COMP90055 Computing Project

Parallelising Brzozowski's DFA Construction Algorithm

Joshua Clark

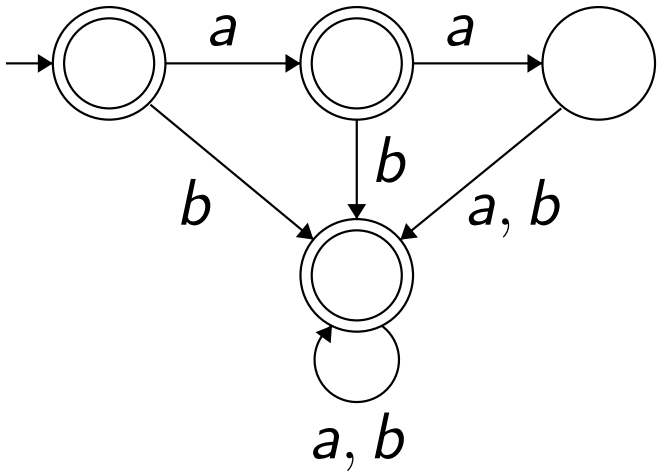
Semester 1, 2019

Research aims, put simply...

We want to take something like this:

$$(a + b)^* \ \& \ \neg(aa)$$

And produce something like this:



And see what speedup we can get by using multiple processors.

Research aims

- What performance gains, if any, are to be had from a parallel compared to a sequential implementation?
- What advantages and/or disadvantages do imperative and functional approaches to this problem offer respectively?
- Brzozowksi's algorithm depends on a notion of regular expression equivalence in order to produce minimal or near-minimal DFAs. What influence does this, and other such parameters, have on attempts at parallelisation?

(Regular) languages

We consider a language, L, to be a set of strings over some finite alphabet Σ.

$$L \subseteq \Sigma^*$$

We are concerned with *regular* languages. A language is regular if it can be described by a *regular expression* or, equivalently, if there is some *Deterministic Finite Automaton* which recognises the language.

Regular expressions

$\emptyset$, ε and $a \in \Sigma$ are all regular expressions.

If R and S are regular expressions then so are:

$$R^*$$
$$R \cdot S$$
$$R + S$$
$$R \ \& \ S$$
$$\neg R$$

Regular expressions

$\emptyset$, ε and $a \in \Sigma$ are all regular expressions.

If R and S are regular expressions then so are:

$$R*$$
$$R \cdot S$$
$$R + S$$
$$R \ \& \ S$$
$$\neg R$$

The inclusion of the highlighted cases, while still regular operations, gives what is known as *extended* regular expressions.

Derivatives of a language

Formally, the derivative with respect to a, where $a \in \Sigma, s \in \Sigma^*$, D_a is defined as:

$$D_a L = \{s \mid as \in L\}$$

Informally, we can think of the derivative of a regular language w.r.t. symbol $a \in \Sigma$ as the strings in the language starting with that symbol, but with that symbol taken away.

$$L = \{\text{aa}, \text{ab}, \text{aab}, \text{bb}\}$$

$$D_a L = \{\text{a}, \text{b}, \text{ab}\}$$

Defining regular expression derivatives

$$D_a\emptyset = \emptyset$$
$$D_a\varepsilon = \emptyset$$
$$D_a b = \begin{cases} \varepsilon, & \text{if } a = b \\ \emptyset, & \text{otherwise} \end{cases}$$

$$D_a(R^*) = D_aR \cdot R^*$$
$$D_a(R \cdot S) = D_aR \cdot S + \mathcal{N}(R) \cdot D_aS$$
$$D_a(R + S) = D_aR + D_aS$$
$$D_a(R \;\&\; S) = D_aR \;\&\; D_aS$$
$$D_a(\neg R) = \neg D_aR$$

where

$$\mathcal{N}(R) = \begin{cases} \varepsilon, & \text{if } \mathit{nullable}(R) \\ \emptyset, & \text{otherwise} \end{cases}$$

We say that a set which contains the empty string is *nullable*.

