On the Parallel Computation Thesis*

Nachum Dershowitz
School of Computer Science, Tel Aviv University, Tel Aviv, Israel
Insitut d'études avancées de Paris, Paris, France
nachum.dershowitz@cs.tau.ac.il

and

Evgenia Falkovich-Derzhavetz
School of Computer Science, Tel Aviv University, Tel Aviv, Israel
jenny.derzhavetz@gmail.com

Abstract

We develop a generic programming language for parallel algorithms, one that works for all data structures and control structures. We show that any parallel algorithm satisfying intuitively-appealing postulates can be modeled by a collection of cells, each of which is an abstract state machine, augmented with the ability to spawn new cells. The cells all run the same algorithm and communicate via a shared global memory. Using a formal definition of what makes such an algorithm effective, we prove the validity of the Parallel Computation Thesis, according to which all reasonable parallel models of computation have roughly equivalent running times.

La filosofia è scritta in questo grandissimo libro che continuamente ci sta aperto innanzi a gli occhi (io dico l'universo) ma non si può intender se prima non s' impara a intender la lingua e conoscere i caratteri ne' quali è scritto. Egli è scritto in lingua matematica e i caratteri sono triangoli, cerchi, ed altre figure geometriche senza i quali mezi è impossibile a intenderne umanamente parola; senza questi è un aggirarsi vanamente per un oscuro laberinto.[1]

—Galileo Galilei (*Il Saggiatore*)

*This work was carried out in partial fulfillment of the requirements for the Ph.D. degree [22] of the second author. The first author's research benefitted from a fellowship at the Paris Institute for Advanced Studies (France), with the financial support of the French state, managed by the French National Research Agency's "Investissements d'avenir" program (ANR-11-LABX-0027-01 Labex RFIEA+). A portion of this article was presented in a talk entitled, "Generic Parallel Algorithms," at *Computability in Europe (CiE) 2014* [16].

[1]"Philosophy is written in this grand book, the universe, which stands continually open

1 Introduction

Evolving systems—be they physical, biological, or computational—are typically viewable on many distinct levels of abstraction. It is a generic notion of a system of objects evolving concurrently, applicable at any level of abstraction, that we seek to capture. We shall refer to the individual objects composing such a system as "cells." In this work, all the component cells of a system live and operate at the same level of abstraction.

As Galileo observed in the above epigraph, the "manual" of the universe is written in mathematical language. This conviction means that the evolution of the universe and the evolution of its components can be expressed in mathematical terms. It has, furthermore, been convincingly argued by Yuri Gurevich [26] (presaged by Emil Post [31]) that logical structures are the right way to view evolving algorithmic states, just as they are ideal for capturing the salient features of static entities. The structure stores the values (taken from the structure's domain) of components of the state that are updated during the computation (such as program variables and program counters) as well as the state's functional capabilities (like the ability to perform arithmetic). We shall view cellular evolution from this perspective.

That there are multiple levels at which to understand the same overall system necessitates an abstraction mechanism. This means that the behavior of the entities at a higher level should be modeled independently of the underlying lower levels, which translates into the requirement that states qua structures are isomorphism-closed (making them oblivious as to how the domain values they deal with are in fact implemented) and that their evolution respects those isomorphisms. The importance of isomorphism-invariance for purposes of abstraction has been repeatedly emphasized [24, 10, 26].

We need to model communication between entities in addition to modeling their individual evolution. To that end, one can allow the control of one cell to access values in another cell—a shared-memory viewpoint, or one can allow cells to request values from other cells—a message-based framework. Similarly, one can allow a cell to set values in another cell or to request those changes of the other cell (depending again on one's viewpoint). Interaction and coöperation have been considered within Gurevich's framework [1, 2]. We take the shared-memory viewpoint here (as an abstraction that need not correspond to any physical reality) and, furthermore, assume that cells work in discrete time via a shared clock.

Many systems, be they natural or artificial, create new entities as they evolve in time. We will, therefore, need to model the "birth" of new component cells in what follows. But we will not, in this paper, consider changes in channels of

to our gaze. But the book cannot be understood unless one first learns to comprehend the language and read the letters in which it is composed. It is written in the language of mathematics, and its characters are triangles, circles, and other geometric figures without which it is humanly impossible to understand a single word of it; without these, one wanders about in a dark labyrinth." Translated by Stillman Drake, *Discoveries and Opinions of Galileo*, Doubleday, New York, 1957.

communication (the "topology") other than at birth (cf. [20]). Were it not for possible interaction with external agents and for the birth of new components, one might have been tempted to view a software system as one large evolving global "organism," rather than as a conglomerate of many interacting individual cells.

In 1976, Ashok Chandra, Dexter Kozen, and Larry Stockmeyer [11] proved that alternating polynomial time is equivalent to deterministic polynomial space. In 1977, Allan Borodin [7] suggested that this result may be generalized:

> *Parallel time and space are roughly equivalent within a polynomial factor.*

This thesis is commonly referred to as the *Parallel Computational Thesis*. In 1978, Steven Fortune and James Wyllie [23] defined a parallel random access machine (PRAM) and proved that "deterministic parallel RAM's with number processes no more then exponential can accept in polynomial time exactly the sets accepted by Turing machines with polynomially bounded tape."

Later, in 1986, Ian Parberry [30] showed that a Common PRAM with a bit more than an exponential number of initial processes can compute any NP problem in constant time.[2] He explains that, in his opinion, this example does not violate the parallel computational thesis but, rather, that this model (PRAM with already an exponential number of processors initially) should probably not be judged "reasonable":

> The parallel computational thesis does not attempt to say that time on *all* parallel machine models is related; ... it talks only about "reasonable" models.... Thus ... a model is a counterexample to the parallel computational thesis only if it is "reasonable."

In what follows, we prove that in fact polynomial time of *effective parallel algorithms*, in a sense to be made formal herein, is equivalent to Turing-machine polynomial space—provided the former use no more than an exponential number of cells. This is akin to our recent work on ordinary, sequential algorithms [15, 17], in which it was shown that all (formally) effective sequential models of computation can be polynomially simulated by a random access machine (RAM).

We proceed as follows: We begin with an informal description of parallel algorithms and their component cells. In Section 3, we axiomatize generic parallel algorithms. Section 4 provides a description of a parallel programming language, based on abstract state machines (ASMs) [25]. (Familiarity with ASMs would be of advantage to the reader, though we do include all necessary definitions.) Then, Section 5 proves that all parallel algorithms, as characterized here, can be programmed with the constructs of the proposed language. In other words, the behavior of every parallel algorithm, regardless of the objects it manipulates, can be described precisely in our generic programming language.

[2] To be precise: any language in NP can be recognized in constant time by a shared-memory machine with $O\left(2^{n^{O(1)}}\right)$ processors and word size $O(T(n)^2)$.

