. G., Nelson, R. J.: "Sorting system with N-line sorting switch"; (1962); uS Patent Number 3,029,413.

[Parberry 1992] Parberry, I.: "The pairwise sorting network"; Parallel Processing Letters; 2 (1992), 205–211.

[Sheeran 1989] Sheeran, M.: "Describing butterfly networks in Ruby"; Proceedings of the Glasgow Workshop on Functional Programming; 182–205; Springer Workshops in Computing, 1989.

[Sheeran 1991] Sheeran, M.: "Sorts of butterflies"; Proceedings of the IVth Higher Order Workshop, Banff 1990; 66–76; Springer, 1991.

[Sheeran 2005] Sheeran, M.: "Hardware design and functional programming: a perfect match"; Journal of Universal Computer Science; 11 (2005), 7, 1135–1158.

[Sheeran 2015] Sheeran, M.: "Functional programming and hardware design: Still interesting after all these years"; Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming; ICFP 2015; 165–165; ACM, New York, NY, USA, 2015.

[Turner 1982] Turner, D. A.: "Recursion equations as a programming language"; Darlington, Henderson, Turner, eds., Functional Programming and its Applications. An advanced course; Cambridge University Press, 1982.

[Van Voorhis 1971] Van Voorhis, D. C.: "A generalization of the divide-sort-merge strategy for sorting networks"; Technical report; CS-TR-71-237, Stanford University, CA, USA (1971).