NOTE

ON OPTIMAL PARALLELIZATION OF SORTING NETWORKS*

Ethan GANNETT, Suraj C. KOTHARI and Hsu-Chun YEN
Department of Computer Science, Iowa State University, Ames, Iowa 50011, USA

Communicated by M. Nivat
Received May 1988

Abstract. This paper provides a unifying mathematical proof which replaces a mechanical certifi-
cation of the optimal parallelization of sorting networks on a case by case basis. Parallelization
of sequential program traces by means of a semantic-preserving transformation is also discussed
in the literature in the context of a method for synthesis of systolic architecture. The issue of optimal
parallelization is important in systolic design. The mathematical proof provides a better insight
into the fundamental aspects of the transformation.

1. Introduction

In his third volume of the Art of Computer Programming, Knuth has presented sorting networks for "constrained" type of sorting [5]. A node in a sorting network is a comparator module which takes elements as inputs, compares them and, if necessary, interchanges them into ascending order. Knuth has also introduced a suitable representation for a sorting network. The representation for insertion sort with five inputs is shown in Fig. 1. The elements enter from the left. The comparator

![Diagram of a sorting network for insertion sort with five inputs]

Fig. 1. Sorting network without overlap for insertion sort, delay = 10.

* An earlier version of this paper was presented at the seventh conference of Foundations of Software Technology and Theoretical Computer Science, Pune, India, December 1987.
modules are represented by vertical connections between two horizontal lines which correspond to elements of the sequence. The numbers come out on the right, in ascending order from top to bottom. Knuth presents sorting networks for several sorting algorithms. He also introduces the Zero-one principle to prove the validity of a sorting network.

An important advantage of sorting networks is the possibility of overlapping operations of comparator modules to minimize the delay through the network. For example the sorting network shown in Fig. 1, performs all comparisons in sequential order and the delay is 10. However the delay can be reduced to 7 by overlapping the operations of two comparator modules as shown in Fig. 2. Note that each comparison is done at the earliest possible time in this new “concurrent” network.

![Sorting network with overlap for insertion sort, delay = 7.](image)

In the context of a sorting network, the problem of optimal concurrency is to find the maximal overlap of comparator modules without changing the underlying semantics of the network. A methodology for parallelization of specific sorting networks and the optimality problem are discussed in [2] and [8]. The cited work was done in the context of a method for synthesis of systolic architecture [3, 4]. Parallelization of a sequential execution trace is a crucial requirement of the specific methodology. The optimality of the resulting systolic architecture depends on the optimality of the derived parallel trace. Thus, in general, automated derivation and a guarantee of optimality are important issues. The automated derivation, as will be seen later, is achieved through the Ravel Transform. The proofs for optimal concurrency for the bitonic sort [2] and the insertion sort [8] are mechanical and use the Bover-Moore theorem prover [1]. The proofs depend on a programming methodology [6, 7] which can deal formally with concurrency. The method proceeds by attempting to successively parallelize the program execution steps based on certain properties of the program that allow changes in the sequential executions without changing the semantics. For a sorting network, the commutativity and the idempotence of comparator module operations are the properties used to change the order of sequential execution in order to parallelize.
This paper includes a "mathematical" proof of optimal concurrency in place of a mechanical proof. We use the formalism in [2] and [8] and provide the necessary extensions to make the mathematical proof possible. A mathematical proof provides more insight into the problem versus a mechanical proof where certain ideas are hidden in a black box, namely the theorem prover. The insight has enabled us to prove new results. Instead of separate proofs as in [2] and [8] for two different sorting networks, we provide a unifying proof technique which works in general. In fact the mathematical proof of optimality holds for any network of comparator modules.

Section 2 describes the necessary formalism and the notation. The main result and proofs are described in Section 3.

2. Traces and their transformations

Following Knuth [5], cs(x, y) denotes the comparator module that accesses the array a[0...n] of elements and compares elements a[x] and a[y] (0 <= x, y <= n) and then interchanges them into order if necessary. A simple comparator module cs(x) (1 <= x <= n) compares adjacent array elements a[x] and a[x-1]. For ease of understanding, we will use simple comparator modules in our examples; the extension to regular comparator modules is straightforward. The sequential trace of a sorting network will consist of a sequential list of comparator modules.