We go on to restrict this general parallel model to effective ones in Section 6, providing us with a working notion of effective parallel algorithm. This is followed by a discussion of how complexity is measured (Section 7) and a classification of parallel random-access machines (Section 8). With that in hand, we prove our main result (Theorem 23) in Section 9, showing that the only possibility for super-Turing efficiency on the part of an effective parallel algorithm is when there are more than an exponential number of processors to start off with.

We conclude with a discussion of these results.

2 System Evolution

Informally, a parallel algorithm consists of a (finite or infinite) set of *cells*, whose individual states all evolve according to the *same* algorithm. The *state* of each cell, at any moment, is a (logical) structure with a domain D and with a tripartite vocabulary $V = F \uplus F' \uplus G$ corresponding to *private (internal)* operations F, *public (global)* G, and *embryonic* F', the latter having the same similarity type as F. The idea is that the F' operations belong to the growing "embryo" within the cell. The "mother" cell is updating the values stored in her own operations F at the same time that she is preparing F' for when the embryo will be born and form its own "living" cell. (There could be any fixed number $k \in \mathbb{N}$ of embryonic copies $F', F'', \ldots, F^{(k)}$, one copy per potential offspring in any single step. But let us leave it simple for now: one child at a time.)

Computation proceeds in discrete steps. Each cell may change the interpretation (as functions over the domain D) that the state it is in gives to any of its private operations F. The interpretations of the operations in G may also be updated by any of the cells in the system. The values assigned by a cell to operations in F' will serve as initial values of the private operations of a newborn cell, as we will see. Initially, all cells agree on (the interpretation of) G and their copies of F' are pristine (that is, completely undefined).

All the individual cells run the same "classical" (that is, non-interactive sequential) algorithm in the sense of [26, 13]. Suffice it to say that the transition function of such an algorithm can be expressed as a set of conditional assignments, each of the form

$$\textbf{if } v_1 = w_1 \ \textbf{\&} \ v_2 \neq w_2 \ \textbf{\&} \cdots \textbf{ then } f(s_1, \ldots, s_n) := s_0$$

where f is any symbol in the vocabulary V of the program (having some arity $n \geq 0$) the v_i, w_j, s_k are arbitrary terms (in that vocabulary), and a condition is a conjunction of equalities and disequalities. When the condition is an empty conjunction, we can just omit the **if** $\cdots$ **then** part. At each step, all the conditional assignments are applied at once in parallel, with previous values available to all the assignments. Whenever all the conjuncts $v_i = w_i$, $v_j \neq w_j$of a condition hold for a given state (that is, when the values assigned by the state to the terms v_i, w_j result in a condition that evaluates to true), the conditional assignment results in an *update* $f(u_1, \ldots, u_n) \mapsto u_0$, where u_k $(k = 0, \ldots, n)$

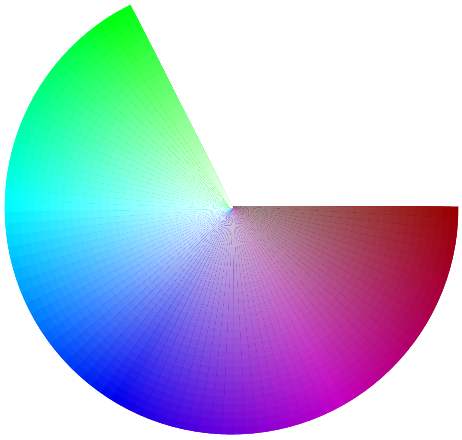


Figure 1: Stopwatch example after 40 minutes. The 2400 ticks are so close to each other that no whitespace can be discerned between them. (Best viewed in color.)

is the value assigned to s_k in the current state. The update has the effect of causing the state to re-assign the value u_0 to the *location* $f(u_1, \ldots, u_n)$, that is, to the graph of the operation f at the point $(u_1, \ldots, u_n)$. Should there be any disagreement between assignments regarding the value to be assigned to any cell location, the whole system aborts.[3] The notion of update will be formalized in the next section.

Example 1. Imagine a ticking stopwatch, its clock face displaying a colored minute hand of unit length. As the hand turns, it changes color, all the while leaving a fading trail behind.

We allocate one cell for each point along the watch hand (of which there are uncountably many). The state of each cell includes a scalar (nullary) symbol α to indicate whether it is active (in other words, it is a point along the current position of the moving hand) or passive (a point along a previous position of the hand). The cell uses symbols r (with nonnegative real value) and θ (a nonpositive angle in radians) for recording its position in radial notation, with r fixed throughout and θ changing clockwise. Each cell uses h, s, and b for specifying its color according to the HSB (hue [in degrees], saturation [in percentage points], brightness [in percentage points]) scheme,[4] with the brightness in past positions diminishing with time. All values are real numbers, taken from $\mathbb{R}$. See Figure 1.

The program changes the brightness b of the cell running it, deactivates it, and sets up a new point (same radius, different angle) by means of primed

[3] Abortion could be replaced with nondeterministic behavior, should one prefer.

[4] Alvy Ray Smith, Color gamut transform pairs, *ACM SIGGRAPH Computer Graphics* 12(3):12–19, 1978.

names. It uses two flags: a global flag β that stops all action when it is turned off (changes value from 1 to 0), and a flag α that is on (has value 1) for newborn cells only. The following set of conditional assignments constitutes the desired program:

$$\begin{array}{rll}
\textbf{if } \beta = 1 & \textbf{then} & \alpha := 0 \\
\textbf{if } \beta = 1 & \textbf{then} & b := b - 1/36 \\
\textbf{if } \alpha = 1 \ \& \ \beta = 1 & \textbf{then} & \alpha' := 1 \\
\textbf{if } \alpha = 1 \ \& \ \beta = 1 & \textbf{then} & r' := r \\
\textbf{if } \alpha = 1 \ \& \ \beta = 1 & \textbf{then} & \theta' := \theta - \pi/1800 \\
\textbf{if } \alpha = 1 \ \& \ \beta = 1 & \textbf{then} & h' := h + 1/10 \\
\textbf{if } \alpha = 1 \ \& \ \beta = 1 & \textbf{then} & s' := s \\
\textbf{if } \alpha = 1 \ \& \ \beta = 1 & \textbf{then} & b' := b \\
\textbf{if } \beta \neq 0 \ \& \ \theta = 0 & \textbf{then} & \beta := 0
\end{array}$$

The global vocabulary is $G = \{0, 1/36, 1/10, 1, \pi, 1800, +, -, /, \beta\}$, where 0, $1/36$, $1/10$, 1, π, and 1800 are scalar constants containing (in other words, interpreted as having) the expected values, and the global arithmetic operations $+$, $-$, and $/$ are interpreted as usual. The local vocabulary is $F = \{\alpha, r, \theta, h, s, b\}$. Among the global symbols, only the value of the flag β changes. The value of β must be 1 (on, as it is at the outset) for anything to happen.

