# Introduction to Cost Semantics for Parallel Languages

Adrien Guatto

MPRI 2.37.1
2019–2020

**Abstract**

In this short note, we present *cost semantics* for parallel $\lambda$-calculi. Such semantics specify not only *what* programs compute, but also *at what cost*. We first apply this approach to a pure call-by-value $\lambda$-calculus. Following Blelloch and Greiner [1995], we sketch a provably-efficient implementation of this calculus on top of a Parallel Random Access Machine. Then, we extend the calculus with operations acting upon some abstract *state* living in an arbitrary partial commutative monoid. We then instantiate this construction to obtain an idealized version of Cilk [Frigo et al., 1998].

## 1 Introduction

During the previous lectures, we studied the performance of Cilk programs in an analytical fashion, in terms of *work* and *span*. Rather than resorting to some low-level machine model, we reasoned informally at the program level. This reasoning, however, was informal.

The goal of this lecture is to show how this approach can be made rigorous using techniques inspired from programming languages semantics.

**Conventions.** By *list* we mean a finite sequence, possibly empty. We write $(X^*, \cdot, \epsilon)$ for the free monoid over $X$, that is, for the set of lists of elements of $X$, equipped with the concatenation operator $x \cdot y$, as well as the empty list $\epsilon$.

We write $X \to_{\mathsf{fin}} Y$ for the set of finite partial maps from $X$ to $Y$. Given such a finite partial map $f$, we write $f[x \mapsto y]$ for the finite partial map sending $x'$ to $y$ when $x = x'$ or to $f(x')$ when $x' \neq x$ and $f(x')$ is defined. We write $\mathsf{dom}(f)$ for the subset of $X$ for which $f$ is defined. Thus, $\mathsf{dom}(f[x \mapsto y]) = \mathsf{dom}(f) \cup \{x\}$. We write $\emptyset$ for an empty finite partial map.

We write $\mathcal{M}_{\mathsf{fin}}(X)$ for the set of finite multisets of elements of $X$. We write $[x_1, \ldots, x_n]$ for the multiset of size $n$ whose elements are $x_1, \ldots, x_n$. By definition, we may have $x_i = x_j$. Given two multisets $A, B \in \mathcal{M}_{\mathsf{fin}}(X)$, we denote $A + B$ their multiset union.

We assume given some disjoint countably infinite sets *Var* and *Loc*. We call *variables* the elements $x, y, z \ldots$ of *Var* and *locations* the elements $\ell, \ell_1, \ell_2 \ldots$ of *Loc*.

All the syntactic objects we consider are always identified up to $\alpha$-equivalence. Furthermore, we follow Barendregt's variable convention: in a given metatheoretical statement, no variable is both free and bound and all bound variables are distinct.

$$
\begin{array}{ll}
\text{Terms} & M, N, S ::= x \mid \lambda x.M \mid M\ N \mid c \\
\text{Constants} & c ::= i \mid \mathtt{add} \mid \mathtt{add}_i \mid \mathtt{sub} \mid \mathtt{sub}_i \mid \ldots \\
\text{Environments} & \gamma, \sigma \in \mathit{Var} \to_{\mathsf{fin}} \mathit{Val} \\
\text{Values} & V, W ::= c \mid (x.M)\{\gamma\} \mid \mathsf{nil}
\end{array}
$$

Figure 1: Syntax of PAL

$$\boxed{M; \gamma \Downarrow V}$$

$$\textsc{Const}\ \frac{}{c; \gamma \Downarrow c} \qquad \textsc{Var}\ \frac{}{x; \gamma \Downarrow \gamma(x)} \qquad \textsc{Fun}\ \frac{}{\lambda x.M; \gamma \Downarrow (x.M)\{\gamma\}}$$

$$\textsc{AppClo}\ \frac{M; \gamma \Downarrow (x.S)\{\sigma\} \qquad N; \gamma \Downarrow V \qquad S; \sigma[x \mapsto V] \Downarrow W}{M\ N; \gamma \Downarrow W} \qquad \textsc{AppConst}\ \frac{M; \gamma \Downarrow c \qquad N; \gamma \Downarrow V}{M\ N; \gamma \Downarrow \delta_c(V)}$$

Figure 2: Plain Semantics of PAL

# 2 A Stateless Calculus

## 2.1 Syntax and Plain Semantics

Our initial goal is to design a calculus for which a cost semantics is both interesting and useful. Such a calculus is subject to the following requirements.

1. It should be as simple as possible to make formal study doable.
2. It should form a reasonable basis for a full-fledged programming language.
3. It has to be implementable efficiently atop a parallel machine model.

The first two requirements steer us towards a variant of $\lambda$-calculus. However, to make $\lambda$-calculus into an idealized programming language, one should tame unrestricted $\beta$-reduction by fixing an evaluation strategy. The third requirement discourages the use of call-by-name and call-by-need, which have rather complicated cost models, especially in a parallel setting. Less importantly, it also suggests adding dedicated arithmetic facilities to our calculus.