Brzozowski's algorithm

function $\textsc{BuildDFA}(q_0, \Sigma)$
    $TODO \leftarrow \{q_0\}$, $Q \leftarrow \{q_0\}$, $\delta \leftarrow \emptyset$, $F \leftarrow \emptyset$
    if $\mathcal{N}(q_0) = \varepsilon$ **then**
        $F \leftarrow \{q_0\}$
    while $TODO \neq \emptyset$ **do**
        let $q \in TODO$
        $TODO \leftarrow TODO \setminus \{q\}$
        $Q \leftarrow Q \cup \{q\}$
        for $a \in \Sigma$ **do**
            $q' \leftarrow D_a q$
            $\delta \leftarrow \delta \cup \{((q, a), q')\}$
            if $q' \notin Q \cup TODO$ **then**
                $TODO \leftarrow TODO \cup \{q'\}$
            if $\mathcal{N}(q') = \varepsilon$ **then**
                $F \leftarrow F \cup \{q'\}$
    return $(Q, \Sigma, \delta, q_0, F)$

Related work: Strauss et al. (2006, 2014, 2017)

- The authors offer a series of process-oriented specifications of Brzozowksi's DFA construction algorithm in the style of Hoare's CSP
- They have implemented one of their concurrent specifications in the language Go
- Benchmarked their implementation against a test suite of randomly generated regular expressions (of varying complexity).

Parallelism and Concurrency in Haskell

As a pure language, Haskell functions produce no side-effects unless explicitly indicated. This means that the distinction drawn between parallelism and concurrency is somewhat different from that offered by Go.

- Concurrency is a method for structuring programs and relates naturally to situations involving I/O and non-determinism
- Parallelism, on the other hand, means speeding up the execution of a (pure, deterministic) program by running it on multiple processors

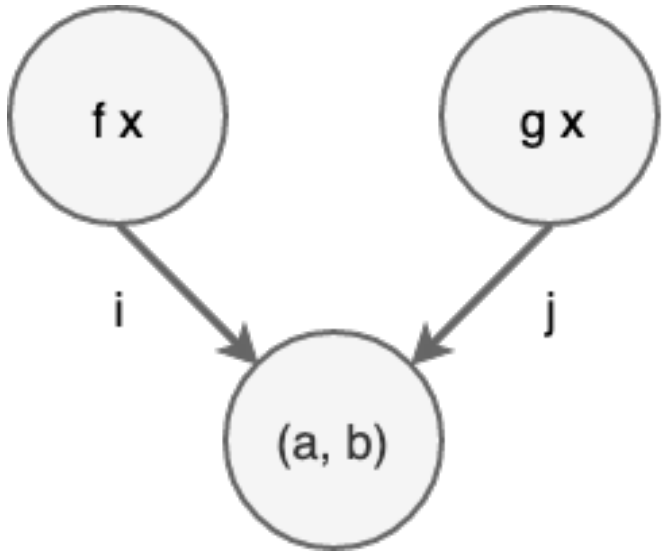
Evaluation strategies: `Control.Parallel.Strategies`

- A Strategy is a higher level abstraction built upon Haskell's atoms for parallelism, *sparks*.
- It is essentially a description of how components of a data structure can be evaluated in parallel.
- Deterministic: the program will always return the same result (regardless of number of processors)
- Modular: decouples an algorithm from the method used to parallelise it

Dataflow parallelism: `Control.Monad.Par`

The idea behind the Par monad is that the programmer can specify the program in terms of a data flow model.

```
runPar $ do
  i <- new
  j <- new
  fork (put i (f 10))
  fork (put j (g 10))
  a <- get i
  b <- get j
  return (a,b)
```



Example code

```
-- sequential
derivatives :: RE -> [Symbol] -> [RE]
derivatives re abc = map (derive re) abc

-- evaluation strategies
derivatives :: RE -> [Symbol] -> [RE]
derivatives re abc = map (derive re) abc
                         `using` parList rdeepseq

-- the Par monad
derivatives :: RE -> [Symbol] -> Par [RE]
derivatives re abc = parMapM (return . derive re) abc
```

Experimental setup

The basic approach follows that employed by Strauss et al. in the hope of allowing for meaningful comparisons to be drawn between the results obtained.