2.1. Trace representation and execution time

Following the work of Lengauer and Huang [8], we use multilevel lists which allow us to represent parallel executions. A comparator module cs(i) will be represented as i in the lists. In a multilevel list, alternate list levels represent sequential execution and parallel execution in turn. The outermost list level (level 1) and all odd list levels will always denote sequential execution. The first nested list level (level 2) and all even list levels of a multilevel list will represent parallel execution. Hence, our representation of a sequential insertion sort trace on a six element array is:

(1 2 1 3 2 1 4 3 2 1 5 4 3 2 1)

If we restrict the execution time of a sorting network to be solely dependent on the execution time of a comparator module and we assert a unit execution time for each comparator module, the sequential insertion sort has execution time n(n + 1)/2.

A parallel trace of the insertion sort example would be:

(1 2 (3 1) (4 2) (5 3 1) (4 2) (3 1) 2 1)

In Section 3, such a trace will be written as:

(1 2 (3 1) (4 2) (5 3 1) (4 2) (3 1) 2 1)
or

\[
\begin{pmatrix}
1 & 2 & 3 & 4 & 5 & 4 & 3 \\
1 & 2 & 1 & \end{pmatrix}
\]

where ( ) denotes parallel execution. The transformation that yielded this trace can be performed on any size array and results in an execution time of \(2n - 1\) (assume instantaneous forks and joins). Thus, the transformation has improved the execution time of the insertion sort from quadratic to linear in the length of the array.

With this representation, we can determine the execution time of any multilevel trace by use of the following recursive Lisp-like function as presented in [8]:

\[
\text{(EXEC-TIME } \text{FLAG } \text{L}) =
\begin{cases}
\text{(IF } \text{NILP } \text{L}) \\
\text{(IF } \text{L = NIL}) \\
\text{(IF } \text{FLAG = 'PAR}) \\
\text{(PLUS} \text{(EXEC-TIME 'PAR (CAR L))} \\
\text{(EXEC-TIME 'SEQ (CDR L))}) \\
\text{(EXEC-TIME 'PAR (CAR L))} \\
\text{(EXEC-TIME 'SEQ (CDR L))})
\end{cases}
\]

where \(L\) is the trace, \text{FLAG} is the mode of execution ('SEQ or 'PAR) and \text{NILP} is the negation of \text{LISTP}. Note that the first call to \text{EXEC-TIME} has \text{FLAG} = 'SEQ.

2.2. Semantic-preserving operations and parallelization

The intent of a semantic-preserving operation on a sequential trace is to transform it to a semantically equivalent trace. The semantic properties of comparator modules are:

- **Idempotence**: A comparator module that can be executed once or any number of times consecutively with the same effect is idempotent.
- **Commutativity**: Two comparator modules \(C_1\) and \(C_2\) can be executed in any order with the same effect are said to be commutative.

We can use these properties to perform semantic-preserving operations on a trace. That is, for commutative comparator modules \(C_1\) and \(C_2\):

- \((C, C_1) \rightarrow (C)\) {deletion of a comparator module by idempotence};
- \((C_1, C) \rightarrow (C, C_1)\) {duplication of a comparator module by idempotence};
The data may vote that all comparator modules are idempotent. Idempotence allows comparator modules to be introduced into or deleted from a trace.

A semantic-preserving transformation $\mathcal{F}$ on a sequential trace $T$ is defined as a composition of finitely many semantic-preserving operations which possibly yields some new trace $T'$. We write $T \rightarrow^\mathcal{F} T'$.

Semantic-preserving operations provide a means of transforming a sequential trace into some other sequential trace. The objective of a parallelizing transformation is to transform a sequential trace to a possibly parallel trace. To do so, we must examine the dependence between comparator modules in a trace since independent modules could be executed in parallel. The notion of dependence between two comparator modules can be defined as follows:

Let $C_1$ and $C_2$ be comparator modules of the form

$$C_1 = cs(i, j), \quad C_2 = cs(k, l).$$

We define $C_1 \sim C_2$ (read as $C_1$ is dependent on $C_2$) if and only if

$$C_1 \sim C_2 \Leftrightarrow \{i, j\} \cap \{k, l\} \neq \emptyset \text{ and } C_1 \neq C_2.$$

Note that if $C_1 = C_2$, we consider these comparator modules independent.

Two comparator modules are said to be independent if they are not dependent. The reader may note that two consecutive comparator modules in a trace are commutative if and only if they are independent.

Similarly, we may define a linearly dependent sequence. Given a sequential trace $T = (C_1, C_2, \ldots, C_k)$, $T$ is called linearly dependent if and only if $C_i \sim C_{i+1}$ ($1 \leq i \leq k-1$).