For these reasons, we follow Blelloch and Greiner [1995] and study the Parallel Applicative Calculus (PAL), an *applicative* (call-by-value) calculus parameterized over a family of constants. Its syntax is given in Figure 1. The definition of terms is unremarkable. Constants $c$ should include at least integer literals $i$ and some arithmetic operations such as $\mathtt{add}$ and $\mathtt{add}_i$. We will explain the latter shortly. This grammar also includes the objects that, while not part of the syntax proper, are needed by our semantics. Values $V, W$ are the results of execution, and can be either a constant $c$ or a *closure* $(x.M)\{\gamma\}$ bundling a $\lambda$-abstraction with an environment $\gamma$ holding the values of the free variables of $M$. The special value $\mathsf{nil}$ plays a technical role in a later section, and can be ignored for now. An environment is a finite map from variables to values.

The cost semantics we will adopt for PAL constitutes an extension of the classic big-step formulation of call-by-value with environments and closures, so it is worth remembering its

definition. It takes the form of an *evaluation* judgment

$$M; \gamma \Downarrow V$$

stating that $M$ evaluates to $V$ in the environment $\gamma$. The rules of this judgment are given in Figure 2. A constant evaluates to itself. A variable $x$ evaluates to the value assigned to it in $\gamma$. Its evaluation is not defined if $x \notin \mathsf{dom}(\gamma)$. A $\lambda$-abstraction evaluates by closing over the environment. Finally, there are two rules for applications $M\,N$ depending on whether the function $M$ evaluates to a closure or to a constant. When $M$ evaluates to a closure $(x.S)\{\sigma\}$, the final result is that of the evaluatation of $S$ in $\sigma$ extended by binding the formal parameter $x$ to the value $W$ of $N$. When $M$ evaluates to a constant $c$, the final result is determined according to a family of partial map $(\delta_c : \mathit{Val} \rightharpoonup \mathit{Val})_{c \in \mathit{Const}}$. This map is a parameter of PAL, as is the language of constants. As an example, it could contain the following clauses.

$$\delta_{\mathtt{add}}(i) = \mathtt{add}_i \qquad \delta_{\mathtt{add}_i}(j) = i + j \qquad \delta_{\mathtt{sub}}(i) = \mathtt{sub}_i \qquad \delta_{\mathtt{sub}_i}(j) = i - j \qquad \ldots$$

This illustrates the purely technical role played by constants such as $\mathtt{add}_i$ and $\mathtt{sub}_i$, which serve as intermediate steps when evaluating `add` and `sub`. An alternative would be to enrich PAL with pairs and formulate `add` and `sub` as uncurried operations.

## 2.2 Cost Semantics

For our purpose, a cost semantics for PAL should give enough information to analyze its performance on a parallel machine. To do so, we will associate to every evaluation a *cost graph* build according to the following grammar.

$$\text{Cost graph} \qquad g ::= \mathbf{0} \mid \mathbf{1} \mid g_1 \oplus g_2 \mid g_1 \otimes g_2$$

Here, $\mathbf{0}$ is the graph with no vertex, $\mathbf{1}$ the graph with a single vertex and no edge, $g_1 \oplus g_2$ the series composition of $g_1$ and $g_2$ and $g_1 \otimes g_2$ their parallel composition. We write $\mathbf{n}$ for $\mathbf{1}$ composed sequentially with itself $n$ times. The work $\mathsf{work}(g)$ and span $\mathsf{span}(g)$ of a cost graph $g$ are defined as expected.

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{work}(\mathbf{0}) & = & 0 & \mathsf{span}(\mathbf{0}) & = & 0 \\
\mathsf{work}(\mathbf{1}) & = & 1 & \mathsf{span}(\mathbf{1}) & = & 1 \\
\mathsf{work}(g_1 \oplus g_2) & = & \mathsf{work}(g_1) + \mathsf{work}(g_2) & \mathsf{span}(g_1 \oplus g_2) & = & \mathsf{span}(g_1) + \mathsf{span}(g_2) \\
\mathsf{work}(g_1 \otimes g_2) & = & \mathsf{work}(g_1) + \mathsf{work}(g_2) & \mathsf{span}(g_1 \otimes g_2) & = & \max(\mathsf{span}(g_1), \mathsf{span}(g_2))
\end{array}$$

Now, we can restate our big-step judgment as $M; \sigma \Downarrow V; g$, where $g$ describes the cost of evaluating $M$. Its rules are given in Figure 3. They compute the same result $V$ as those of Figure 2. Since we are mostly interested in asymptotics, elementary operations are assumed to have unit costs (rules Const, Var, and Fun). Rule AppClo is more interesting: it specifies that the function and its argument are evaluated in parallel. The overhead of the rule itself is $\mathbf{2}$. This choice of constant is needed to obtain a tight correspondance with the machine model described in the next section — one may replace it with $\mathbf{1}$ without affecting asymptotic complexity results. Rule AppConst is similar but simpler.