At each step, one per second, the conditions $\beta = 1$, $\alpha = 1$ & $\beta = 1$, and $\beta \neq 0$ & $\theta = 0$ are evaluated in the current state, with every term evaluated according to interpretations currently given to the symbols in the vocabulary. All those assignments whose condition holds true cause updates, which are then applied to the interpretations given by the cell's state to the symbols in its vocabulary. For example, if β has global value 1 and the local location $b()$ in the state currently has value 50, then b is reassigned according to the update $b() \mapsto 49.97\overline{2}$ derived from the assignment $b := b - 1/36$.

Initially, the state has $\alpha = \beta = 1$ (both active), $\theta = 2\pi$ (pointing right to "3 o'clock"), $r \in [0..1]$, and $h = 0$ (a red hue). For each $r \in [0..1]$, there is a cell C_r with initial saturation $s = 100 \times r$ proportional to its distance from the center, and initial brightness $b = 100$, which is maximal. If θ is initially 2π, then after 3600 iterations of the above program it will be 0 and β will be reset to 0, at which point no assignment statement will apply again. When no assignment applies (because no condition evaluates to true), the algorithm halts in its current state. Unlike other states, this *terminal* state is not followed by a "next" state. □

A single *global step* of the algorithm comprises the following stages.

1. First, each cell C takes one step on its own, according to the classical algorithm it is running, producing a set U of *updates*, specifying changes to the values of locations in its state.

2. Then, cells' private operations F and embryonic operations F' are updated per U.

3. At the same time, the union of all the cells' public updates together are applied to (every cell's view of) the public G, updating the interpretations given to the operations G. If there is any disagreement between cells regarding these updates (the same location getting contradictory new values), the whole system aborts.

4. Assuming there are no conflicts, *mitosis* takes place as follows: Each cell C in which the values of any operations in F' were modified splits into two cells, a mother C and daughter C'. The daughter C' inherits G, as updated, from her mother; her F is a copy of her mother's F'. For both mother and daughter, F' is re-initialized to the default *undefined* value everywhere.

5. If one wishes, an individual cell can be allowed to *die* and be dropped from the global organism whenever it has no local next state, as when it suffers an internal clash between attempted updates.

Example 1 (Continued). In our stopwatch example, with every step of the parallel algorithm, each cell along the minute hand creates a duplicate cell (the primed symbols) on a line positioned one second ($-\pi/1800$ radians) forward, while the points along the current line dim (its brightness level decreased proportionately to the elapsed time). After one hour (3600 seconds), the minute hand returns to its initial position ($\theta = 0 = 2\pi$ radians) and everything shuts down ($\beta = 0$). At this point, a cell at position (r, θ) will have attained its final color $(h, s, b) = (360 - 180\theta/\pi, 100r, 100 - 50\theta/\pi)$. □

3 Parallel Algorithms

3.1 Parallel Systems

An algorithm is ordinarily viewed as a kind of state-transition system composed of a set of states and a transition function (or, more generally, a relation) over states [29].

Definition 1 (Parallel System)**.** A *parallel system* $\langle \mathcal{S}, \mathcal{S}^0, \eta, \mathcal{I}, \lambda \rangle$ consists of

- a set[5] $\mathcal{S}$ of *states*,
- a (nonempty) subset $\mathcal{S}^0 \subseteq \mathcal{S}$ of which are *initial*,
- a (partial) *transition* (next state) function $\eta : \mathcal{S} \rightharpoonup \mathcal{S}$,
- a set of *identifiers* $\mathcal{I}$, collectively called a *conglomerate*, and
- a *localization* operator $\lambda : \mathcal{S} \xrightarrow{1-1} \mathcal{S}^{\mathcal{I}}$ that injectively maps each state $X \in \mathcal{S}$ to an identifier-indexed set $\{X_i\}_{i \in \mathcal{I}}$ of *local* states $X_i \in \mathcal{S}$.

[5] Or class. We gloss over the distinction.

Whenever η is undefined for a state $X \in \mathcal{S}$, we say that X is *terminal*.[6] By $S^{\ddagger}$ we denote these terminal states in $\mathcal{S}$.

From now on, we write $X_{\mathcal{I}}$ for the set of local states $\lambda(X)$ of state X and X_i for its localization to a particular identifier $i \in \mathcal{I}$. By $\mathcal{S}^{\lambda} = \bigcup_{X \in \mathcal{S}} X_{\mathcal{I}} \subseteq \mathcal{S}$ we denote the set of all local states. These are the possible states of the individual cells. Every state X can always be recovered from its localizations $X_{\mathcal{I}}$, that is, $\lambda^{-1}(X_{\mathcal{I}}) = X$.

A *computation* for such a system is a finite or infinite sequence

$$X^0 \rightsquigarrow_{\eta} X^1 \rightsquigarrow_{\eta} \cdots$$

of states $X^i \in \mathcal{S}$, such that $X^0 \in \mathcal{S}^0$, $X^{i+1} = \eta(X^i)$ for each $i = 0, 1, \ldots$, and the sequence is finite only if its last state is terminal.

We explain next what the states of a parallel algorithm should look like and then discuss what makes transitions algorithmic.

3.2 Abstract States

A parallel algorithm involves multiple local processes, what we called "cells," all executing the very same algorithm. Each cell in a conglomerate has its own unique identifier, taken from the index set $\mathcal{I}$. In what follows, we refer to a cell by its identifier $i \in \mathcal{I}$.

Bear in mind that, in this work, cells are unaware of their own identifier or the identifiers of other cells. Identifiers are just an artifact to allow *us* to distinguish one cell from another. (Parent cells may however impart their children with some of their own private knowledge.) When comparing states, the individual identifiers should be ignored: Two global states are the "same" if they are the same up to permutation of cell identifiers. (Nothing precludes there being two cells with separate identifiers but identical behavior, each always in the same state as the other.) Similarly, two computations are the same if there is a permutation of identifiers for the states in one that makes it identical to the other.

As explained in the beginning of the previous section, the states of the cells of algorithmic systems are formalized as (first-order) logical structures over some vocabulary fixed by the algorithm. Since each cell is running the same algorithm, the states of all cells must have similar working vocabularies (symbols and arities). Furthermore, during any run, all cells need to operate over the same domain, as they are sharing global values. Since cells do not communicate with each other, except via global locations, there is no need for the algorithm to explicitly refer to identifiers or to operations in other cells. But, since we are dealing with parallel algorithms with both private and shared memory, we will need to distinguish between shared operations and local ones. Accordingly, at any given moment each cell i is in a *local* state, which is a structure with some domain (universe, carrier) D and vocabulary $G \uplus F_i$ that interprets its

[6] Alternatively, one could declare that terminal states are their own next state.

operations $G \uplus F_i$ as functions over D. The (current) meanings of operations in G are stored (conceptually, at least) in *global* locations, accessible to all cells, while private data is stored in the cell's private copy of F_i.