A parallelizing transform $\mathcal{G}$ turns a sequential trace $T$ into a possibly multilevel list $T'$ by exploiting independence of comparator modules and grouping those modules together into parallel lists. We write $T \rightarrow^\mathcal{G} T'$.

2.3. Semantic equivalency and minimal (parallel) execution time

Through the use of semantic preserving transforms and parallelizing transforms, we can define a semantically equivalent trace.

A multilevel trace $T''$ is said to be semantically equivalent to a sequential trace $T$ if and only if $T''$ is obtained from $T$ by a semantic preserving transform and/or a parallelizing transform. We write $T \rightarrow^\mathcal{F} T' \rightarrow^\mathcal{G} T''$ or $T \rightarrow^\mathcal{F} T' \rightarrow^\mathcal{G} T''$ where $\mathcal{F}$ and/or $\mathcal{G}$ may be the identity transformation. We write $T'' \equiv s T$ (to be read as $T''$ is semantically equivalent to $T$).

Consider the trace $T = (1 \ 5 \ 7 \ 1 \ 2 \ 3 \ 6)$. The following are examples of traces semantically equivalent to $T$:

1. $(1 \ 5 \ 7 \ 2 \ 3 \ 6)$ (the transformation involves commuting 1 with 7 and 5, then by idempotence the 1 may be deleted);
(2) \( (1 \ 5 \ 7 \ 1 \ 2 \ (3 \ 5) \ 6) \) \{by idempotence, we may introduce a second 5 and commute it with 7, 1, 2, 3, but not 6; 3 and 5 may be executed in parallel due to their independence\};

(3) \( ((1 \ 5 \ 7 \ (1 \ 2 \ 3) \ 6) \) \{note that (1 2 3) is in sequence. 1, 5, 7 and the sequence (1 2 3) are in parallel\};

whereas the following are not semantically equivalent to \( T\):

(4) \( (1 \ 5 \ 7 \ 1 \ 2 \ 3) \) \{6 is missing\};

(5) \( (1 \ 5 \ 7 \ (13) \ 2 \ 6) \) \{2 is not commutative with 3\};

(6) \( (1 \ 5 \ 7 \ ((1 \ 2) \ 6 \ 3)) \) \{3 is not independent from (1 2) and cannot be executed in parallel with it\}.

2.4. Subsequences

Our proof of an optimally concurrent transform for sorting networks involves the analysis of a subsequence of a trace. A subsequence of a sequential trace \( T = (C_1, C_2, \ldots, C_n) \) is of the type \( (C_{i_1}, C_{i_2}, \ldots, C_{i_k}) \) where \( \forall i \leq j \leq n \) and \( i < j < j+1 \).

Therefore, a list of any comparator modules taken from a trace is a subsequence if the original sequential order of those modules is preserved in that subsequence.

The definition of a linearly dependent sequence can now easily be applied to subsequences to provide the notion of a linearly dependent subsequence. Additionally, the maximum linearly dependent subsequence of a trace \( T \) is defined as a linearly dependent subsequence of \( T \) with the maximum length (i.e. the maximum number of dependent comparator modules). This length will be denoted by \( l(T) \).

2.5. The optimal transformation

The objective of an optimally concurrent transform \( T'' \), which is semantically equivalent to the sequential trace \( T \), is to provide the minimum (parallel) execution time of \( T \) such that

\[
\text{EXECETIME}(\text{FLAG } T'') = \min\{\text{EXECETIME}(\text{FLAG } T') | T' = s T\}.
\]

The optimal transformation as proposed in [8] consists of two functions: RAVEL and RAVEL-TRANS.

The function \( \text{RAVEL}(I, T') \) adds a comparator module \( I \) to a two-level trace \( T' \) as described below. Since \( T' \) is a two-level trace, \( T'' \) is a sequential list consisting of parallel lists \( (T' = (L_1, L_2, \ldots, L_n)) \). RAVEL places \( I \) into either:

(1) the last parallel list \( L_n \) if \( I \) is commutative with all comparator modules in \( T' \) and \( I \) is not previously encountered in \( T' \);

(2) some parallel list \( L_i \) such that \( I \) is dependent on some comparator module in parallel list \( L_{i+1} \), \( I \) is commutative with all comparator modules in \( L_j \), \( j < i \) and \( I \) is not previously encountered in \( T' \);

(3) a new parallel list at the left hand side of \( T' \) is created to hold \( I \) if \( I \) is dependent on some comparator module in \( L_1 \) or \( I \) is the first element raveled into \( T' \);

(4) \( I \) is encountered in \( T'' \) and by idempotence \( I \) is not added to \( T' \).
As an example, consider (RAVEL 7 ((1) (2) (3 6))). The new $T'$ is ((1) (2 7) (3 6)). The 7 is commutative with 1 and 2, but is dependent on 6 because 7 accesses array elements 7 and 6. Hence, 7 is placed in parallel with 2.