$$\boxed{M;\gamma \Downarrow V;g}$$

$$\textsc{Const}\ \frac{}{c;\gamma \Downarrow c;\mathbf{1}} \qquad \textsc{Var}\ \frac{}{x;\gamma \Downarrow \gamma(x);\mathbf{1}} \qquad \textsc{Fun}\ \frac{}{\lambda x.M;\gamma \Downarrow (x.M)\{\gamma\};\mathbf{1}}$$

$$\textsc{AppClo}\ \frac{M;\gamma \Downarrow (x.S)\{\sigma\};g_1 \qquad N;\gamma \Downarrow V;g_2 \qquad S;\sigma[x \mapsto V] \Downarrow W;g_3}{M\,N;\gamma \Downarrow W;(g_1 \otimes g_2) \oplus g_3 \oplus \mathbf{2}}$$

$$\textsc{AppConst}\ \frac{M;\gamma \Downarrow c;g_1 \qquad N;\gamma \Downarrow V;g_2}{M\,N;\gamma \Downarrow \delta_c(V);(g_1 \otimes g_2) \oplus \mathbf{2}}$$

Figure 3: Cost Semantics of PAL

## 2.3 Towards A Provable Implementation

We have defined a cost semantics for PAL, but how do we know that this semantics is sensible, in the sense that it can be used to reason about real-world performance, even when one only cares about asymptotics? After all, mathematics does not prevent us from associating evaluations with arbitrary cost graphs.

The solution is, as in the sequential case, to implement PAL on top of some widely-accepted, concrete parallel machine model, and then show that this implementation is sound with respect to our cost semantics. In other words, it should compute the same values as the cost semantics, and with the same work and span (up to constant factors).

We follow the exact same route as Blelloch and Greiner [1995], using an abstract machine as an intermediate point between PAL and the concrete machine model. We then implement this abstract machine onto the concrete machine. The only difference with their work is that we use a variant of the Krivine/CK machine rather than the SECD machine.

### 2.3.1 A Parallel Abstract Machine

The abstract machine schedules the execution of PAL programs onto several parallel *threads*. It also uses a global store to hold thread results. The grammar below describes its components.

| | |
|---|---|
| Configuration | $C ::= T \,/\, R$ |
| Thread pools | $T \in \mathcal{M}_{\mathsf{fin}}(Thr)$ |
| Threads | $t ::= \langle V \mid - \mid K\rangle \mid \langle M \mid \sigma \mid K\rangle$ |
| Stack | $K ::= \Box^{\ell}_{\mathbf{t}} \mid \Box^{\ell}_{\mathbf{a}} :: K \mid \Box^{\ell}_{\mathbf{f}} :: K \mid (V\,\Box) :: K$ |
| Results | $R \in Loc \rightarrow_{\mathsf{fin}} Val$ |

The global store $R$ is a finite map from locations to values. A thread pool $T$ is a bag of threads. Each thread $t$ is a call-by-value Krivine machine, which pairs a stack $K$ with either a value $V$ or a closure $(M|\sigma)$. The empty stack $\Box^{\ell}_{\mathbf{t}}$ designates a final result to be stored in $R(\ell)$. The stack $(V\,\Box) :: K$ appears as is in the usual CK machine — it represents the evaluation context $K[V\,\Box]$. The contexts $\Box^{\ell}_{\mathbf{a}} :: K$ and $\Box^{\ell}_{\mathbf{f}} :: K$ are used for synchronization, as will be made evident by machine transitions.

$$\langle c \,|\, \sigma \,|\, K\rangle \rightsquigarrow \langle c \,|\, - \,|\, K\rangle \qquad \textsc{SConst}$$

$$\langle x \,|\, \sigma \,|\, K\rangle \rightsquigarrow \langle \sigma(x) \,|\, - \,|\, K\rangle \qquad \textsc{SVar}$$

$$\langle \lambda x.M \,|\, \sigma \,|\, K\rangle \rightsquigarrow \langle (x.\sigma)\{M\} \,|\, - \,|\, K\rangle \qquad \textsc{SFun}$$

$$\langle V \,|\, - \,|\, ((x.M)\{\sigma\}\,\square) :: K\rangle \rightsquigarrow \langle M \,|\, \sigma[x \mapsto V] \,|\, K\rangle \qquad \textsc{SAppClo}$$

$$\langle V \,|\, - \,|\, (c\,\square) :: K\rangle \rightsquigarrow \langle \delta(c, V) \,|\, - \,|\, K\rangle \qquad \textsc{SAppConst}$$

$$\textsc{EStep}\ \frac{t \rightsquigarrow t'}{[t] \,/\, R \Rightarrow_{\mathbf{e}} [t'] \,/\, R} \qquad \textsc{ESpawn}\ \frac{\ell \notin \mathsf{dom}(R)}{[\langle M\,N \,|\, \sigma \,|\, K\rangle] \,/\, R \Rightarrow_{\mathbf{e}} [\langle M \,|\, \sigma \,|\, \square^{\ell}_{\mathbf{a}} :: K\rangle, \langle N \,|\, \sigma \,|\, \square^{\ell}_{\mathbf{f}} :: K\rangle] \,/\, R[\ell \hookrightarrow \mathsf{nil}]}$$