- generate two sets of regular expressions (with different depth syntax trees), with $|\Sigma| = 4$ and $|\Sigma| = 94$ respectively.
- compare running times for the different implementations: sequential, strategies (with and without "chunking") and the Par monad.
- speedup and efficiency compared to number of cores utilised are considered as metrics of overall performance

All code was compiled using GHC version 8.6.5 and run on a 2015 iMac with a 3.1GHz 4 core Intel i5 processor and 8GB of RAM.

Results: Speedup, $|\Sigma| = 4$

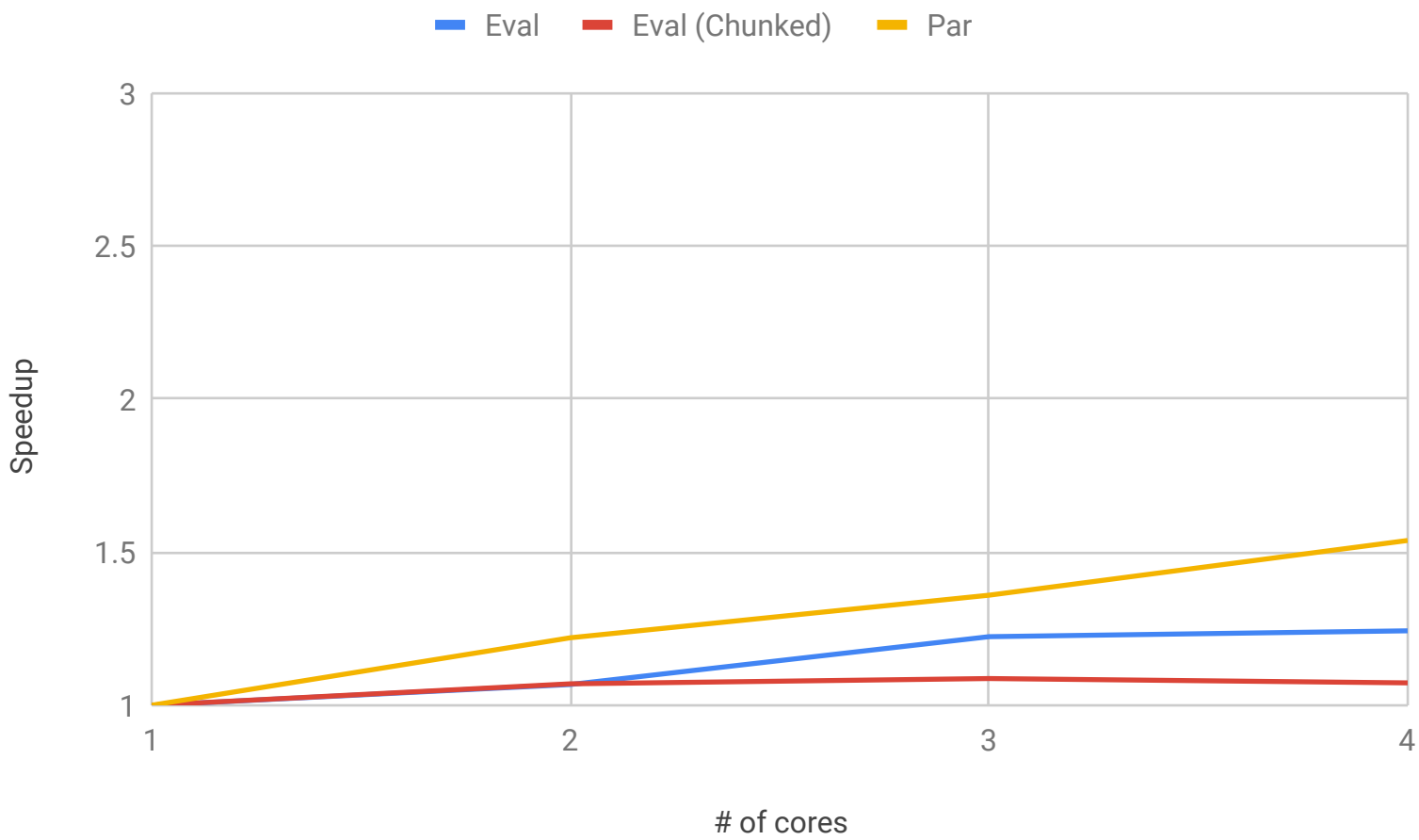


Figure: Speedup versus number of cores, $|\Sigma| = 4$

Results: Efficiency, $|\Sigma| = 4$

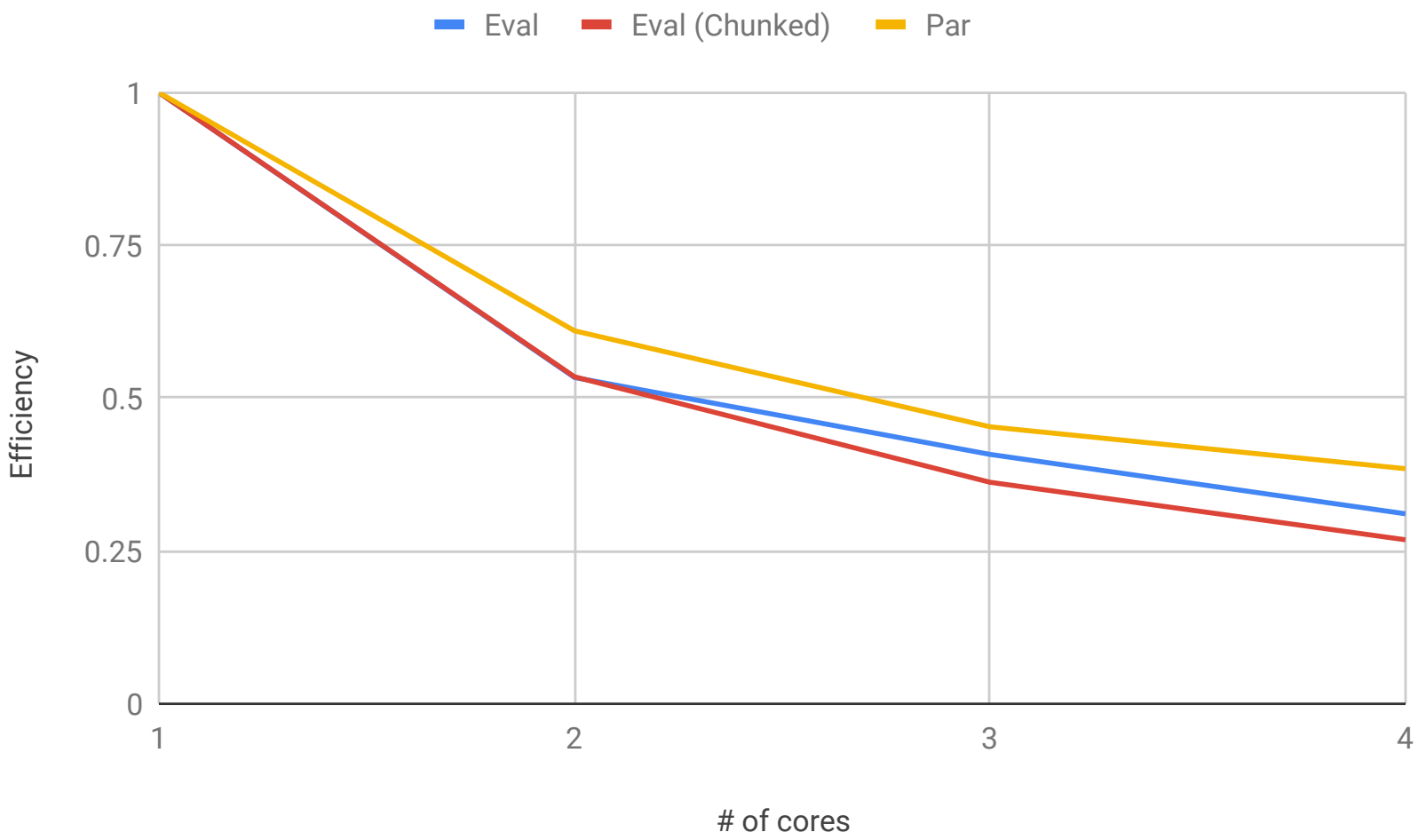


Figure: Efficiency versus number of cores, $|\Sigma| = 4$

