# Multi-threaded composition of finite-state automata

**Bryan Jurish**
Berlin-Brandenburg Academy of Sciences
Jägerstr 22/23 · 10117 Berlin · Germany
jurish@bbaw.de

**Kay-Michael Würzner**
University of Potsdam · Psychology Dept.
Karl-Liebknecht-Str. 24-25 · 14476 Potsdam · Germany
wuerzner@uni-potsdam.de

## Abstract

We investigate the composition of finite-state automata in a multiprocessor environment, presenting a parallel variant of a widely-used composition algorithm. We provide an approximate upper bound for composition speedup of the parallel variant with respect to serial execution, and empirically evaluate the performance of our implementation with respect to this bound.

## 1 Introduction

Finite-state automata[1] allow for efficient implementations in terms of directed graphs with designated initial and final states, as well as labeled edges facilitating efficient storage and lookup. Complex systems based on (weighted) finite-state automata have been successfully used in language processing (Mohri et al., 2007), image compression (Culik II and Kari, 1993), computational biology (Krogh et al., 1994), and many other applications. Composition is a binary operation on finite-state transducers (FSTs) which creates transitions for matching output- and input-labels of the outgoing transitions of two operand states. It is an important operation in both compile-time construction (where it may be employed e.g. to combine different levels of representation) and – since lookup may be considered a special case of composition – run-time querying of the aforementioned systems.

Despite the increasing trend towards multiprocessor systems and the resulting demand for efficient parallel implementations for common operations (Sutter, 2005), no generic parallel algorithm for the composition of FSTs has yet been established, although many efforts have been made to improve composition performance in special cases. In Holub and Štekr (2009), a parallel implementation for the case of string lookup in a deterministic finite-state acceptor (FSA) is presented. A generalization to $n$ operands which prevents the construction of large intermediate results is given in Allauzen and Mohri (2009). A good deal of work has focussed on *dynamic*, *on-the-fly*, or *lazy* implementations (Hori et al., 2004; Cheng et al., 2007; Mohri et al., 2007), in which the composition of FSTs is only partly computed, new states and transitions being added to the result only when necessary.

In this article, we present a parallel variant of a widely-used composition algorithm (Hanneforth, 2004; Allauzen et al., 2007, etc.) which can make use of multiprocessor architectures by employing multiple concurrent threads of execution. We provide an approximate upper bound for composition speedup using a state-wise parallel algorithm with respect to serial execution, and empirically evaluate the performance of our implementation with respect to this bound.

## 2 Preliminaries

### 2.1 Definitions

**Definition 1** (FST). *A* finite-state transducer[2] *is a 6-tuple* $\mathcal{T} = \langle \Sigma, \Gamma, Q, q_0, F, E \rangle$ *with* $\Sigma$ *a finite input alphabet,* $\Gamma$ *a finite output alphabet,* $Q$ *a finite set of automaton states,* $q_0 \in Q$ *the designated initial state,* $F \subseteq Q$ *a set of final states, and* $E \subseteq Q \times Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\})$ *a set of transitions.*

For a transition $e = (q_1, q_2, a, b) \in E$, we denote by $\mathrm{p}[e]$ its source state $q_1$, by $\mathrm{n}[e]$ its destination state $q_2$, by $\mathrm{i}[e]$ its input label $a$, and by $\mathrm{o}[e]$

[1]Where appropriate, we use the term *automata* as a generic subsuming both *acceptors* and *transducers*.

[2]Here and in the sequel, we restrict our attention to *unweighted* automata. The algorithms described here trivially extend to the weighted case. In fact, the implementations used in the current experiments (Sec. 4) operate on weighted automata.

its output label $b$. A finite-state acceptor (FSA) can be regarded as an FST with $\Sigma = \Gamma$ and $\mathrm{i}[e] = \mathrm{o}[e]$ for all $e \in E$.

A *path* $\pi$ is a sequence $e_1 \dots e_n$ of $n$ transitions such that $\mathrm{n}[e_i] = \mathrm{p}[e_{i+1}]$ for $1 \leq i < n$. We denote by $|\pi|$ the length of $\pi$: $|e_1 \dots e_n| = n$. Extending the notation for transitions, we define the source and destination states of a path as $\mathrm{p}[\pi] = \mathrm{p}[e_1]$ and $\mathrm{n}[\pi] = \mathrm{n}[e_n]$, respectively. The input label string $\mathrm{i}[\pi]$ yielded by a path $\pi$ is the concatenation of the input labels of its transitions: $\mathrm{i}[\pi] = \mathrm{i}[e_1]\mathrm{i}[e_2] \dots \mathrm{i}[e_n]$; the output label string $\mathrm{o}[\pi]$ is defined analogously.