$$\textsc{EIdleA}\ \frac{}{[\langle V \,|\, - \,|\, \square^{\ell}_{\mathbf{a}} :: K\rangle] \,/\, R \Rightarrow_{\mathbf{e}} [\langle V \,|\, - \,|\, \square^{\ell}_{\mathbf{a}} :: K\rangle] \,/\, R} \qquad \textsc{EIdleF}\ \frac{}{[\langle V \,|\, - \,|\, \square^{\ell}_{\mathbf{f}} :: K\rangle] \,/\, R \Rightarrow_{\mathbf{e}} [\langle V \,|\, - \,|\, \square^{\ell}_{\mathbf{f}} :: K\rangle] \,/\, R}$$

$$\textsc{EFinish}\ \frac{}{[\langle V \,|\, - \,|\, \square^{\ell}_{\mathbf{t}}\rangle] \,/\, R \Rightarrow_{\mathbf{e}} [] \,/\, R[\ell \hookrightarrow V]} \qquad \textsc{EPar}\ \frac{T_1 \,/\, R \Rightarrow_{\mathbf{e}} T_1' \,/\, R' \qquad T_2 \,/\, R' \Rightarrow_{\mathbf{e}} T_2' \,/\, R''}{T_1 + T_2 \,/\, R \Rightarrow_{\mathbf{e}} T_1' + T_2 \,/\, R''}$$

$$\textsc{FStore}\ \frac{R(\ell) = \mathsf{nil}}{[\langle V \,|\, - \,|\, \square^{\ell}_{\mathbf{f}} :: K\rangle] \,/\, R \Rightarrow_{\mathbf{f}} [] \,/\, R[\ell \hookrightarrow V]} \qquad \textsc{FSync}\ \frac{R(\ell) \neq \mathsf{nil}}{[\langle V \,|\, - \,|\, \square^{\ell}_{\mathbf{f}} :: K\rangle] \,/\, R \Rightarrow_{\mathbf{f}} [\langle R(\ell) \,|\, - \,|\, (V\,\square) :: K\rangle] \,/\, R}$$

$$\textsc{FIdle}\ \frac{t \neq \langle - \,|\, - \,|\, \square^{-}_{\mathbf{f}} :: -\rangle}{[t] \,/\, R \Rightarrow_{\mathbf{a}} [t] \,/\, R} \qquad \textsc{FPar}\ \frac{T_1 \,/\, R \Rightarrow_{\mathbf{f}} T_1' \,/\, R' \qquad T_2 \,/\, R' \Rightarrow_{\mathbf{f}} T_2' \,/\, R''}{T_1 + T_2 \,/\, R \Rightarrow_{\mathbf{f}} T_1' + T_2 \,/\, R''}$$

$$\textsc{AStore}\ \frac{R(\ell) = \mathsf{nil}}{[\langle V \,|\, - \,|\, \square^{\ell}_{\mathbf{a}} :: K\rangle] \,/\, R \Rightarrow_{\mathbf{a}} [] \,/\, R[\ell \hookrightarrow V]} \qquad \textsc{ASync}\ \frac{R(\ell) \neq \mathsf{nil}}{[\langle V \,|\, - \,|\, \square^{\ell}_{\mathbf{a}} :: K\rangle] \,/\, R \Rightarrow_{\mathbf{a}} [\langle V \,|\, - \,|\, (R(\ell)\,\square) :: K\rangle] \,/\, R}$$

$$\textsc{AIdle}\ \frac{t \neq \langle - \,|\, - \,|\, \square^{-}_{\mathbf{a}} :: -\rangle}{[t] \,/\, R \Rightarrow_{\mathbf{a}} [t] \,/\, R} \qquad \textsc{APar}\ \frac{T_1 \,/\, R \Rightarrow_{\mathbf{a}} T_1' \,/\, R' \qquad T_2 \,/\, R' \Rightarrow_{\mathbf{a}} T_2' \,/\, R''}{T_1 + T_2 \,/\, R \Rightarrow_{\mathbf{a}} T_1' + T_2 \,/\, R''}$$

$$\textsc{Step}\ \frac{T \,/\, R \Rightarrow_{\mathbf{e}} T' \,/\, R' \qquad T' \,/\, R' \Rightarrow_{\mathbf{f}} T'' \,/\, R'' \qquad T'' \,/\, R' \Rightarrow_{\mathbf{a}} T''' \,/\, R'''}{T \,/\, R \Rightarrow T''' \,/\, R'''}$$

Figure 4: Parallel Abstract Machine for PAL

The transitions between threads and configurations are given in Figure 4. The relation $\rightsquigarrow$ on threads is the usual one for the call-by-value Krivine machine, with most transitions related to applications missing. The main relation, $\Rightarrow$, operates on configurations. Its execution divides into three phases modeled by the relations $\Rightarrow_{\mathbf{e}}$, $\Rightarrow_{\mathbf{a}}$, and $\Rightarrow_{\mathbf{f}}$, as per rule Step. All three perform exactly one transition per thread in the current thread pool — this form of parallelism arises from rules EPar, APar, and FPar.