(RAVEL-TRANSFORM $T$) ravel the sequential trace $T$ element after element by recursively calling RAVEL. The comparator modules from $T$ are raveled into $T'$ starting from the rightmost module in $T$ and working leftward.

Using RAVEL-TRANSFORM on the trace $T = (1 5 7 1 2 3 6)$ results in the concurrent trace $T' = ((1) (5 7 2) (3 6))$.

3. The main result

In this section, we show that RAVEL-TRANSFORM is “optimal” in the sense that it produces a parallel trace with minimal execution time, from a given sequential trace. Although a similar result was shown in [2, 8] using a mechanical theorem prover, we feel, however, that a mathematical proof provides more insight into the problem vs. a mechanical proof where certain key ideas are hidden in a black box, namely the theorem prover. To show the main result, a sequence of lemmas are required. First, we have the following easily shown lemma.

Lemma 3.1. RAVEL-TRANSFORM is a semantic preserving and parallelizing transform.

Proof. The proof follows directly from the definition of RAVEL-TRANSFORM. □

In what follows, we show that for every multilevel list, there exists a semantically equivalent 2-level list which has exactly the same execution time. For ease of expression, ⟨...⟩ and (,...) are used to denote sequential and parallel lists, respectively (i.e., terms in ⟨...⟩ and (,...) are executed in sequence and in parallel, respectively). Before presenting the detailed proof, we consider examples of converting a 3-level list into a semantically equivalent 2-level list with the same execution time. This, hopefully, will allow the reader to have a better understanding of the proof. Now, consider a 3-level sequential list ⟨(((a b)(c d))((e f)(g h)))⟩, where $a, b, c, \ldots, h$ are comparator modules. In this list, one can see that

- ⟨(a b)(c d)⟩ and ⟨(e f)(g h)⟩ are executed in sequence,
- ⟨a b⟩ and ⟨c d⟩ (also ⟨e f⟩ and ⟨g h⟩) are executed in parallel,
- $a$ and $b$ (also $c$ and $d$, $e$ and $f$, $g$ and $h$) are executed in sequence.

So, ⟨(a b)(c d)⟩ ≡ s (⟨a c⟩⟨b d⟩) and ⟨(e f)(g h)⟩ ≡ s (⟨e g⟩⟨f h⟩). As a result, ⟨(((a b)-(c d))((e f)(g h)))⟩ ≡ s (⟨a c⟩⟨b d⟩⟨e g⟩⟨f h⟩), i.e.,

\[
\left(\left(\langle a \ b \rangle \ \langle e \ f \rangle \right) \ \langle c \ d \rangle \ \langle g \ h \rangle \right) \ \equiv \ s \left(\langle a \ \langle c \ \langle b \ d \rangle \ \langle e \ \langle g \ \langle f \ h \rangle \rangle \rangle \rangle \right).
\]
Now, consider a 3-level parallel list \(((a \ b)(c \ d))((e \ f)(g \ h))\). In this case, \((a \ h)\) and \((c \ d)\) (also \((e \ f)\) and \((g \ h)\)) are executed sequentially, while \(a, h\) (also \(c, d\) etc.) are executed in parallel. It is then reasonably easy to see that \(((a \ b)(c \ d)) - (e \ f)(g \ h)) = s ((a \ h e f)(c \ d g h)), i.e.,

\[
\begin{smallarray}
\begin{array}{c}
(a \ b)
\end{array}
\begin{array}{c}
(c \ d)
\end{array}
\begin{array}{c}
(e \ f)
\end{array}
\begin{array}{c}
(g \ h)
\end{array}
\end{smallarray}
\begin{smallarray}
\begin{array}{c}
a\ c
\end{array}
\begin{array}{c}
b\ d
\end{array}
\begin{array}{c}
e\ g
\end{array}
\begin{array}{c}
f\ h
\end{array}
\end{smallarray}
\]

In general, we have the following.