For $q \in Q$, $x \in \Sigma^*$, $y \in \Gamma^*$, and $R \subseteq Q$, $\Pi(q, x, y, R)$ denotes the set of paths from $q$ to some $r \in R$ with input string $x$ and output string $y$, and $\Pi(q, R) = \bigcup_{x \in \Sigma^*, y \in \Gamma^*} \Pi(q, x, y, R)$ denotes the set of paths originating at $q$ and ending at some $r \in R$. A state $r \in Q$ is said to be *accessible from* a state $q \in Q$ if there exists a path $\pi$ with $\mathrm{p}[\pi] = q$ and $\mathrm{n}[\pi] = r$; $r$ is *accessible* if it is accessible from $q_0$. $[\![\mathcal{T}]\!] \subseteq \Sigma^* \times \Gamma^*$ denotes the string relation associated with $\mathcal{T}$, and is defined as the set of input- and output-string pairs labelling successful paths in $\mathcal{T}$: $[\![\mathcal{T}]\!] = \{(\mathrm{i}[\pi], \mathrm{o}[\pi]) : \pi \in \Pi(q_0, F)\}$

**Definition 2** (Depth). *For any accessible $q \in Q$, $\mathrm{depth}(q)$ denotes the depth of $q$, defined as the length of the shortest path from the initial state to $q$:*

$$\mathrm{depth}(q) = \begin{cases} 0 & \textit{if } q = q_0 \\ \min_{\pi \in \Pi(q_0, \{q\})} |\pi| & \textit{otherwise} \end{cases}$$

*The depth of a transducer $\mathcal{T}$ is defined as the maximum depth over all its accessible states:* $\mathrm{depth}(\mathcal{T}) = \max_{q \in Q : \Pi(q_0, \{q\}) \neq \emptyset} \mathrm{depth}(q)$.

## 2.2 Composition

Composition is a binary operation on FSTs $\mathcal{T}_1$ and $\mathcal{T}_2$ which share an "inner" alphabet. Informally, the composition operation matches transitions from $\mathcal{T}_1$ to transitions from $\mathcal{T}_2$ if the corresponding output and input labels coincide. Formally:

**Definition 3** (Composition of FSTs). *Given two FSTs $\mathcal{T}_1 = \langle \Sigma, \Gamma, Q_1, q_{0_1}, F_1, E_1 \rangle$ and $\mathcal{T}_2 = \langle \Gamma, \Delta, Q_2, q_{0_2}, F_2, E_2 \rangle$, the composition of $\mathcal{T}_1$ and $\mathcal{T}_2$ is denoted by $\mathcal{T}_1 \circ \mathcal{T}_2$, and is itself an FST whose string relation is the composition of the string relations of $\mathcal{T}_1$ and $\mathcal{T}_2$, $[\![\mathcal{T}_1 \circ \mathcal{T}_2]\!] = [\![\mathcal{T}_1]\!] \circ [\![\mathcal{T}_2]\!]$, i.e. for all $x \in \Sigma^*$, $y \in \Delta^*$, $(x, y) \in [\![\mathcal{T}_1 \circ \mathcal{T}_2]\!]$ if and only if there exists some $z \in \Gamma^*$ such that $(x, z) \in [\![\mathcal{T}_1]\!]$ and $(z, y) \in [\![\mathcal{T}_2]\!]$. Further, $\mathcal{C} = \langle \Sigma, \Delta, (Q_1 \times Q_2), E, (q_{0_1}, q_{0_2}), (F_1 \times F_2) \rangle$ is such an FST, $[\![\mathcal{C}]\!] = [\![\mathcal{T}_1 \circ \mathcal{T}_2]\!]$, where:*

$$E = \bigcup_{\substack{(q,r,a,b) \in E_1 \\ (s,t,b,c) \in E_2}} \{((q, s), (r, t), a, c)\} \tag{1}$$

The construction above is only correct if $\mathcal{T}_1$ and $\mathcal{T}_2$ are $\varepsilon$-free on their output and input tapes, respectively. The generalization to $\varepsilon$-transitions has been discussed e.g. by Mohri (2009). Since the generalization can be reduced to the above construction applied to $\varepsilon$-free WFSTs, we ignore it here in the interest of clarity.

The most common serial algorithm (Mohri, 2009) for computing the composition of two WFSTs is presented here as Algorithm 1, and can be considered a variant of the standard intersection algorithm for unweighted FSAs as described by Hopcroft and Ullman (1979). Unlike the "brute force" construction of Definition 3, the algorithm generates only those states and transitions of the output automaton which are accessible from the initial state. In this manner, the algorithm manages to avoid the combinatorial explosion of states and transitions implicit in Definition 3 in the overwhelming majority of cases.

## 2.3 Amdahl's Law

Amdahl's law (Amdahl, 1967) describes the theoretical bound on speeding up a program in terms of improvements made to specific parts of that program. In particular, it can be used to predict the maximum speedup resulting from executing specific parts of a program in parallel on a multiprocessor architecture.

$$S_N = \frac{1}{(1 - P) + \frac{P}{N}} \tag{2}$$

Equation (2) states that the maximum speedup $S_N$ of a parallel program is the inverse sum of the serial proportion of that program (those parts which cannot be executed in parallel, $1 - P$) and the parallel proportion $P$ divided by the number of processors $N$. As the number of available processors increases, $\frac{P}{N}$ approaches zero and $S_N$ approaches $\frac{1}{1-P}$.