1. The *evaluation* relation $\Rightarrow_{\mathbf{e}}$ performs either a sequential step $\rightsquigarrow$ (EStep), or computes $M\,N$ by spawning two new threads to compute $M$ and $N$ in parallel (ESpawn). In this case, we allocate a fresh location $\ell$ to store the result of the first thread to terminate. This location is initialized to $\mathsf{nil}$, so that a location is only allocated once.

2. The main rules of the *function synchronization* relation $\Rightarrow_{\mathbf{a}}$ are FStore and FSync. They deal with threads of the form $\langle V \mid - \mid \Box^{\ell}_{\mathbf{a}} :: K\rangle$, with the rule FIdle ensuring that all other threads are left unchanged. Such a thread has finished computing the value $V$ of the function term in some application that was spawned previously. What happens depends on whether the evaluation of the corresponding argument term has finished during the previous $\Rightarrow_{\mathbf{a}}$-phase. If it is the case, then $R(\ell)$ contains its result. Then, rule FSync proceeds by, so to speak, joining the two threads into $\langle R(\ell) \mid - \mid (V\,\Box) :: K\rangle$, which will resume evaluating at the next $\Rightarrow_{\mathbf{e}}$-phase. Otherwise, $R(\ell)$ is $\mathsf{nil}$, and rule FStore replaces it with $V$. This binding will be used once the argument term terminates during a subsequent $\Rightarrow_{\mathbf{a}}$-phase.

3. The *argument synchronization* relation $\Rightarrow_{\mathbf{a}}$ is similar to the previous one, except that it deals with the argument terms spawned by applications, rather than the function ones.

To summarize, this semantics works by alternating phases that make sequential progress, spawning new threads as needed, and phases that check whether each spawned thread has finished running its subcomputation, in which case its result should be either stored or give rise to a new continuation thread.

As stated, our semantics is not deterministic, since new locations get allocated into the store in a nondeterministic manner by rule ESpawn. However, this is the sole source of nondeterminism: since the semantics is parallel and greedy, every thread terminates as early as possible. In the following, we reason up to the renaming of locations by a bijection. This can of course be made formal, at the cost of heavier theorem statements.

In the remainder of this subsection, we prove that the parallel abstract machine is a *provably-efficient* implementation of the cost semantics. In other words, it computes correct results, and its performance matches the work and span specified by our big-step judgment. To do so, we need a number of technical results, such as the fact that the store $R$ grows *monotonically*.

**Definition 1** (Store Extension)**.** $R'$ is an *extension* of $R$, denoted $R \sqsubseteq R'$, when $\mathsf{dom}(R) \subseteq \mathsf{dom}(R')$ and for all $\ell \in \mathsf{dom}(R)$, $R(\ell) \neq \mathsf{nil}$ implies $R'(\ell) = R(\ell)$.

**Proposition 1** (Store Monotonicity)**.** *If* $T \,/\, R \Rightarrow T' \,/\, R'$ *then* $R \sqsubseteq R'$.

We can now state our main result.

**Theorem 1.** *If* $M; \emptyset \Downarrow V; g$ *then there is* $R$ *such that* $[\langle M \mid \emptyset \mid \Box^{\ell}_{\mathbf{t}}\rangle] \,/\, \emptyset[\ell \hookrightarrow \mathsf{nil}] \Rightarrow^{\mathsf{span}(g)} [] \,/\, R$ *with* $R(\ell) = V$*. Furthermore, this reduction sequence creates* $\mathsf{work}(g)$ *threads.*

*Proof sketch.* To prove the theorem, we need to generalize its statement to nonempty environments and general stacks: if $M;\gamma \Downarrow V;g$ then, for all $R$, $\ell \notin \mathsf{dom}(R)$, and $K$ of the form $\Box^{\ell}_{\mathbf{t}}$, $\Box^{\ell}_{\mathbf{a}} :: K'$, or $\Box^{\ell}_{\mathbf{a}} :: K'$, there exists $R' \sqsupseteq R$ such that

$$T + [\langle M \mid \gamma \mid K \rangle] \,/\, R[\ell \hookrightarrow \mathsf{nil}] \Rightarrow^{\mathsf{span}(g)} T' \,/\, R'$$

with $R'(\ell) = V$. In addition, this reduction sequence creates one thread per work item.

