Parallelization of an Ubiquitous Sequential Computation

Franz A. Heinsen
franz@glassroom.com

Abstract

We show how to compute the elements of a sequence $x_t = a_t x_{t-1} + b_t$ in parallel, given $t = (1, 2, \ldots, n)$, $a_t \in \mathbb{R}^n$, $b_t \in \mathbb{R}^n$, and initial value $x_0 \in \mathbb{R}$. On n parallel processors, the computation of n elements incurs $\mathcal{O}(\log n)$ time and $\mathcal{O}(n)$ space. Sequences of this form are ubiquitous in science and engineering, making their parallelization useful for a vast number of applications. We implement parallelization in software, test it on parallel hardware, and verify that it executes faster than sequential computation by a factor of $\frac{n}{\log n}$.[1]

1 Summary

Sequences of the form $x_t = a_t x_{t-1} + b_t$ are ubiquitous in science and engineering. For example, in the natural sciences, such sequences can model quantities or populations that decay or grow by a varying rate $a_t > 0$ between net inflows or outflows b_t at each time step t. In economics, such sequences can model investments that earn a different rate of return $a_t = (1 + r_t)$ between net deposits or withdrawals b_t over each time period t. In engineering applications, such sequences are often low-level components of larger models, *e.g.*, linearized recurrent neural networks whose layers decay token features in a sequence of tokens.

Given a finite sequence $x_t = a_t x_{t-1} + b_t$ with n steps, $t = (1, 2, \ldots, n)$, where $a_t \in \mathbb{R}^n$, $b_t \in \mathbb{R}^n$, and initial value $x_0 \in \mathbb{R}$, it's not immediately obvious how one would compute all elements in parallel, because each element is a non-associative transformation of the previous one. *In practice, we routinely see software code that computes sequences of this form one element at a time.*

[1] Source code for replicating our results is available online at https://github.com/glassroom/heinsen_sequence.

The vector $\log x_t$ is computable as a composition of two cumulative, or prefix, sums, each of which *is* parallelizable:

$$\log x_t = a^*_t + \log\left(x_0 + b^*_t\right) \tag{1}$$

where a^*_t and b^*_t are the two prefix sums:

$$\begin{aligned} a^*_t &= \overset{\text{cum}}{\sum_t} \log a_t \\ b^*_t &= \overset{\text{cum}}{\sum_t} e^{\log b_t - a^*_t}. \end{aligned} \tag{2}$$

The operator $\overset{\text{cum}}{\sum}$ computes a *vector* whose elements are a prefix sum, *i.e.*, a cumulative sum.

We obtain x_t with elementwise exponentiation:

$$x_t = e^{a^*_t + \log(x_0 + b^*_t)}. \tag{3}$$

Prefix sums are associative,[2] making it possible to compute them by parts in parallel. Well-known parallel algorithms for efficiently computing the prefix sum of a sequence with n elements incur $\mathcal{O}(\log n)$ time and $\mathcal{O}(n)$ space on n parallel processors (Ladner and Fischer, 1980) (Hillis and Steele, 1986). Prefix sums generalize to any binary operation that is associative, making them a useful primitive in data-parallel models of computation (Blelloch, 1990). Many software frameworks for numerical computing provide efficient parallel implementations of the prefix sum.

The computation of two prefix sums has the same computational complexity on n parallel processors as a single prefix sum: $\mathcal{O}(\log n)$ time and $\mathcal{O}(n)$ space. The computation of n elementwise

[2] Given a sequence a, b, c,

$$\overset{\text{cum}}{\sum}\Big(a, \overset{\text{cum}}{\sum}\Big(b, c\Big)\Big) = \overset{\text{cum}}{\sum}\Big(\overset{\text{cum}}{\sum}\Big(a, b\Big), c\Big).$$

operations (*e.g.*, logarithms and exponentials) on n parallel processors incurs constant time and, if the computation is *in situ*, no additional space.

If any $a_t < 0$, any $b_t < 0$, or $x_0 < 0$, one or more of the logarithms computed in the interim will be in $\mathbb{C}$, but all elements of x_t will always be in $\mathbb{R}$, because they are defined as multiplications and additions of previous elements in $\mathbb{R}$, which is closed under both operations.