## 3 State-wise parallelization

Using a first-in-first-out protocol for the visitation queue $V$, Algorithm 1 effectively performs a

**Algorithm 1:** COMPOSE: serial composition of $\varepsilon$-free FSTs

```
Input: $\mathcal{T}_1 = \langle \Sigma, \Gamma, Q_1, q_{0_1}, F_1, E_1 \rangle$ an FST
Input: $\mathcal{T}_2 = \langle \Gamma, \Delta, Q_2, q_{0_2}, F_2, E_2 \rangle$ an FST
1  function COMPOSE($\mathcal{T}_1, \mathcal{T}_2$)
2      $Q, F, E \leftarrow \emptyset$                                      /* initialize */
3      $V \leftarrow \{(q_{0_1}, q_{0_2})\}$                                 /* visitation queue */
4      while $V \neq \emptyset$ do
5          $(q_1, q_2) \leftarrow \texttt{pop}(V)$                            /* visit state */
6          $Q \leftarrow Q \cup \{(q_1, q_2)\}$
7          if $(q_1, q_2) \in F_1 \times F_2$ then                          /* final state */
8              $F \leftarrow F \cup \{(q_1, q_2)\}$
9          for $(e_1, e_2) \in E[q_1] \times E[q_2]$ with $\mathrm{o}[e_1] = \mathrm{i}[e_2]$ do       /* align transitions */
10             $E \leftarrow E \cup \{((q_1, q_2), (\mathrm{n}[e_1], \mathrm{n}[e_2]), \mathrm{i}[e_1], \mathrm{o}[e_2])\}$
11             if $(\mathrm{n}[e_1], \mathrm{n}[e_2]) \notin Q$ then
12                 $\texttt{push}(V, (\mathrm{n}[e_1], \mathrm{n}[e_2]))$              /* enqueue for visitation */
13     return $\mathcal{C} = \langle \Sigma, \Delta, Q, (q_{0_1}, q_{0_2}), F, E \rangle$
```

breadth-first traversal of the output automaton $\mathcal{C}$. Pairs of destination states for aligned transitions are enqueued only if they have not yet been visited. Our parallelization scheme is based on concurrent processing of multiple result states – multiple concurrent executions of the while loop at lines 4-12. Such state-wise parallelization is a common approach for breadth-first traversals of graphs in a multiprocessor environment (Roosta, 2000).

### 3.1 An approximate upper bound for state-wise parallel speedup

In this section, we investigate the potential speedup resulting from a state-wise parallelization of Algorithm 1 as described above. We assume that Algorithm 1 constructs an automaton $\mathcal{C} = \langle \Sigma, \Gamma, Q, q_0, F, E \rangle$, and show that the maximum speedup of a state-wise parallel composition algorithm depends on the topological properties of $\mathcal{C}$, specifically on its state-to-depth ratio.

We call a single evaluation of lines 4-12 a *visitation* of the state $q = (q_1, q_2) \in Q$, and we call a state *discovered* when it has been pushed to the visitation queue during the visitation of a predecessor at line 12. For each $q \in Q$, let $t_0(q)$ represent the time at which the visitation of $q$ begins, let $t_1(q)$ be the earliest time at which the visitation of $q$ has completed. We assume a strict breadth-first visitation order on states: for all $q, q' \in Q$,

$$\mathrm{depth}(q) < \mathrm{depth}(q') \implies t_1(q) \leq t_0(q') \quad (3)$$

i.e. visitation of all states at depth $d$ must have completed before any state with depth $d' > d$ can itself be visited. In order to approximate a worst-case scenario for state-wise parallelization, we assume that the duration of a visitation $t_{\mathrm{visit}}(q)$ is independent of the state $q$ being visited, defining this constant as our elementary time unit:

$$\forall q \in Q, (t_1(q) - t_0(q)) = t_{\mathrm{visit}}(q) = 1 \quad (4)$$

We restrict our attention to the execution of the visitation loop of lines 4-12, ignoring the constant administrative overhead of lines 2-3 and 13, and assume the convention that visitation of the initial state begins at $t_0(q_0) = 0$.