The evolution of the ith cell should utilize only the values of the defined operations of the ith localization. Transitions for a cell can only change values of its own operations and that of its progeny. We say that a local state X_i is *dormant* if every one of its private operations, those in F_i, is fully undefined ($f_i(\bar{u}) = \bot$ for all $\bar{u}$), using some distinguished domain value $\bot$, signifying "undefined," for the purpose. These are nascent cells, yet to be born. We will refer to local state X_i as an *i-state* once it's no longer dormant. The operations in states are always *strict*, meaning that $f(\ldots, \bot, \ldots) = \bot$ for all f.

Postulate I (Abstractness)**.** A parallel system $\langle \mathcal{S}, \mathcal{S}^0, \eta, \mathcal{I}, \lambda \rangle$ is *abstract* if the following properties hold:

1. All states $\mathcal{S}$ are (first-order) structures over the same vocabulary V.[7]
2. Transitions η and localization λ preserve the domain of states.
3. Its states $\mathcal{S}$—and also the set of initial states $\mathcal{S}^0$ and the set of terminal states—are closed under isomorphism (of first-order structures).
4. Isomorphic states are either both terminal or else their next states are isomorphic, via the same isomorphism.

States as structures make it possible to consider any data structure sans encodings. In this sense, algorithms are generic. The structures are "first-order" in syntax, though domains may include sequences, or sets, or other higher-order objects, in which case the state would provide operations for dealing with those objects. States with infinitary operations, like the supremum of infinitely many objects, are precluded. Closure under isomorphism ensures that the algorithm can operate on the chosen level of abstraction and that states' internal representation of data is invisible to the algorithm. This means that the behavior of an *algorithm*, in contradistinction with its "implementation" as a program in some particular programming language, cannot depend on the memory address of some variable.

Let X be a state of an abstract parallel system with domain D, and let g be some operation in V of arity n and $\bar{u} \in D^n$ be an n-tuple of elements of that domain. By *interpreting* g, state X determines the value $[\![g]\!]_X(\bar{u})$ of its *location* $g(\bar{u})$. For any ground term t, we write $[\![t]\!]_X$ for the value of t as interpreted in X, where $[\![g(t^1, \ldots, t^n)]\!]_X = [\![g]\!]_X([\![t^1]\!]_X, \ldots, [\![t^n]\!]_X)$. Since states are structures, we may view each state X as the set of *point-values* of the form $g(\bar{u}) \mapsto [\![g(\bar{u})]\!]_X$, giving the values stored in each location $g(\bar{u})$ of X.

Each cell works with only part of the global state at its disposal. For that purpose, each cell $i \in \mathcal{I}$ has a *working* vocabulary $V_i = G \cup F_i \subseteq V$, consisting of global operations G plus private ones F_i. Put differently, the *global* operations

[7] Because cells reproduce, it is simpler to view a cell as a restricted *view* of the global state than as a piece of the state that can grow or multiply.

$G = \bigcap_{i \in \mathcal{I}} V_i$ are those that are shared among cells;[8] the *local* operations $F_i = V_i \setminus G$ are the others. Cell i, as it evolves, will not touch outside operations (F_j for $j \neq i$), except perhaps in the process of giving birth to new cells.

The whole point of a parallel system is that its global state is the sum total of the local states of its cells. Taking advantage of the "set of points" way of viewing states, we require the following:

Postulate II (Cellularity)**.** An abstract parallel system $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \eta, \lambda \rangle$ is *cellular* if

1. Every local state $X_i \in \mathcal{S}^\lambda$ has a finite *working* vocabulary $V_i \subseteq V$, all of which have the same fixed similarity type (their arities match).

2. For all states $X \in \mathcal{S}$,
$$X = \bigcup_{i \in \mathcal{I}} X_{\restriction i}$$
where $X_{\restriction i} = X_i \restriction V_i$ is the set of point values for just the working vocabulary V_i of cell X_i.

It follows that the vocabulary V of a state X is the union of the working vocabularies V_i of its localizations X_i. Furthermore, different localizations X_i of the same global state X cannot disagree about the interpretation of any of the shared global operations G.

We mean for the ith localization of a localization X_i to be X_i itself and for all other localizations of X_i to have completely undefined local operations. A global state X is not the localization of any state, as it has many distinct localizations. Its different localizations only share global points $g(\bar{u}) \mapsto v$ for $g \in G$; the point values for local operations (F_i) reside in the relevant local state (X_i).

Suppose $F_i = \{f^1, \ldots, f^k\}$. Then, for our convenience, we will add indices to the individual local operations, imagining that $F_i = \{f_i^1, \ldots, f_i^k\}$, where, for each $j = 1, \ldots, k$, the operations f_i^j have the same arity for all $i \in \mathcal{I}$. It will also be convenient in what follows to denote by $F^j = \{f_i^j \mid i \in \mathcal{I}\}$ the corresponding local operations f^j across all cells. The *global* state of the algorithm will be a structure over the combined (usually infinite) vocabulary $V = G \cup F^*$, where $F^* = \bigcup_{i \in \mathcal{I}} F_i = \bigcup_{j=1}^k F^j$.

3.3 Algorithmic Transitions

A parallel system is considered to be "algorithmic" if it is possible to fully and finitely describe the actions of the transition function η on states. If every referenced location in the description of an algorithm has the same value in two states, then the behavior of the algorithm ought to be the same for both of those states. This, the essence of what makes a process algorithmic, is a crucial insight of [26].

[8] We are assuming—without loss of generality—that if two states share a working symbol, then all do.

When state X with transition η is not terminal, we say that the point $g(\bar{u}) \mapsto v$ is an *update* of X if η changes the value of $g(\bar{u})$ to be v in $\eta(X)$, which was not its value in X. By $\Delta^\eta(X)$, we denote the set of all updates to X under η. As we are viewing a state X as a set of points, we have $\Delta^\eta(X) = \eta(X) \setminus X$. This update function Δ^η characterizes the transition function η. We expect that the set of updates always be finite for local cells.