**Lemma 3.2.** For every multilevel list, there exists a semantically equivalent 2-level list with the same execution time.

**Proof.** The proof is done by induction on the number of levels \((n)\). Let \(L\) be an \(n\)-level list.

*(Induction base):* The case \(n = 1\) or \(2\) is trivial.

*(Induction hypothesis):* Assume that the assertion is true for \(k\)-level lists, where \(k \geq 2\). In other words, given an arbitrary \(k\)-level list \(L\), there exists a 2-level list \(L'\) such that \(L \equiv s L'\) and \(\text{EXECETIME}(L) = \text{EXECETIME}(L')\).

*(Induction step):* Consider the case when \(n = k + 1\). Let \(L\) be a level (depth) \(k + 1\) sublist of \(L\), i.e., \(L\) consists of three alternations of sequential and parallel executions. In what follows, we show how to rewrite the 3-level list \(L\), using semantic preserving operations, as a 2-level list with the same execution time. Now, one of the following two cases is true.

**Case 1 (\(L\) is sequential):** Let \(L = (L_1 L_2 \ldots L_m)\), where \(L_1, L_2, \ldots, L_m\) are executed sequentially and by definition we have \(\text{EXECETIME}(L) = \sum_{i=1}^{m} \text{EXECETIME}(L_i)\). Each \(L_i\) is a parallel list which is of the form \((L_{i,1} L_{i,2} \ldots L_{i,d_i})\), for some \(d_i \geq 1\), and each \(L_{i,j}\) is a sequential list of the form \((C_{i,j,1} \ldots C_{i,j,f_{i,j}})\), for some \(f_{i,j} \geq 1\), where each \(C_{i,j,l}\) is a comparator module which requires a unit execution time. Without loss of generality, we may further assume that for every \(i, j\) and \(j'\), \(f_{i,j} = f_{i,j'} = h_i\) (i.e., for every \(i\), \(L_{i,j} \equiv s L_{i,j'}\) contains the same number of comparator modules). (Otherwise, an equivalent list satisfying this requirement can easily be constructed using idempotence operations.) Note that \(\text{EXECETIME}(L) = \sum_{i=1}^{m} h_i\). At this moment, one should be able to observe that for every \(1 \leq i \leq m\) and \(1 \leq l \leq h_i\), \(C_{i,1,l}, C_{i,2,l}, \ldots, C_{i,d_i,l}\) can be executed in parallel. Based on this observation, it is reasonably easy to see that the 2-level list \(L = \ldots \langle C_{i,1,1} \ldots C_{i,1,d_i} \rangle \langle C_{i,2,1} \ldots C_{i,2,d_i} \rangle \ldots \langle C_{i,h_i,1} \ldots C_{i,h_i,d_i} \rangle \ldots \ldots \langle C_{k,1,1} \ldots C_{k,1,d_k} \rangle \langle C_{k,2,1} \ldots C_{k,2,d_k} \rangle \ldots \langle C_{k,h_k,1} \ldots C_{k,h_k,d_k} \rangle \ldots \) is semantically equivalent to \(L\). Furthermore, \(\text{EXECETIME}(L) = \sum_{i=1}^{m} h_i\) (\(= \text{EXECETIME}(L)\)). Hence, there exists a \(k\)-level list which is semantically equivalent to \(L\) and both have the same execution time. The assertion then follows directly from the induction hypothesis.
Case 2 ($L$ is parallel): Let $L = \langle L_1 L_2 \ldots L_m \rangle$, where $L_1$, $L_2$, \ldots, $L_m$ are executed in parallel and by definition we have $\text{EXECTIME}(L) = \max_{1 \leq i \leq m} \{ \text{EXECTIME}(L_i) \}$. Each $L_i$ is a sequential list of the form $(L_{i,1} L_{i,2} \ldots L_{i,d_i})$, for some $d_i \geq 1$, while each $L_{i,j}$ is a parallel list of the form $(C_{i,j,1} \ldots C_{i,j,f_i})$, for some $f_i \geq 1$.