**Lemma 1** (Serial composition running time). *Let $\mathcal{C} = \langle \Sigma, \Gamma, Q, q_0, F, E \rangle$ be an FST constructed by Algorithm 1. Serial execution of the algorithm requires exactly $t_{\mathrm{serial}} = |Q|$ time units.*

*Proof.* Since each state is visited exactly once and no two states can be visited concurrently, the algorithm terminates after exactly $|Q|$ visitations. No code is executed between visitations, so $t_{\mathrm{serial}} = \sum_{q \in Q} t_{\mathrm{visit}}(q) = |Q|$ by Equation 4. $\square$

**Lemma 2** (Parallel state visitation bound). *Let $\mathcal{C} = \langle \Sigma, \Gamma, Q, q_0, F, E \rangle$ be an FST constructed by Algorithm 1 executed in state-wise parallel fashion using an unbounded number of processors $N \geq |Q|$. The completion time of any given state's visitation is determined by that state's depth:*

$$\forall q \in Q, t_1(q) = \mathrm{depth}(q) + 1$$

*Proof.* We proceed by induction over $\mathrm{depth}(q)$. For $\mathrm{depth}(q) = 0$, it must be that $q = q_0$ and

$t_0(q) = t_0(q_0) = 0$ by convention. Since visitation time is constant, $t_1(q) = t_0(q) + t_{\text{visit}}(q) = 1 = \text{depth}(q) + 1$, so the lemma holds.

For the inductive step, consider an arbitrary $q \in Q$ with $\text{depth}(q) = d$ and assume that the lemma holds for all $q' \in Q$ with $\text{depth}(q') < d$. Since $\text{depth}(q) = d$, there exist $p \in Q$ and $e \in E$ with $\text{p}[e] = p$, $\text{n}[e] = q$, and $\text{depth}(p) = d - 1$ by Definition 2. By inductive hypothesis, $t_1(p) = \text{depth}(p) + 1 = d$. Since we have $N \geq Q$ processors available and at most $|Q|$ visitations to perform, we can begin processing $q$ as soon as it is discovered during the visitation of $p$: $t_0(p) < t_0(q) \leq t_1(p)$, but the strict breadth-first visitation constraint of Equation (3) dictates that visitation of $q$ cannot begin until all states of lesser depth have been visited, so $t_0(q) = t_1(p) = d$ and $t_1(q) = t_0(q) + t_{\text{visit}}(q) = d + 1$ by Equation (4). $\square$

**Lemma 3** (State-wise parallel composition running time). *Let $\mathcal{C} = \langle \Sigma, \Gamma, Q, q_0, F, E \rangle$ be an FST constructed by Algorithm 1. State-wise parallel execution of the algorithm with $N \geq Q$ available processors requires exactly $t_{\text{parallel}:N} = 1 + \text{depth}(\mathcal{C})$ time units.*

*Proof.* The algorithm terminates when all states have been visited at time $t_1(\mathcal{C}) = \max_{q \in Q} t_1(q)$. By Lemma 2, all states in each depth-slice $\text{depth}^{-1}(d) \subseteq Q$ are visited concurrently, completing at time $d + 1$. Since $\text{depth}(\mathcal{C}) = \max_{q \in Q} \text{depth}(Q)$ by Definition 2, $t_{\text{parallel}:N} = t_1(\mathcal{C}) = 1 + \text{depth}(\mathcal{C})$. $\square$

Having derived approximate running times of both serial and parallel executions, we can now compute the maximum speedup of a state-wise parallel execution.

**Theorem 1** (Maximum speedup of state-wise parallelization). *The maximum speedup $S_{\max}$ of a state-wise parallel execution of Algorithm 1 constructing an output FST $\mathcal{C} = \langle \Sigma, \Gamma, Q, q_0, F, E \rangle$ with respect to serial execution is approximated for $N \geq |Q|$ by:*

$$S_{\max} = \frac{t_{\text{serial}}(\mathcal{C})}{t_{\text{parallel}:N}(\mathcal{C})} = \frac{|Q|}{1 + \text{depth}(\mathcal{C})}$$

*Proof.* Follows from Lemmas 1 and 3. $\square$

**Corollary 1** (Composition parallelizability). *$P_{\max}$ is an approximate upper bound on the proportion of the total execution time of Algorithm 1 which can be effectively run in parallel:*

$$P_{\max} = 1 - \frac{1}{S_{\max}}$$

*Proof.* Follows from Theorem 1 and Amdahl's law (Eq. 2). $\square$

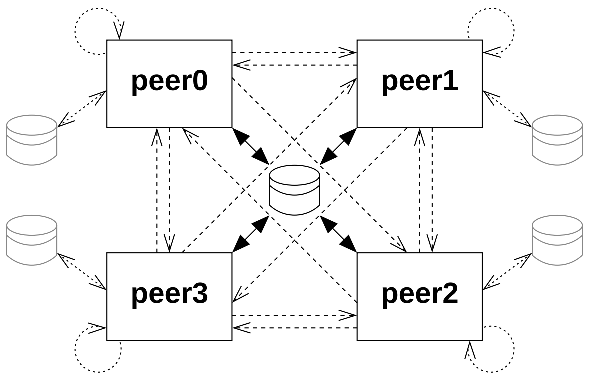

Figure 1: Data-flow diagram for the peer-to-peer parallel composition algorithm using 4 peers. Lock-free data access is displayed with dotted lines, data channels shared by exactly two peers appear as dashed lines, and globally shared data channels which can be locked by any peer are solid black.