To facilitate state comparison, we define a *anonymization* operator $^\sharp$ that wipes off the identifier, that is, it erases the id-index from operation symbols.[9] Let $V^\sharp = G \cup F^\sharp$ be the anonymized vocabulary, where each $f \in F$ is a symbol of the same arity as $f_i \in F_i$ for each $i \in \mathcal{I}$. The anonymized local state $X_i^\sharp$ is obtained from i-state X_i by restricting attention to its working vocabulary V_i and pretending that the private symbols f_i^j are simply f^j, for all j. To compare the states of different cells, we should ignore their specific identifiers. Accordingly, we say that $X_i = Y_k$, for two local states, if $X_i^\sharp = Y_k^\sharp$, that is, if the two are identical when anonymized. This entails that global operations G have the same meanings in X as in Y and that local operations F_i have the same meaning in X_i as the corresponding F_k have in Y_k.

We say that transition η generates the "same" updates for X_i and Y_k, and write $\Delta^\eta(X_i) = \Delta^\eta(Y_k)$, if the updates to global operations G are the same in both X_i and Y_k, the updates to private operations F_i in X_i are the same as the updates to F_k in Y_k, and the updates to other locations are the same up to the choice of indices for updates to daughter cells. We denote by $\Delta_i^\eta(X)$ the set of all updates to locations of F_i in X. As before, we write $\Delta_i^\eta(X) = \Delta_k^\eta(Y)$ if $\Delta_i^\eta(X)^\sharp = \Delta_k^\eta(Y)^\sharp$.

At the heart of algorithmicity is the requirement that it be possible to describe the local effects of transitions in terms of the information in each local state. To that end, we will make use of *templates*, which refer to global locations in the current state as well as to local locations in each cell. Parallel algorithms use these templates to describe state transitions, without referring to individual cells.

Templates are designed to capture the uniform behavior of cells via terms over an unadorned vocabulary $V^\sharp$. For each $i \in \mathcal{I}$, a template t induces a *critical* term t_i, obtained by replacing each occurrence of an anonymous symbol f^j in t by the specific f_i^j in the working vocabulary of cell i.

Let X_i and X_j be distinct localizations of global state X. We write $X_i \equiv_T X_j$ and say that the two states *agree* if $[\![t_i]\!]_X = [\![t_j]\!]_X$ for each template $t \in T$. Similarly, we may compare localizations of distinct global states. We write $X_i \equiv_T Y_j$, for localization X_i of X and Y_j of Y, if $[\![t_i]\!]_X = [\![t_j]\!]_Y$ for each $t \in T$. States X and Y agree on T, indicated $X \equiv_T Y$, if there is a bijection $i \leftrightarrow i'$ of identifiers $\mathcal{I}$ such that $[\![t_i]\!]_X = [\![t_{i'}]\!]_Y$ for all $t \in T$ and $i \in \mathcal{I}$. In other words, every template has the identical value in the two states when localized to the vocabulary of the corresponding cells.

Each cell is fully responsible for its local updates. The updates generated by an individual cell may not depend on its identifier, but only on global and

[9] Or—more generally—identifies the matching operation symbols in different local states.

local locations that are available to it. Cells cannot interfere with each other's behavior.

Postulate III (Locality)**.** An abstract cellular system $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \eta, \lambda \rangle$ with vocabulary V is *localized* if there exists a finite set T of templates over $V^\sharp$ such that

$$\Delta^\eta(X_i) = \Delta^\eta(Y_j)$$

whenever $X_i \equiv_T Y_j$, for any local states $X_i, Y_j \in \mathcal{S}^\lambda$.

3.4 Childhood

Lastly, we need to consider what it means for a cell to engender new cells.

Each newborn cell has exactly one mother who does all the initialization work; older cells can only be modified by themselves:

Postulate IV (Motherhood)**.** An abstract cellular system $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \eta, \lambda \rangle$ is *maternal* if the following hold:

1. For every newborn cell $k \in \mathcal{I}$ of state $X \in \mathcal{S}$ (X_k is a newborn if it is dormant but $\eta(X)_k$ is not), there is a "mother" cell $m \in \mathcal{I}$ such that

$$\Delta^\eta_k(X) \subseteq \Delta^\eta(X_m)$$

2. Every other (nondormant) cell $k \in \mathcal{I}$ produces its own updates:

$$\Delta^\eta_k(X) = \Delta^\eta_k(X_k)$$

Once a cell has been born, no one else modifies its internals. They are autonomous.

Postulate V (Fertility)**.** An abstract cellular system $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \eta, \lambda \rangle$ is *fertile* if there exists a bound $n \in \mathbb{N}$, such that, for any local state $X_i \in \mathcal{S}^\lambda$, its next state $\eta(X_i)$ has at most n nondormant localizations.

The idea here is that each cell may participate in the creation of only a bounded number (independent of the particular cell and its current state) of new processes in a single step. This is in contrast with the possibly infinite number of spawned cells in the formalization of parallel algorithms in [2].

3.5 Parallel Algorithms

The **abstractness** and **locality** postulates are akin to those for sequential algorithms [26]. The **cellularity** and **motherhood** postulates capture what makes evolution parallel, while **fertility** places a limit on how much a system can grow in one parallel step.

With the above requirements in place, we can now state what a parallel algorithm is.

Definition 2 (Parallel Algorithm)**.** A *parallel algorithm* is a parallel state-transition system (according to Definition 1) that satisfies Postulates I–V.

Proposition 3. *For any parallel algorithm* $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \eta, \lambda \rangle$*, if* $X_i \equiv_T Y_j$ *for nondormant cells* $X_i, Y_j \in \mathcal{S}^\lambda$ *and templates* T*, then*

$$\Delta_i^\eta(X) = \Delta_j^\eta(Y)$$

This does not hold for newborns X_i and Y_j, which, though they agree on T before birth, are likely different after birth.

Proof. By **locality**, $\Delta^\eta(X_i) = \Delta^\eta(Y_j)$, which entails $\Delta_i^\eta(X_i)^\sharp = \Delta_j^\eta(Y_j)^\sharp$. By **motherhood**, $\Delta_i^\eta(X)^\sharp = \Delta_j^\eta(Y)^\sharp$, or $\Delta_i^\eta(X) = \Delta_j^\eta(Y)$. □

The updates of a parallel algorithm are determined by the updates to its cells:

Proposition 4. *For any parallel algorithm* $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \eta, \lambda \rangle$*, for all nonterminal states* $X \in \mathcal{S} \setminus \mathcal{S}^\ddagger$*,*

$$\Delta^\eta(X) = \bigcup\nolimits_{i \in \mathcal{I}} \Delta^\eta(X_i)$$

Proof. For local updates, we have for any cell k, thanks to **motherhood**, that $\Delta_k^\eta(X) \subseteq \Delta^\eta(X_m)$ for some cell m, the mother in the case of a newborn and k itself otherwise. Global updates must also be part of cellular updates $\Delta^\eta(X_i)$, as a consequence of **cellularity**. So, $\Delta^\eta(X) \subseteq \bigcup_{i \in \mathcal{I}} \Delta^\eta(X_i)$.