We proceed by induction over the cost-semantics derivation. The only interesting rule is AppClo, the case AppConst being a simpler variant. By the induction hypothesis, $M$ completes in $\mathsf{span}(g_1)$ steps and $N$ in $\mathsf{span}(g_2)$ steps. Assume that $\mathsf{span}(g_1) < \mathsf{span}(g_2)$. Then, $M$ terminates first, and $N$ must terminate with ASync, which triggers the evaluation of the closure body. This takes $\mathsf{span}(g_2)+\mathsf{span}(g_3)+2 = \max(\mathsf{span}(g_1), \mathsf{span}(g_2))+\mathsf{span}(g_3)+2$ steps, with the 2 additional steps coming from ESpawn and ASync. The case of $\mathsf{span}(g_2) < \mathsf{span}(g_1)$ is symmetric, with the argument term triggering the evaluation of the closure body. In the case of $\mathsf{span}(g_1) = \mathsf{span}(g_2)$, it is again the function term that triggers the evaluation of the closure, since $\Rightarrow_{\mathbf{f}}$-phases happen before $\Rightarrow_{\mathbf{a}}$-phases in the definition of $\Rightarrow$. □

### 2.3.2 From Abstract to Concrete

The structure of the parallel abstract machine makes makes it simple to implement on top of a low-level machine model. We choose the popular Parallel Random Access Machine (PRAM) model, in its Concurrent-Read Exclusive-Write (CREW) variant. A PRAM is a variant of the classic RAM machine, itself a convenient extension of the Turing machine. In a PRAM, $p$ processors access a shared memory in parallel. Each step of the machine performs one step on each processor. The CREW variant is reminiscent of the *data-race-free* model studied in L. Maranget's part of the course: any number of processors may read the same memory cell during the same step, but two processors writing to the same cell raises an error.

As usual with PRAM arguments, we describe the simulation in an informal way. The multiset $T$ of threads is represented in an array of size $q \triangleq |T|$. We fix an ordering of its threads. Processor $i \in [1, p]$ is responsible for threads $[iq/p, \ldots, (i+1)q/p-1]$. In addition to the thread array, memory also holds the contents of $R$. This part of memory, which we will call $R$-cells, is the only one that is *logically* shared between processors; however, no data race may occur. The simulation proceeds in four phases, at the end of which all processors wait for one another.

1. Each processor evaluates the $\Rightarrow_{\mathbf{e}}$ transitions on its segment of the thread array, updating it as needed. This requires no communication between processors, assuming we maintain a bound on the maximum memory cell used up to now, and use it in order to allocate fresh cells when simulating ESpawn. This guarantees that the only write to an $R$-cell, which occurs in ESpawn, cannot be concurrent with another write. Nor can it be concurrent to a read since those happen during later phases.

2. Each processor evaluates the $\Rightarrow_{\mathbf{f}}$ transitions on its segment of the thread array. At most one processor will read or write to the same $R$-cell during this phase.

3. Similarly for $\Rightarrow_{\mathbf{a}}$ transitions.

4. Finally, each thread has generated zero, one, or two threads to be processed during the next simulation step. They need to be flattened into a new thread array. This can

be accomplished using a prefix-sum computation to compute their offsets into the new array, as seen during the first lecture.

Of course, a more formal description would have to explain how to implement functional programming constructs on top of a (P)RAM machine. For example, we must assume the machine has unit-time operations implementing all the constants in PAL. The only non-trivial thing is the handling of environments, which can be implemented using balanced trees. This is why the following result features an overhead of $v_M$, where $v_M$ is the logarithm of the number of variables in $M$.

**Lemma 1.** *A simulation step runs in $kv_e(\lceil q/p \rceil + \log p)$ time, with $k$ some positive constant.*

**Theorem 2.** *If $M; \emptyset \Downarrow V; g$ then $M$ is computed in $kv_M(\mathsf{work}(g)/p + \mathsf{span}(g) \log p)$ time, with $k$ some positive constant.*

*Proof.* By combining Theorem 1 and Lemma 1 as in the proof of Brent's bound. □

Blelloch and Greiner [1995] also discuss the reverse direction, where one simulates an arbitrary PRAM algorithm onto the parallel abstract machine. This requires implementing the random-access memory in a purely functional way, which has some overhead.

# 3 A Stateful Calculus

While its simplicity makes it an appealing object of study, the calculus presented in the previous section is a bit too far from actual parallel programming languages. In particular, our experience writing Cilk code should have convinced us that state actually plays a crucial role when implementing real algorithms. For this reason, we would like to extend the calculus to deal with some global state.

One option would be to fix some notion of state, perhaps as a form of heap. We will rather construct a calculus parameterized by an arbitrary *partial commutative monoid* (PCM), a concept that arose in the study of separation logic. Its elements will provide the abstract notion of state, and its monoid law a way to split the state into distinct parts which can be acted upon independently.

We start by recalling the definition of a PCM before describing our calculus and its cost semantics. We then instantiate the construction to obtain an idealized variant of Cilk.

## 3.1 Partial Commutative Monoids

We follow the modular algebraic approach to PCMs outlined by Jung et al. [2015].