## 3.2 Algorithm

Apart from the bounds implied by automaton topology, practical issues such as shared data structures and the necessarily associated synchronization between otherwise independent threads must be considered in the development of any parallel algorithm. Access to shared data is typically controlled by mutual exclusion locks (*mutexes*): when altering a shared data structure $s$, a thread $t$ must first lock that structure. Other threads must then wait for $t$ to unlock $s$ before they can access it themselves. Competition for mutex locks therefore has a strong impact on the overall performance of multi-threaded implementations, since lock acquisition is an inherently synchronous operation, and thus decreases the proportion of the program which can actually be executed in parallel.

Our approach is presented as pseudo-code in Algorithm 2 and schematically depicted in Figure 1. We make use of a set of $N > 1$ "peer" threads $p_{i \in N}$, each of which simulates the execution of lines 4-12 from Algorithm 1. To minimize competition over shared data structures, each peer allocates and maintains its own local partition of

**Algorithm 2:** PPCOMPOSE: peer-to-peer parallel composition

```
Input: $\mathcal{T}_1 = \langle \Sigma, \Gamma, Q_1, q_{0_1}, F_1, E_1 \rangle$ an FST
Input: $\mathcal{T}_2 = \langle \Gamma, \Delta, Q_2, q_{0_2}, F_2, E_2 \rangle$ an FST
Input: $N \in \mathbb{N}$, number of peer threads to spawn
1  function PPCOMPOSE($\mathcal{T}_1, \mathcal{T}_2, N$)
2      for $0 \leq i, j < N$ do $V_{i,j} \leftarrow \emptyset$                /* initialize queue matrix */
3      $V_{0,r(q_{0_1}, q_{0_2})} = \{(q_{0_1}, q_{0_2})\}$
4      $n_{\text{up}} = 1$                                            /* shared queue-size counter */
5      for $0 \leq i < N$ do spawn PEER($i$)                     /* spawn peer threads */
6      wait_on_all_peers()
7      return $\mathcal{C} = \langle \Sigma, \Delta, \bigcup_{i \in N} Q'_i, (q_{0_1}, q_{0_2}), \bigcup_{i \in N} F'_i, \bigcup_{i \in N} E'_i \rangle$
8  procedure PEER($i$)
9      $Q'_i, F'_i, E'_i \leftarrow \emptyset$                             /* initialize peer-local data */
10     while true do
11         find $j$ with $0 \leq j < N$ and $V_{j,i} \neq \emptyset$
12         $v \leftarrow$ pop($V_{j,i}$)
13         if $v =$ eof then return                                 /* terminate thread */
14         if $v \notin Q'_i$ then VISIT($i, v$)                              /* visit state */
15         if $n_{\text{up}} = 1$ then
16             for $0 \leq j < N$ do push($V_{i,j}$, eof)                  /* terminate peers */
17         $n_{\text{up}} \leftarrow n_{\text{up}} - 1$                                 /* update shared counter */
18 procedure VISIT($i, (q_1, q_2)$)
19     $Q'_i \leftarrow Q'_i \cup \{(q_1, q_2)\}$
20     if $(q_1, q_2) \in F_1 \times F_2$ then                                    /* final state */
21         $F'_i \leftarrow F'_i \cup \{(q_1, q_2)\}$
22     for $(e_1, e_2) \in E[q_1] \times E[q_2]$ with $\text{o}[e_1] = \text{i}[e_2]$ do        /* align transitions */
23         $E'_i \leftarrow E'_i \cup \{((q_1, q_2), (\text{n}[e_1], \text{n}[e_2]), \text{i}[e_1], \text{o}[e_2])\}$
24         $n_{\text{up}} \leftarrow n_{\text{up}} + 1$                                 /* update shared counter */
25         push($V_{i,r(\text{n}[e_1],\text{n}[e_2])}, (\text{n}[e_1], \text{n}[e_2])$)          /* enqueue for visitation */
```

the output automaton structure ($Q'_i$, $F'_i$, $E'_i$), and the visitation queue $V$ is replaced by an $(N \times N)$ matrix of peer-to-peer local message queues: $V_{i,j}$ contains messages originating at peer $p_i$ and destined for peer $p_j$.

This technique relies for its correctness on the prior specification of a partitioning function $r : Q_1 \times Q_2 \to N$ over states of the result automaton, used to determine which peer is responsible for visiting any such state. For the experiments described in section 4, our input automata provided injective functions $[\![\cdot]\!]_1 : Q_1 \to \mathbb{N}$ and $[\![\cdot]\!]_2 : Q_2 \to \mathbb{N}$, and we used the partitioning function given in Equation (5).

$$r(q_1, q_2) = \left\lfloor \frac{[\![q_1]\!]_1 + [\![q_2]\!]_2}{2} \right\rfloor \mod N \qquad (5)$$

Since the visitation queue $V$ is no longer a shared atomic structure, an additional shared global counter $n_{\text{up}}$ is required to keep track of the number of visitation requests currently enqueued in any $V_{i,j}$. Only when the last such request has been processed does the algorithm terminate by sending a designated message eof $\notin Q_1 \times Q_2$ to all peers at line 16.