In the other direction, if δ is an update in some $\Delta^\eta(X_i) = \eta(X_i) \setminus X_i$ for a nondormant X_i, then, by **cellularity**, $\delta \in \eta(X)$ as a point value. But δ is not in X_i, nor is it in $X_{\restriction j}$ for $j \neq i$, either because it is local to i or because cells agree on global values. Finally, if δ updates a newborn cell k, then by **cellularity**, $\delta \in \eta(X)$ and by assumption it was nascent in X. □

In [26], ordinary (nonparallel) algorithms are required to satisfy a **bounded exploration** postulate, analogous to the following—but for a lone process rather than a system of processes and with finitely many ground terms rather than templates:

Definition 5 (Algorithmic System)**.** An abstract cellular system $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \eta, \lambda \rangle$ is *algorithmic* if there exists a finite set T of templates such that $\Delta^\eta(X) = \Delta^\eta(Y)$ whenever $X \equiv_T Y$, for any states $X, Y \in \mathcal{S}$.

Proposition 6. *Every parallel algorithm is algorithmic.*

Proof. Let $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \eta, \lambda \rangle$ be a parallel algorithm and suppose $X \equiv_T Y$ for nonterminal states $X, Y \in \mathcal{S}$ and some finite set of templates T, meaning that corresponding local states of X and Y assign the same values to all templates. For every update $\delta \in \Delta^\eta(X)$ of X, by Proposition 4, there must be a cell $i \in \mathcal{I}$ such that $\delta \in \Delta^\eta(X_i)$. Since $X \equiv_T Y$, there exists a cell $i' \in \mathcal{I}$ of Y such that $X_i \equiv_T Y_{i'}$. From **locality** we deduce that δ is also an update of $Y_{i'}$, and again by Proposition 4 that it is an update of Y. Hence, $\Delta^\eta(X) \subseteq \Delta^\eta(Y)$. Inclusion of $\Delta^\eta(Y)$ in $\Delta^\eta(X)$ is symmetric. Therefore, $\Delta^\eta(X) = \Delta^\eta(Y)$, as required. □

In the simple case when parallelism is bounded, a parallel algorithm is an instance of the ordinary sequential algorithms of Gurevich [26].

Proposition 7. *Any parallel algorithm for a system with a* finite *conglomerate may be described by an ordinary, sequential abstract state machine.*

Proof. The sequential algorithm operates over the global states of the parallel algorithm. The set of templates T that satisfies **locality** refers to only finitely many operations $f^j \in F$ and $g^j \in G$. The vocabulary V of the global states may be restricted to just those global g^j plus all the local f_i^j for those f^j appearing in the templates and for all the finitely many $i \in \mathcal{I}$. Also, we can instantiate T for all $i \in \mathcal{I}$ so that it becomes a finite set of *terms* over V, instead of templates. The requirements for an abstract state machine are fulfilled thanks to the **abstractness** (Postulate I) and **algorithmic** (Proposition 6) properties. So what we have is a *classical (sequential) algorithm* with *critical terms* T as defined in [26]. □

4 Parallel Abstract Machines

The two basic commands of parallel programs are (guarded parallel) assignment and (guarded) creation. They are normally expressed in the anonymous vocabulary $V^\sharp$ of templates and can be applied to any local state.

Assignment. An *atomic assignment* is a command of the form $f(t^1, \ldots, t^n) := t^0$, where $t^0, \ldots, t^n$ are templates and $f \in V$ is of arity n.

Let X_i be a nondormant i-state and suppose $[\![t_i^j]\!]_X = u_i^j$ for $j = 0, \ldots, n$. If $f \in F$, then application of an assignment a to the local state X_i generates a local update $\Delta^a(X_i) = \{f_i(u_i^1, \ldots, u_i^n) \mapsto u_i^0\}$ for that identifier i.[10] If $f \in G$, then it produces a global update $\Delta^a(X_i) = \{f(u_i^1, \ldots, u_i^n) \mapsto u_i^0\}$. If any one of the t^j is undefined ($u_i^j = \bot$) in X_i, then $\Delta^a(X_i) = \varnothing$. No updates are created for dormant localizations.[11]

The application of a to a global state X generates the combined update set $\Delta^a(X) = \bigcup_{i \in \mathcal{I}} \Delta^a(X_i)$. Whenever $\Delta^a(X)$ includes conflicting updates—from different cells—to the same global location, executing the command fails and the system aborts.

Parallel assignment. More generally, a *parallel assignment* is a command consisting of a finite set of atomic assignments, written out as $[\, a_1 \parallel a_2 \parallel \cdots \parallel a_\ell \,]$, using $\parallel$ as punctuation between the assignments in the set.

[10] The similarity type of the vocabulary V_i of X_i tells us which operator $f_i \in V_i$ corresponds to $f \in V^\sharp$.

[11] The postulates for parallel algorithms do not actually preclude the spontaneous awakening of all dormant cells at one and the same time, signaled by some configuration of global values. Should such behavior be desired, the corresponding program would need to be applied to dormant cells, too.

The update set generated by parallel assignment $a = [\, a_1 \parallel a_2 \parallel \cdots \parallel a_\ell \,]$ is $\Delta^a(X) = \bigcup_{j=1}^{\ell} \Delta^{a_j}(X)$. Whenever $\Delta^a(X)$ includes conflicts (for global or local locations), execution fails.

Creation. This is a command taking the syntactic form **new** a, where a is a parallel assignment. We abbreviate it $\nu.a$. Each such statement, when executed, results in another child.

Suppose a is a single assignment $f'(t^1, \ldots, t^n) := t^0$, and let X_i be a nondormant i-state. The prime signifies that f belongs to the vocabulary of the embryonic cell. Primed symbols appear at the head of the left sides of the assignments in a creation command. The transition initializes some dormant localization X_k by setting its $f_k([\![t_i^1]\!]_X, \ldots, [\![t_i^n]\!]_X)$ to $[\![t_i^0]\!]_X$. Then $\Delta^{\nu.a}(X_i) = \{f_k(u_i^1, \ldots, u_i^n) \mapsto u_i^0\}$, where each $u_i^j = [\![t_i^j]\!]_X$. As before, if any one of the u_i^j is undefined, then $\Delta^{\nu.a}(X_i) = \varnothing$. For each mother cell i, the transition chooses a different daughter cell k. In general, the update is appended to the total set of updates $\Delta^{\nu.a}(X) = \bigcup_{i \in \mathcal{I}} \Delta^{\nu.a}(X_i)$. If a is a parallel assignment $[a_1 \parallel a_2 \parallel \ldots \parallel a_\ell]$, then application of $\nu.a$ chooses a unique dormant X_k for each X_i such that all the templates in the atomic assignments are defined for X_i. In this case, $\Delta^{\nu.a}(X) = \bigcup_{j=1}^{\ell} \Delta^{\nu.a_j}(X)$. If there is no way to choose a unique k for each i such that the command can be applied to it (there are not enough unassigned identifiers in $\mathcal{I}$), then executing the creation command fails.