Without loss of generality, we can assume that $d_i = d$, for all $i$. Note that the execution time for $L$ is then clearly given as $\text{EXECTIME}(L) = \max_{1 \leq i \leq m} \{ d_i \} = d$. Let $F = \langle L_1 L_2 \ldots L_d \rangle$ is just $(I, is semantically equivalent to $L$. Furthermore, we have that $\text{EXECTIME}(L') = d$ ($= \text{EXECTIME}(L)$). Hence, there exists a $k$-level list which is semantically equivalent to $L$ with the same execution time. The assertion then follows directly from the induction hypothesis. □

In what follows, we show that in an attempt to parallelize a sequential trace $T$, a maximum linearly dependent subsequence of $T$ plays a crucial role. More precisely, the length ($l(T)$) of a maximum linearly dependent subsequence provides a lower bound for the parallel execution time under a semantic preserving transformation. To show this, first recall that a linearly dependent subsequence $C_1, C_2, \ldots, C_k$ of a sequence trace $T$ is a subsequence in which $C_i - C_{i+1}, 1 \leq i \leq k$. In other words, $C_i$ and $C_{i+1}, 1 \leq i < k$, cannot commute with each other. Also recall that a semantically equivalent trace of $T$ is obtained by applying a finite number of idempotence and/or commutative operations of $T$. As a consequence, the order $C_1, C_2, \ldots, C_k$ must be preserved in any semantically equivalent trace (of $T$) (otherwise, an invalid commutative operation involving $C_i$ and $C_{i+1}$ would have been performed). Therefore, we have the following.

**Lemma 3.3** Let $C_1, C_2, \ldots, C_k$ be a linearly dependent subsequence of a sequential trace $T$. For every sequential trace $T'$, if $T' \equiv s T$, then $T'$ must also contain $C_1, C_2, \ldots, C_k$ as a subsequence.

**Lemma 3.4.** Given an arbitrary sequential trace $T$, let $L$ be a semantically equivalent multilevel list (of $T$). Then the (parallel) execution time of $L$ is greater than or equal to $l(T)$.

**Proof.** According to Lemma 3.2, there exists a 2-level semantically equivalent list $L'$ with the same execution time. Let $L' = (L_1 L_2 \ldots L_m)$, for some $m$, and let $L_i = \langle C_{i,1} C_{i,2} \ldots C_{i,d_i} \rangle, 1 \leq i \leq m$. Then $\text{EXECTIME}(L') = m$. Let $C_1, C_2, \ldots, C_k$ be a maximum linearly dependent subsequence of the sequential trace $L'$. Clearly, $l(T) = k$. According to Lemma 3.3, any corresponding sequential trace of $L'$ must also contain $C_1, C_2, \ldots, C_k$. Now suppose $k > m$. According to the pigeon-hole principle, there must exist $i, 1 \leq i < k$, such that $C_i$ and $C_{i+1}$ are in some $L_t$, for some $1 \leq t \leq m$. Hence, $C_i$ and $C_{i+1}$ are independent—a contradiction. This completes the proof of the lemma. □
Lemma 3.5. Given a sequential trace \( T \), \textsc{ravel-transform} produces a 2-level list with execution time \( l(T) \).

Proof. Let \( L = (L_1, L_2, \ldots, L_k) \) be the output of \textsc{ravel-transform} on \( T \). Clearly, \( \text{EXECTIME}(L) = k \). According to the definition of \textsc{ravel-transform}, there must exist some \( C_1, C_2, \ldots, C_k \) in \( L_1, L_2, \ldots, L_k \), respectively, such that \( C_1, C_2, \ldots, C_k \) are linearly dependent. Consequently, \( l(T) \) (the length of the maximum linearly dependent subsequence) is no less than \( k \). This together with the result of Lemma 3.4 yield the result that \textsc{ravel-transform} produces a list with execution time \( l(T) \). □

The following theorem follows immediately from Lemmas 3.4 and 3.5.

Theorem 3.6. Given an arbitrary sequential trace \( T \), the minimum (parallel) execution time of \( T \) is \( l(T) \).

Corollary 3.7. \textsc{ravel-transform} is an optimal parallel transform.

4. Discussion

The motivation for this paper stemmed from a critical analysis of a systolic design methodology as proposed in [3]. The methodology utilizes an enhanced version of the Ravel Transform to parallelize sequential systolic execution traces. The transformation is a fundamental step in their design process. As such, it is desirable to provide a general optimality proof for a given class of algorithms as opposed to the case-by-case approach as proposed in [3]. This paper has provided such a general proof for the Ravel Transform as it applies to the class of algorithms expressed through sorting networks.

Acknowledgment

We would like to thank Anil Sharma for many helpful discussions concerning this paper.

References