Also worth noting is that due to the peer-wise partitioning of result data – in particular that of $Q$ into $Q'_{i \in N}$ – a new visitation request must be enqueued at line 25 for every aligned transition, and the decision of whether or not a visit is actually required deferred to the responsible peer at line 14. This can be expected to result in larger queues, correspondingly increased memory requirements, and an additional $\mathcal{O}(E - Q)$ copy operations compared to Algorithm 1. Any additional computational load – in particular the inclusion of an implicit epsilon-filter such as described by Mohri

(2009) – associated only with a single state visitation will remain confined to a single peer, and can thus only serve to improve the performance of the parallel algorithm with respect to its serial counterpart.

Our parallelization strategy does not alter the worst-case complexity of the composition algorithm, since the partitioning function may fail for certain pathological configurations. More precisely, whenever there is an $i \in \mathbb{N}$ such that $r(q_1, q_2) = i$ for all accessible $(q_1, q_2) \in Q$, then the unique peer-thread $p_i$ will be responsible for visiting all of the states of the output transducer. In this case, Algorithm 2 essentially reduces to Algorithm 1 with the worst-case complexity $\mathcal{O}(|Q_1 \times Q_2| + |E_1 \times E_2|)$, which is dominated by $\mathcal{O}(|E_1 \times E_2|)$.

Many other operations on finite-state transducers have traditional implementations in terms of a state-processing queue, exhibiting the same high-level structure as Algorithm 1. The parallelization strategy used here may also be employed in such cases, whenever a suitable state-partitioning function (analogous to $r$) can be defined. Consider for example the case of unweighted acceptor determinization as presented by Hopcroft and Ullman (1979): output automaton states are identified by sets of prefix-equivalent states of the input automaton. A generic partitioning function for mapping state sets to peers would suffice to extend our parallelization strategy to this algorithm.

## 4 Experiment

To investigate the practical utility of our multi-threaded composition algorithm, we compared running times of Algorithms 1 and 2.

### 4.1 Materials

We began by generating a sample set of random input FSTs $\mathcal{T}_{i \in I}$. Each input $\mathcal{T}_i$ was built from a deterministic trie "skeleton" of a given size $|Q_i|$ with maximum depth 32. The trie skeleton was then augmented by adding random edges between existing states, and finally randomizing all edge labels. The target size parameters $|Q_i|$ were piecewise-uniformly distributed within exponentially sized bins such that $2^5 \leq |Q_i| \leq 2^{21}$, i.e. input automata had between 32 and 2,097,152 states. The number $e_i$ of randomly added edges was dependent on the number of states, $e_i = c_i \times |Q_i|$, where the coefficients $c_i$ were uniformly distributed over the interval $[0, 16]$. In- and output-alphabets were identical for each $\mathcal{T}_i$ with alphabet sizes uniformly distributed over the discrete set $\bigcup_{i=0}^{10}\{2^i\}$, i.e. between 2 and 1024. For each input automaton $\mathcal{T}_i$ so generated, we measured the running time $t_{\mathrm{serial},i}$ of the serial composition algorithm $\mathrm{COMPOSE}(\mathcal{T}_i^{-1}, \mathcal{T}_i)$, and retained only those samples $\mathcal{T}_i$ for which $\frac{1}{64}$ sec $\leq t_{\mathrm{serial},i} \leq 8$ sec.

We implemented Algorithms 1 and 2 in C++, using the GNU C compiler (g++ version 4.4.5), the C++ standard template library, as well as the auxiliary libraries `boost` (Schäling, 2011) and `TBB` (Reinders, 2007) for common data structures and programming idioms. Automata were represented using adjacency vectors, and the edge alignments of Algorithm 1 line 9 (respectively Alg. 2 line 22) were implemented using an optimized routine[3] for sorted edge-vectors in order to minimize running times for both conditions. The implementations were tested on a dedicated 64-bit machine with 16 logical processors running debian linux.

### 4.2 Method

For each of the 2,266 random input automata $\mathcal{T}_i$, we measured the time required[4] to compute the composition $\mathcal{C}_i = (\mathcal{T}_i^{-1} \circ \mathcal{T}_i)$ using the serial algorithm ($t_{\mathrm{serial},i}$) as well as the parallel algorithm ($t_{\mathrm{pp}:N,i}$), varying the number of peer-threads employed by the latter, $N \in \{2, 4, 8, 16\}$.[5] Each of these 11,330 invocations was iterated 8 times in order to ameliorate cache effects. Raw and mean running times were stored for each invocation, together with various structural properties of the input and result automata.