Guard. An *atomic guard* is a condition of the form $t = s$ or $t \neq s$. A guard $t = s$ evaluates to T (true) for i-state X_i if $[\![t_i]\!]_X = [\![s_i]\!]_X$, and $t \neq s$ is T if $[\![t_i]\!]_X \neq [\![s_i]\!]_X$. More generally, a *guard* may be a conjunction c of atomic guards c_1 **&** c_2 **&** $\ldots$ **&** c_n, which is T for X_i when each c_j is.

Guarded assignment. This is a command programmed as **if** c **then** a, where c is a guard and a is a parallel assignment, and denoted $c\colon a$ for short. Application of $c\colon a$ to state X generates the set of updates $\Delta^{c\colon a}(X) = \bigcup\{\Delta^a(X_i) \mid \in \mathcal{I},\ [\![c]\!]_{X_i} = \mathrm{T}\}$.

Guarded creation. This command takes the form **if** c **then new** a, where a is a parallel assignment and c, a guard. Abbreviate this $c\colon \nu.a$. The assignments are executed on each X_i for which the guard c evaluates to T. We used primed symbols to refer to locations in the new child's state. For example, $f'(x) := g(x, y)$, with g global and f, x, y local, has the effect $f_k([\![x_i]\!]X) \mapsto [\![g(x_i, y_i)]\!]X$. where k is the child's identifier and i is the parent's.

Definition 8 (Parallel Abstract Machine [PAM])**.** A *parallel abstract (state) machine (PAM)* is a parallel algorithm whose transition function is defined by a *(PAM) program* P that consists of a finite set $\{r_1, \ldots, r_n\}$ of guarded parallel assignments and creation commands (as just described). The effect of P on a state X is obtained by executing all its commands simultaneously, that is,

$\Delta^P(X) = \bigcup_{r \in P} \Delta^r(X)$. Should $\Delta^P(X)$ have conflicting updates, then they are not applied and execution aborts.

For state X and program P, $P(X)$ denotes the state obtained by applying P to X. If no command in P applies, then P is undefined for X.

Note that for each instance of cell creation, the program chooses new unused indices from $\mathcal{I}$ in some fashion. Since we always treat states and computations as identical if they are the same up to permutation of cells (that is, of indices to operation symbols), the specific choice of index is inconsequential.

Example 1 (Continued). With the above commands in hand, the program for the stopwatch example from Section 2 could look as follows:

$$\begin{aligned}
&\textbf{if } \beta = 1 \textbf{ then } [\quad \alpha := 0 \\
&\qquad\qquad\qquad\qquad \| \quad b := b - 1/36 \ ] \\
&\textbf{if } \alpha = 1 \ \& \ \beta = 1 \textbf{ then new } [\quad \alpha' := 1 \\
&\qquad\qquad\qquad\qquad \| \quad r' := r \\
&\qquad\qquad\qquad\qquad \| \quad \theta' := \theta - \pi/1800 \\
&\qquad\qquad\qquad\qquad \| \quad h' := h + 1/10 \\
&\qquad\qquad\qquad\qquad \| \quad s' := s \\
&\qquad\qquad\qquad\qquad \| \quad b' := b \ ] \\
&\textbf{if } \beta \neq 0 \ \& \ \theta = 0 \textbf{ then } \beta := 0
\end{aligned}$$

The assignment $r' := r$, for example, sets the value of the radial coördinate r in the child cell that is about to be born equal to its current value in the parent cell. □

5 Representation Theorem

A parallel program P is a *characteristic program* of algorithm $\mathcal{A}$ if $P(X) = \eta(X)$ for each state X of $\mathcal{A}$, that is, if it fully and precisely describes a single step of the parallel algorithm.

For simplicity in the descriptions that follow, we shall presume that $\mathcal{A}$ is over a local vocabulary $F = F^1$, with one local operation only. We will also assume that mothers give birth to at most one child per step ($n = 2$ in the **fertility** postulate). All proofs can be easily extended to the general case.

According to Proposition 4, $\Delta^\eta(X) = \bigcup_i \Delta^\eta(X_i)$. So we start with localized i-states X_i, and show that the transition of a single X_i can be described by a command composed of guarded parallel assignment and creation commands.

By our simplifying assumption, the algorithm has only one local operation, f. So X_i's defined locations are over the vocabulary $G \cup \{f_i\}$. Furthermore, we limit creation to at most one child per transition. Hence, defined locations of $\eta(X_i)$ are over $G \cup \{f_i, f_k\}$ for some embryonic $k \in \mathcal{I}$, $k \neq i$. So we may treat X_i and $\eta(X_i)$ as ordinary states of an ordinary (nonparallel) algorithm over finite vocabulary $G \cup \{f_i, f_k\}$ with critical terms $T_i \cup T_k$.

Let $\delta = h(u_1, \ldots, u_n) \mapsto v$ be an update in $\Delta^\eta(X_i)$. According to [32, Lemma 5] (or [26, Lemma 6.2]), for each value $u_j = 0, \ldots, n$ there exists a term $t^j \in T_i \cup T_k$ such that $[\![t^j]\!]_{X_i} = u_j$. Let α_δ be the ordinary assignment command $h(t^1, \ldots, t^n) := t^0$. We have $\Delta^{\alpha_\delta}(X_i) = \{\delta\}$. Denote by α_i the assignment obtained as a parallel composition of α_δ for all $\delta \in \Delta^\eta(X_i)$. Obviously, $\Delta^{\alpha_i}(X_i) = \Delta^\eta(X_i)$.

Take a look at $h(t^1, \ldots, t^n) := t^0$, bearing in mind that $[\![t^j]\!]_{X_i} = u_j$ for $j = 0, \ldots, n$. In particular, t^j must be defined (not $\perp$) in X_i. Since the only defined locations of X_i are those of $G \cup \{f_i\}$, we may conclude that t^j are all terms over $G \cup \{f_i\}$, not referring at all to values in the child cell. And since all defined locations of $\eta(X_i)$ are over $G \cup \{f_i, f_k\}$, we may conclude that $h \in G \cup \{f_i, f_k\}$. Accordingly, we partition α_i into two parallel assignment commands: a_i are all those commands with $h \in G \cup \{f_i\}$ and n_i are commands with $h = f_k$. Obviously, $\alpha_i = [\, a_i \parallel n_i \,]$. We may call this the *characteristic assignment* of X_i.