**Definition 2.** A *partial commutative monoid* $\mathbb{M}$ is a tuple $(|\mathbb{M}|, \cdot_{\mathbb{M}}, \epsilon_{\mathbb{M}}, \bot_{\mathbb{M}})$ where $(\mathbb{M}, \cdot_{\mathbb{M}}, \epsilon_{\mathbb{M}})$ is a commutative monoid and $\bot_{\mathbb{M}} \in |\mathbb{M}|$ is absorbing for $\cdot_{\mathbb{M}}$.

As usual, we omit the subscripts and brackets $|-|$ when the PCM is clear from the context. Given such a PCM, we write $(-) \odot (=)$ for the partial function from $\mathbb{M}^2$ to $\mathbb{M}$ such that $x \odot y \triangleq x \cdot y$ when $x \cdot y \neq \bot$, and is undefined otherwise. When defining the carrier of a PCM, we frequently omit the elements 1 and 0, and specify only $\odot$.

**Definition 3.** The *exclusive* PCM over a set $X$, denoted $\mathsf{Ex}(X)$, is $(X \uplus \{\epsilon, \bot\}, \cdot, \epsilon, \bot)$ with the monoid law defined only by $x \odot \epsilon = \epsilon \odot x = x$ for all $x \in X$.

$$\boxed{M;\gamma;A \Downarrow V;B;g}$$

$$\textsc{Const}\ \frac{}{c;\gamma;A \Downarrow c;A;\mathbf{1}} \qquad \textsc{Var}\ \frac{\gamma(x) = V}{x;\gamma;A \Downarrow V;A;\mathbf{1}} \qquad \textsc{Fun}\ \frac{}{\lambda x.M;\gamma;A \Downarrow (x.M)\{\gamma\};A;\mathbf{1}}$$

$$\textsc{AppClo}\ \frac{M;\gamma;A \Downarrow (x.S)\{\sigma\};A';g_1 \qquad N;\gamma;B \Downarrow V;B';g_2 \qquad S;A' \circledast B';\sigma[x \mapsto V] \Downarrow W;C;g_3}{M\,N;\gamma;A \circledast B \Downarrow W;C;(g_1 \otimes g_2) \oplus g_3 \oplus \mathbf{1}}$$

$$\textsc{AppConst}\ \frac{M;\gamma;A \Downarrow c;A';g_1 \qquad N;\gamma;B \Downarrow V;B';g_2 \qquad (V, A' \circledast B')\ \delta_c\ (W, C)}{M\,N;\gamma;A \circledast B \Downarrow W;C;(g_1 \otimes g_2) \oplus \mathbf{1}}$$

Figure 5: Cost Semantics of $\text{SPAL}_{\mathbb{M}}$

Given a map $f : X \to \mathbb{M}$, we define $\mathsf{dom}(f) \triangleq \{x \mid f(x) \neq \bot\}$.

**Definition 4.** The *finite map* PCM over a set $X$ and a monoid $\mathbb{M}$, denoted $\mathsf{FMap}(X, \mathbb{M})$, is defined as $(\{f \in \Pi_{x \in X}|\mathbb{M}| \mid \mathsf{dom}(f) \text{ finite}\} \cup \bot, *, x \mapsto \epsilon_{\mathbb{M}}, \bot)$ where $f \circledast g$ is defined as $x \mapsto f(x) \cdot g(x)$ only when $f(x) \cdot g(x) \neq \bot_{\mathbb{M}}$ for all $x \in \mathsf{dom}(f) \cup \mathsf{dom}(g)$.

**Definition 5.** The *plain heap* PCM over a set $X$, denoted $\mathsf{Heap}(X)$, is $\mathsf{FMap}(\mathit{Loc}, \mathsf{Ex}(X))$.

It is worth unfolding the last definition. The set of non-$\bot$ elements of $\mathsf{Heap}(X)$ is isomorphic to maps $\mathit{Loc} \to_{\mathsf{fin}} X$. The composite $f \circledast g$ is defined only when $\mathsf{dom}(f) \cap \mathsf{dom}(g) = \emptyset$, in which case it sends $\ell \in \mathsf{dom}(f) \cup \mathsf{dom}(g)$ to $f(\ell)$ if $\ell \in \mathsf{dom}(g)$ and to $g(\ell)$ otherwise.

We write $\ell \hookrightarrow x$ for the element of $\mathsf{Heap}(X)$ sending $\ell$ to $x$ and $\ell' \neq \ell$ to $\epsilon$.

### 3.2 The Generic Calculus

We now fix a PCM $\mathbb{M}$ and use it build our calculus, the Stateful Parallel Applicative Calculus over $\mathbb{M}$, which we abbreviate $\text{SPAL}_{\mathbb{M}}$. We write $A, B, C$ for the elements of $\mathbb{M}$.