2 Proof

We are computing $x_t = a_t x_{t-1} + b_t$, for $t = (1, 2, \ldots, n)$, with $a_t \in \mathbb{R}^n$, $b_t \in \mathbb{R}^n$, and initial value $x_0 \in \mathbb{R}$. Expand the expression that computes each element, $x_1, x_2, \ldots, x_n$, to make it a function of x_0 and all trailing elements of a_t and b_t, and factor out all trailing coefficients:

$$\begin{aligned}
x_1 &= a_1 x_0 + b_1 \\
&= a_1 \left(x_0 + \frac{b_1}{a_1} \right) \\
x_2 &= a_2 x_1 + b_2 \\
&= a_1 a_2 \left(x_0 + \frac{b_1}{a_1} + \frac{b_2}{a_1 a_2} \right) \\
&\vdots \\
x_n &= a_n x_{n-1} + b_n \\
&= \left(\prod_t a_t \right) \left(x_0 + \frac{b_1}{a_1} + \frac{b_2}{a_1 a_2} + \cdots + \frac{b_n}{\prod_t a_t} \right).
\end{aligned} \tag{4}$$

Combine all expressions in (4) into one expression that computes all elements of vector x_t:

$$\begin{aligned}
x_t &= \left(\prod_t^{\text{cum}} a_t \right) \odot \left(x_0 + \sum_t^{\text{cum}} \frac{b_t}{\prod_t^{\text{cum}} a_t} \right) \\
&= \left(\prod_t^{\text{cum}} a_t \right) \odot \left(x_0 + \sum_t^{\text{cum}} \exp\left(\log \frac{b_t}{\prod_t^{\text{cum}} a_t} \right) \right) \\
&= \left(\prod_t^{\text{cum}} a_t \right) \odot \left(x_0 + \sum_t^{\text{cum}} e^{\log b_t - \sum_t^{\text{cum}} \log a_t} \right),
\end{aligned} \tag{5}$$

where the operators $\prod^{\text{cum}}$ and $\sum^{\text{cum}}$ compute *vectors* whose elements are, respectively, a cumulative product and sum, and $\odot$ denotes an element-wise or Hadamard product.

Taking the logarithm on both sides, we obtain:

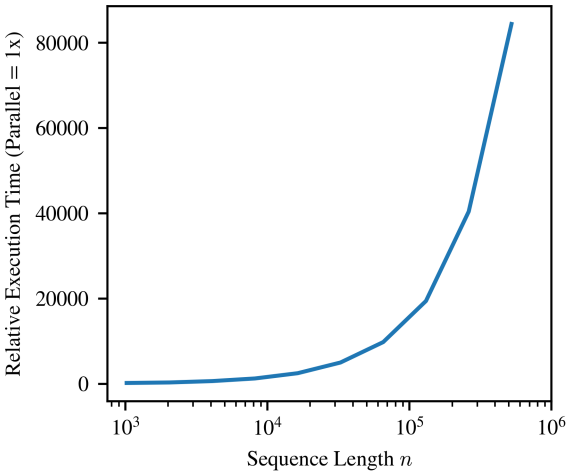

Figure 1: Time to compute n elements sequentially, relative to parallel computation, on an Nvidia GPU. Each point is the mean of 30 runs.

$$\log x_t = \underbrace{\sum_t^{\text{cum}} \log a_t}_{a_t^*} + \log \left(x_0 + \underbrace{\sum_t^{\text{cum}} e^{\log b_t - \overbrace{\sum_t^{\text{cum}} \log a_t}^{a_t^*}}}_{b_t^*} \right), \tag{6}$$

which is the same as (1).

3 Implementation

We implement (3) in software. For numerical stability and slightly improved efficiency, we modify the computation of x_t as follows:

$$x_t = e^{a_t^* - \text{tail}(\text{LCSE}(\text{cat}(\log x_0, \log b_t - a_t^*)))}, \tag{7}$$

where $\text{cat}(\cdot)$ denotes concatenation, $\text{LCSE}(\cdot)$ is the LogCumSumExp function (short for "logarithm of a cumulative sum of exponentials"), applying the familiar log-sum-exp trick as necessary, and $\text{tail}(\cdot)$ removes its argument's first element.

We test our implementation on parallel hardware and verify that it executes faster than sequential computation by a factor of $\frac{n}{\log n}$ (Figure 1).

References

Guy E. Blelloch. 1990. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA.

W. Daniel Hillis and Guy L. Steele. 1986. Data parallel algorithms. *Commun. ACM* 29(12):1170–1183.

Richard E. Ladner and Michael J. Fischer. 1980. Parallel prefix computation. *J. ACM* 27(4):831–838.