Results: Speedup, $|\Sigma| = 94$

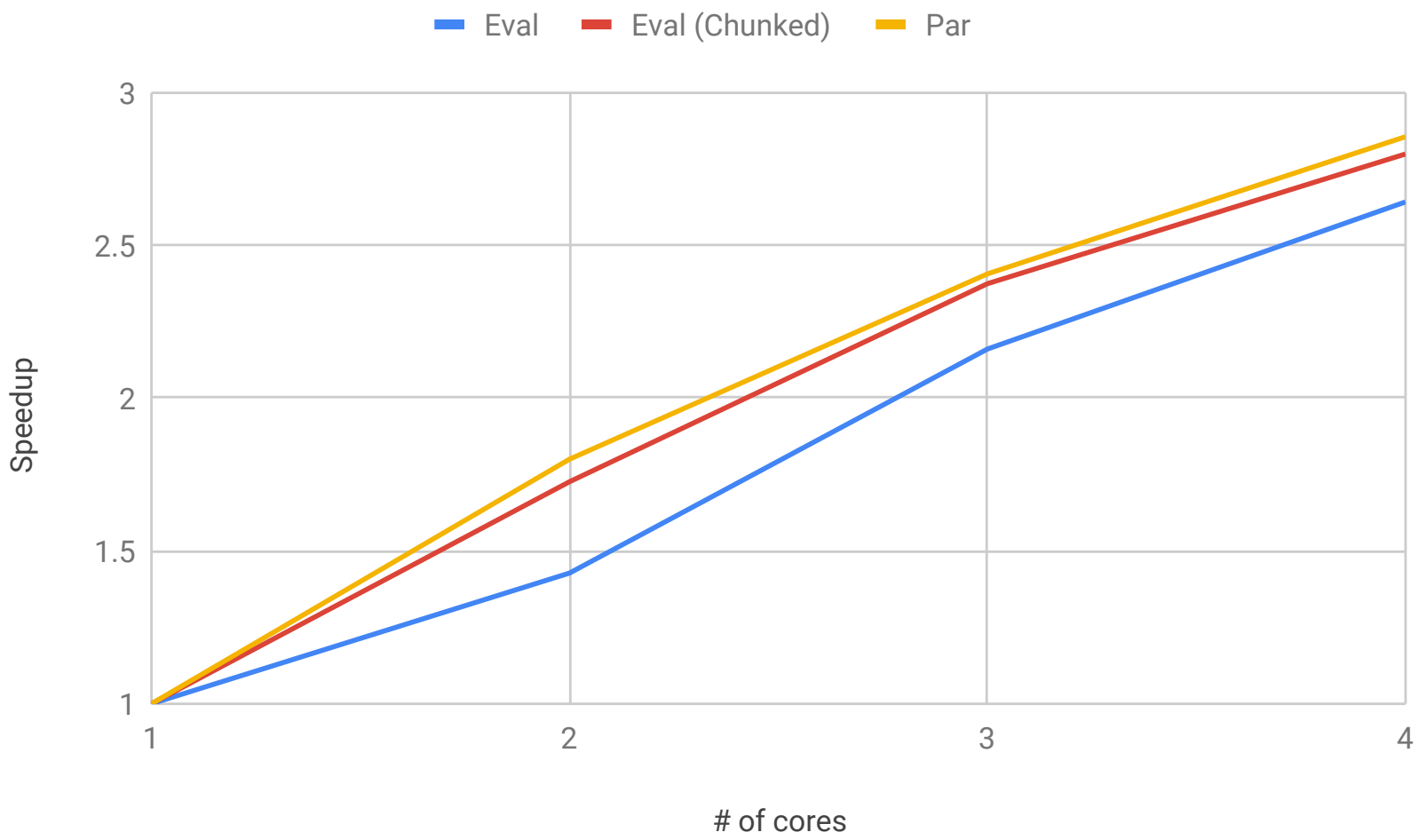


Figure: Speedup versus number of cores, $|\Sigma| = 94$

Results: Efficiency, $|\Sigma| = 94$

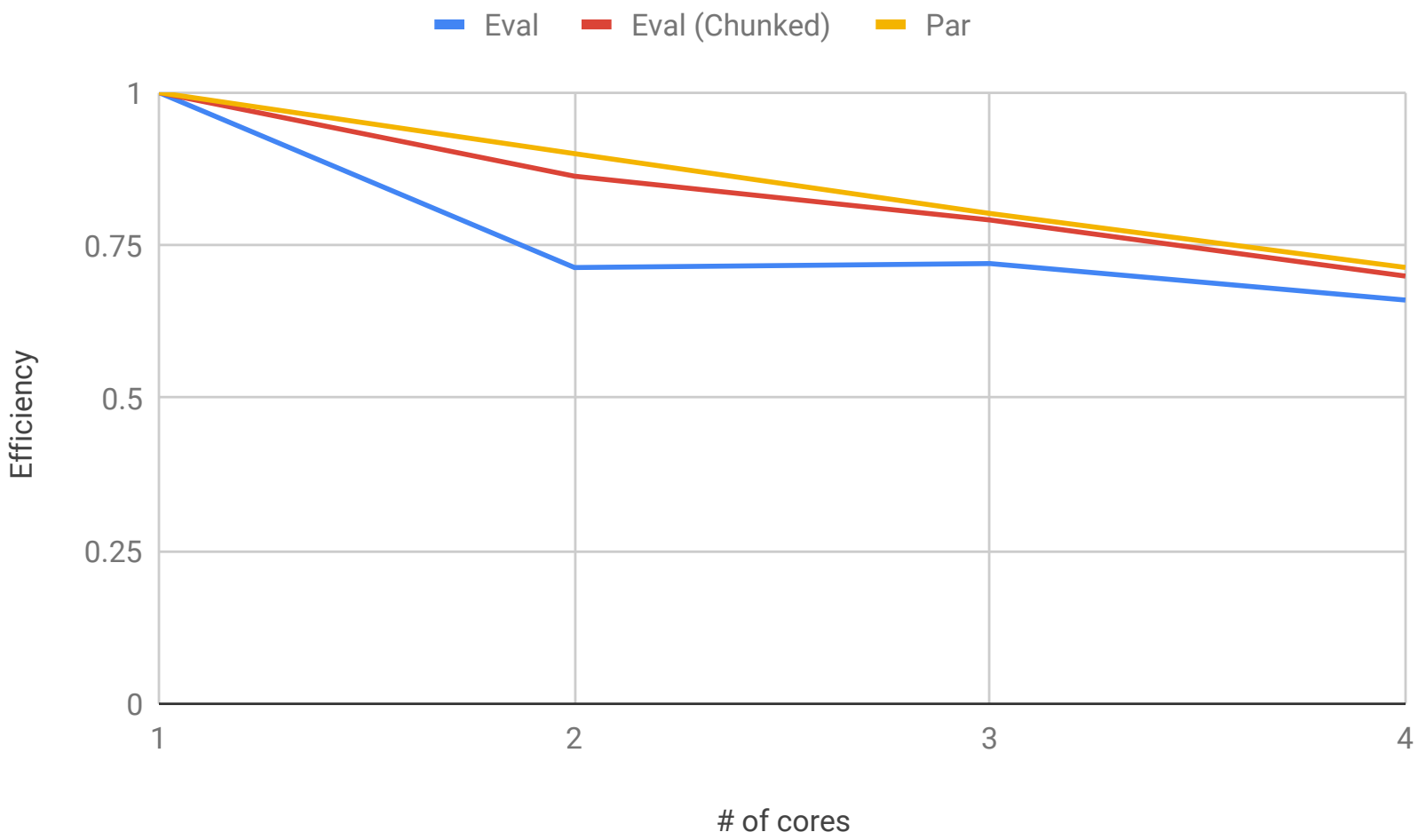


Figure: Efficiency versus number of cores, $|\Sigma| = 94$

Conclusions

- The Par monad approach outperformed the Evaluation Strategies approach for both the small and larger alphabet.
- Generally the boost in speedup to be gained and overall efficiency diminished with an increased number of cores.
- More significant performance gains were achieved when the alphabet was larger.
  - interestingly, this reverses the findings reported by Strauss et al. (they achieved a median speedup of 1.72 for the small alphabet, yet only 1.09 for the large alphabet case)