This calculus and PAL share the same syntax. However, its language of constants will generally feature items related to $\mathbb{M}$, or contain operations able to act upon an element of the monoid. Its cost semantics, again expressed as a big-step judgment $M;\sigma;A \Downarrow V;B;g$, modifies that of PAL to express the relation between the initial state $A$ and the final state $B$ reached after evaluation has terminated. The rules of this semantics are given in Figure 5.

- Rules Const, Var, and Fun do not affect the state.
- Parallelism is handled via the monoid law: to apply rules AppClo and AppConst, one has to split the input state into distinct parts, but also to recombine the final states.
- Constants may alter the current state. It is convenient to generalize $\delta$ from a partial function to a family of relations $(\delta_c \subseteq (\mathit{Val} \times \mathbb{M})^2)_{c \in \mathit{Const}}$.

This means that the only way to do a primitive state modification is via the proper constants. To illustrate this fact, let us describe a useful family of constants for $\text{SPAL}_{\mathsf{Heap}(\mathbb{Z})}$.

$$c ::= i \mid \cdots \mid \ell \mid \texttt{alloc} \mid \texttt{read} \mid \texttt{write} \mid \texttt{write}_\ell$$

Their semantics is unsurprising.

$$\begin{aligned}(i, A)\,\delta_{\texttt{alloc}}\ & (\ell, A \circledast \ell \hookrightarrow i)\\ (\ell, A \circledast \ell \hookrightarrow i)\,\delta_{\texttt{read}}\ & (i, A \circledast \ell \hookrightarrow i)\\ (\ell, A)\,\delta_{\texttt{write}}\ & (\texttt{write}_\ell, A)\\ (i, A \circledast \ell \hookrightarrow j)\,\delta_{\texttt{write}_\ell}\ & (i, A \circledast \ell \hookrightarrow i)\end{aligned}$$

One should try to play with this definition. How does it differ from a language like Cilk?

### 3.3 A Cilk-like Instance

To allow for concurrent reads yet disallow races, we use *fractional* permissions, another well-known idea from separation logic [Boyland, 2003, Jung et al., 2015].

**Definition 6.** The PCM of *fractional permissions* over a set $X$, denoted $\mathsf{Frac}(X)$, is $((\mathbb{Q} \cap (0,1] \times X) \cup \{\epsilon, \bot\}, \cdot, \epsilon, \bot)$, where the monoid law is defined only by

$$(q, x) \odot (q', x') \triangleq (q + q', x) \text{ if } x = x' \text{ and } q + q' \leq 1.$$

**Definition 7.** The *fractional heap* PCM over a set $X$, denoted $\mathsf{Frac}(X)$, is $\mathsf{FMap}(\mathit{Loc}, \mathsf{Frac}(X))$.

Again, let us unfold this definition. The set of non-$\bot$ elements of $\mathsf{Frac}(X)$ is isomorphic to maps $\mathit{Loc} \to_{\mathsf{fin}} (\mathbb{Q} \cap (0,1]) \times X$. The composite $f \circledast g$ is defined only when for all $\ell \in \mathsf{dom}(f) \cup \mathsf{dom}(g)$, $f(\ell) = (q, x)$ and $g(\ell) = (q', x)$ and $q + q' \leq 1$, in which case it sends $\ell$ to $(q + q', x)$.

We write $\ell \overset{q}{\hookrightarrow} x$ for the element of $\mathsf{Heap}(X)$ sending $\ell$ to $(q, x)$ and $\ell' \neq \ell$ to $\epsilon$.

The semantics of constants only allows writing to a location $\ell$ that is fully owned, in the sense that $\ell \overset{1}{\hookrightarrow} i$ for some $i$. A location is fully owned by the code that created it.

$$\begin{aligned}(i, A)\,\delta_{\texttt{alloc}}\ & (\ell, A \circledast \ell \overset{1}{\hookrightarrow} i)\\ (\ell, A \circledast \ell \overset{q}{\hookrightarrow} i)\,\delta_{\texttt{read}}\ & (i, A \circledast \ell \overset{q}{\hookrightarrow} i)\\ (\ell, A)\,\delta_{\texttt{write}}\ & (\texttt{write}_\ell, A)\\ (i, A \circledast \ell \overset{1}{\hookrightarrow} j)\,\delta_{\texttt{write}_\ell}\ & (i, A \circledast \ell \overset{1}{\hookrightarrow} i)\end{aligned}$$

## References


Guy Blelloch and John Greiner. Parallelism in Sequential Functional Languages. In *Functional Programming Languages and Computer Architecture (FPCA '95)*. ACM, 1995. URL http://www.cs.cmu.edu/~blelloch/papers/BG95.pdf.

John Boyland. Checking interference with fractional permissions. Springer, 2003. URL http://www.cs.uwm.edu/faculty/boyland/papers/permissions.pdf.

Matteo Frigo, Charles Leiserson, and Keith Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Programming Language Design and Implementation (PLDI'98)*. ACM, 1998. URL http://supertech.csail.mit.edu/papers/cilk5.pdf.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. ACM, 2015. URL https://iris-project.org/pdfs/2015-popl-iris1-final.pdf.