### 4.3 Results & Discussion

Inspection of the generated sample set revealed that all of the tested compositions exhibited an "embarrassingly parallel" topology: applying Corollary 1 yielded $P_{\max} > 99\%$ for all $\mathcal{C}_i$ ($\mu =$

[3]Specifically, we used a modified version of the edge alignment code from the GFSM (Jurish, 2009) library function `gfsm_automaton_compose_visit_()` to implement a generic function template shared by both serial and parallel implementations.

[4]Neither I/O nor serialization of the result transducer were included in our measurements of the running time.

[5]While not necessarily representative of FST compositions in general, restricting our attention to compositions of the form $(\mathcal{T}^{-1} \circ \mathcal{T})$ ensures that the output automaton $\mathcal{C}$ is at least as large as $\mathcal{T}$ itself, since the main diagonal of the output state-set $\mathrm{Id}(Q_{\mathcal{T}})$ will be accessible whenever $\mathcal{T}$ contains no non-accessible states.

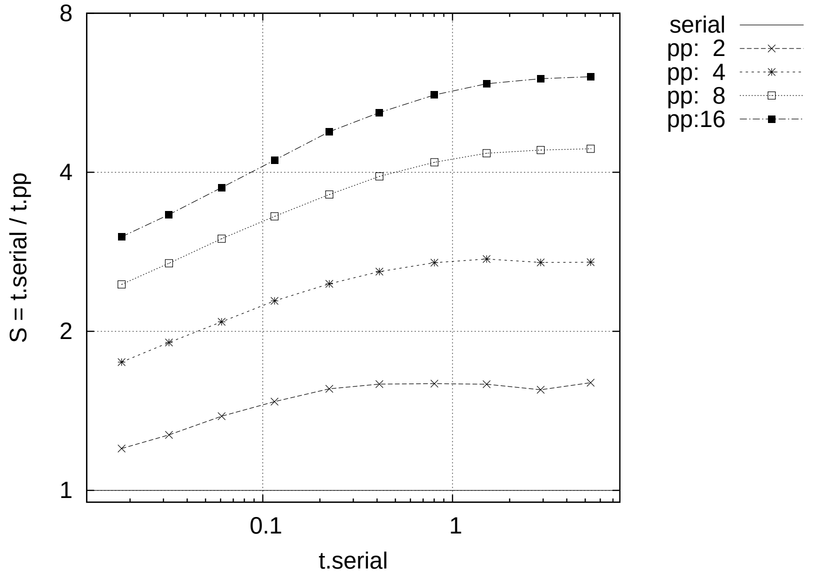

Figure 2: Observed speedup for peer-to-peer parallel composition with respect to serial running time for randomly generated automata (log-scaled).

$.9998, \sigma = 2.949 \times 10^{-4}$). Although it is certainly the case that not all FST compositions possess such characteristics (as follows from Theorem 1), the uniformly large state-to-depth ratio exhibited by our sample set makes it a particularly good testing ground for state-wise parallel processing techniques such as Algorithm 2.

Results of the runtime measurements are depicted in Figure 2, expressed as the measured speedup for the parallel implementation with respect to the serial one. The samples $i$ were sorted into 10 exponentially sized bins by serial running time $t_{\text{serial},i}$, and the speedup data $S_{N,i} = t_{\text{serial},i}/t_{\text{pp}:N,i}$ are plotted on logarithmic scales for each $n \in N$ as a piecewise linear fit over bin-wise averages.[6]

Immediately apparent from the data in Figure 2 is a typical pattern of diminishing returns for increasing values of $N$, reflecting an empirical value of $P \ll 1$. Solving Equation (2) for $P$ and applying it to the measured speedup values yields the data in Table 1, corresponding to an estimated $P = .749$ ($\sigma = .154$) over all tested parallel compositions. Interestingly, the empirically estimated values for $P$ increase monotonically with $N$, which indicates that the use of a distributed message queue did in fact reduce the competition over shared resources between peer-threads, thereby effectively reducing the serial portion of the program itself.

| $N$ | $\mu_S$ | $\sigma_S$ | $\mu_P$ | $\sigma_P$ |
|---|---|---|---|---|
| 2 | 1.474 | 0.209 | 0.615 | 0.207 |
| 4 | 2.372 | 0.423 | 0.751 | 0.116 |
| 8 | 3.585 | 0.788 | 0.806 | 0.083 |
| 16 | 4.701 | 1.156 | 0.824 | 0.066 |

Table 1: Global mean ($\mu$) and standard deviation ($\sigma$) of observed speedup ($S$) and associated degree of parallelization ($P$) for peer-to-peer parallel composition using $N$ concurrent threads.

Despite this tendency, the data from Table 1 clearly indicate that the measured performance of our implementation remained well below the upper bound given by Theorem 1 for our sample set. This discrepancy can be attributed in part to constant overhead associated with thread allocation and maintenance (cf. Section 3.2), whose effects can also be observed in the tendency of speedup to improve with increasing serial running time as seen in Figure 2. Restricting our attention to the 598 samples with $t_{\text{serial},i} \geq 1$ second, we computed an average empirical estimate of $P = .8867$ for $N = 16$ ($\sigma = .01676$). The remaining discrepancy between the empirical estimates and the "embarrassingly parallel" theoreti-

[6]Since $P_{\max} \approx 1$ for all of our test automata, we ignore the contribution of automaton topology to actual observed speedup here. For a more heterogeneous test set, it would make more sense to measure speedup relative to the respective automaton-dependent maxima, i.e. $S_{N,i}/S_{\max,N,i}$.

cal upper bounds must be attributed to the necessarily serial aspects of inter-thread communication and synchronization.

## 5 Conclusion

We have presented a generic state-wise parallel algorithm for computing the composition of $\varepsilon$-free finite-state transducers which minimizes competition for global locks by employing distributed data structures and localized communication channels. This algorithm relies on the prior specification of a partitioning function which effectively maps each state of the result automaton to the execution thread responsible for processing that state.

An approximate upper bound for composition speedup using a state-wise parallel algorithm such as that presented here was proposed and defined in terms of the state-to-depth ratio of the result automaton. Empirical investigation of the actual speedup achieved by our algorithm on a test set of "embarrassingly parallel" compositions showed that although the use of distributed data structures and communication channels was successful in reducing the serial proportion of the code when more processors were used, constant overhead and the remaining synchronizations led to the pattern of diminishing returns associated with an actual parallel execution of about $89\%$ of the program.

We are interested in evaluating the performance of our approach in other scenarios, including string lookup in (weighted) FSTs, $n$-way or "cascaded" composition, and "lazy" online constructions. We believe that the peer-to-peer parallelization strategy can also be employed to improve the performance of other common finite-state algebraic operations in a multiprocessor context.

## Acknowledgements

Research was supported by the *Deutsche Forschungsgemeinschaft* grants KL 337/12-2 and KL 955/19-1. Additionally, we would like to thank this article's anonymous reviewers for their helpful comments.

## References

Cyril Allauzen and Mehryar Mohri. 2009. N-way composition of weighted finite-state transducers. *International Journal of Foundations of Computer Science*, 20(4):613–627.

Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. OpenFst: A General and Efficient Weighted Finite-State Transducer Library. In Jan Holub and Jan Žďárek, editors, *Implementation and Application of Automata*, volume 4783 of *Lecture Notes in Computer Science*, pages 11–23. Springer, Berlin/Heidelberg.

Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS spring joint computer conference*, pages 483–485, New York, NY. ACM.

Octavian Cheng, John Dines, and Mathew Magimai Doss. 2007. A Generalized Dynamic Composition Algorithm of Weighted Finite State Transducers for Large Vocabulary Speech Recognition. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2007)*, volume 4, pages 345–348. IEEE.

Karel Culik II and Jarkko Kari. 1993. Image compression using weighted finite automata. *Computers & Graphics*, 17(3):305–313.

Thomas Hanneforth. 2004. FSM<2.0> – C++ library for manipulating (weighted) finite automata. http://www.fsmlib.org.

Jan Holub and Stanislav Štekr. 2009. On parallel implementations of deterministic finite automata. In Sebastian Maneth, editor, *Implementation and Application of Automata*, volume 5642 of *Lecture Notes in Computer Science*, pages 54–64, Berlin/Heidelberg. Springer.

John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, Reading, MA.

Takaaki Hori, Chiori Hori, and Yasuhiro Minami. 2004. Fast On-The-Fly Composition for Weighted Finite-State Transducers in 1.8 Million-Word Vocabulary Continuous Speech Recognition. *INTERSPEECH*, pages 289–292.

Bryan Jurish. 2009. libgfsm C library, version 0.0.10. `http://kaskade.dwds.de/~moocow/projects/gfsm/`.

Anders Krogh, Michael Brown, I. Saira Mian, Kimmen Sjölander, and David Haussler. 1994. Hidden Markov models in computational biology : applications to protein modeling. *Journal of Molecular Biology*, 235(5):1501–1531.

Mehryar Mohri, Fernando Carlos Pereira, and Michael Dennis Riley. 2007. Speech Recognition with Weighted Finite-State Transducers. In Larry Rabiner and Fred Juang, editors, *Handbook on Speech Processing and Speech Communication, Part E: Speech recognition*, pages 1–31. Springer, Berlin/Heidelberg.

Mehryar Mohri. 2009. Weighted automata algorithms. In Manfred Droste, Werner Kuich, and

Heiko Vogler, editors, *Handbook of Weighted Automata*, Monographs in Theoretical Computer Science, chapter Concepts of Weighted Recognizability, pages 213–254. Springer, Berlin/Heidelberg.

James Reinders. 2007. *Intel Threading Building Blocks*. O'Reilly, Sebastopol, CA.

Seyed H. Roosta. 2000. *Parallel Processing and Parallel Algorithms: Theory and Computation*. Springer, Berlin/Heidelberg.

Boris Schäling. 2011. *The Boost C++ Libraries*. XML Press.

Herb Sutter. 2005. The free lunch is over – a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3).