Let $a^\sharp$ be obtained from a_i by replacing f_i with f. Then $a^\sharp$ is an assignment command over the templates T. From the definition of parallel assignment, we obtain that $\Delta^{a^\sharp}(X_i) = \Delta^{a_i}(X_i)$. Let $n^\sharp$ be obtained from n_i by replacing f_i and f_k with f. From the definitions of parallel creation and of comparing updates for different cells, we obtain that $\Delta^{\nu.n^\sharp}(X_i) = \Delta^{n_i}(X_i)$. Define the program $\alpha^\sharp = [\, a^\sharp \parallel \nu.n^\sharp \,]$. Then $\Delta^{\alpha^\sharp}(X_i) = \Delta^{a^\sharp}(X_i) \cup \Delta^{\nu.n^\sharp}(X_i) = \Delta^{\alpha_i}(X_i)$.

Lemma 9. *Let X_i be a local state of a parallel algorithm $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \eta, \lambda \rangle$ and α_i the characteristic assignment for X_i under η. Then $\alpha^\sharp(X_i) = \alpha_i(X_i) = \eta(X_i)$.*

Proof. That $\alpha^\sharp(X_i)$ is $\eta(X_i)$ follows from the above discussion. That $\alpha_i(X_i)$ is $\eta(X_i)$ follows from [32, Lemma 11]. □

Updates of local states depend on the values of templates only.

Lemma 10. *Let X_i be a local state of a parallel algorithm $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \eta, \lambda \rangle$ and α_i the characteristic assignment for X_i under η. If Y_i is a local state with the same identifier i and $X_i \equiv_T Y_i$, then $\alpha^\sharp(Y_i) = \alpha_i(Y_i) = \eta(Y_i)$.*

Proof. Since $\alpha^\sharp$ is a command over T it will contain updates based on the values of T only. Considering that $X_i \equiv_T Y_i$, we will have $\Delta^{\alpha^\sharp}(X_i) = \Delta^{\alpha^\sharp}(Y_i)$. It follows from the previous lemma that $\alpha^\sharp(X_i) = \eta(X_i)$. According to the **locality** postulate, we have $\Delta^\eta(Y_i) = \Delta^\eta(X_i)$, again since $X_i \equiv_T Y_i$. Combining all together, we conclude that $\alpha^\sharp(Y_i) = \eta(Y_i)$, as claimed. □

Every local state X_i induces an equivalence relation $\sim_{X_i}$ on templates T according to which $s \sim_{X_i} t$ iff $[\![s_i]\!]_{X_i} = [\![t_i]\!]_{X_i}$. We show next that update commands for i-states X_i are determined by this relation.

Lemma 11. *Let X_i be an i-state of a parallel algorithm $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \eta, \lambda \rangle$, Y_j a j-state, and α_j the characteristic assignment for Y_j under η. If $\sim_{X_i} = \sim_{Y_j}$, then $\alpha^\sharp(X_i) = \alpha_j(X_i) = \eta(X_i)$.*

Proof. We may treat X_i and Y_j as ordinary states over finite vocabularies, as we did earlier in this section. We are given that $[\![s_i]\!]_{X_i} = [\![t_i]\!]_{X_i}$ iff $[\![s_j]\!]_{Y_j} = [\![t_j]\!]_{Y_j}$ for any templates $s, t \in T$. By [32, Lemma 13], we get $\alpha_i(X_i) = \eta(X_i)$, thanks to **abstractness** and **algorithmicity**. By Lemma 9, we may conclude that $\alpha^\sharp(X_i) = \eta(X_i)$. Recall that we consider states to be equal if they are equal up to a permutation of identifiers. □

We are ready to prove that any parallel algorithm may be described by a parallel program.

Theorem 12 (Representation)**.** *For each parallel algorithm, there exists a characteristic parallel abstract machine.*

Proof. For any equivalence relation $\sim$ on templates T, we define the guard $c_\sim$ to be the conjunction of equalities $s = t$ for all $s, t \in T$ such that $s \sim t$, plus the conjunction of disequalities $s \neq t$ for all $s, t \in T$ such that $s \not\sim t$. For each possible relation $\sim$, we choose a local state X_i of the algorithm with the relation $\sim$ between its instantiated templates T_i (provided there is such a state), and call it $X_\sim$. Then we can define the program $R_\sim = \textbf{if } c_\sim \textbf{ then } \alpha^\sharp_\sim$, where $\alpha_\sim$ is the characteristic assignment for $X_\sim$. Obviously $c_\sim$ evaluates to $\textsc{t}$ on $X_\sim$, and hence $R_\sim(X_\sim) = \alpha^\sharp_\sim(X_\sim)$.

Let P be the PAM consisting of commands $R_\sim$ for all possible equivalence relations $\sim$ of T for which there is at least one state $X_\sim$. Since T is finite (by **locality**), it has only finitely many distinct equivalence relations, and so program P is finite. We claim that P is a characteristic program of the algorithm, that is, $P(X) = \eta(X)$ for any state X.

Consider some local state X_i satisfying the relation $\sim_i$ on templates. By Lemma 11, $\alpha_{\sim_i}(X_i) = \eta(X_i)$. Exactly one guard in P applies to X_i and that is $c_\sim$. So $P(X_i) = P_{\sim_i}(X_i) = \alpha^\sharp_{\sim_i}(X_i) = \alpha_{\sim_i}(X_i) = \eta(X_i)$.

Assume finally that X is a general state of the algorithm. By Proposition 4, the update of X is the union of updates of all its localizations X_i. By the **abstractness** axiom, X_i is also a state. According to **locality**, updates for X_i do not depend on whether X_i is considered as a standalone state or a localization of a general state. So it is enough to show that $\Delta^P(X_i) = \Delta^\eta(X_i)$ for all $i \in \mathcal{I}$, which was just established in the previous paragraph. □

6 Effective Parallel Algorithms

For an algorithm to be effective, it must be possible, not only to describe transitions finitely via templates, but also to fully describe its initial states, that starting subset of the algorithm's states containing input values. Included in that description are the operations with which those initial states are endowed. Only a global state that can be fully described finitely and operationally can be considered effective.

6.1 Effective States

In general, an algorithm's domain might be uncountable—as in Gaussian elimination over the reals, but, when we speak of "effective" algorithms, we are only interested in that countable part of the domain that can be described effectively. Thus, we may as well restrict our discussion to countable domains and assume that every domain element can be described by a term in the algebra of the states of the algorithm. Furthermore, a state's operations normally involves an infinite table lookup. Thus, the initial state of an algorithm may contain ineffective infinite information, in which case the algorithm could not be deemed effective. So we need to place finiteness restrictions on the initial states of algorithms. Another problem is that the same domain element might be accessible via several terms, generating nontrivial relations, which might hide noncomputable information.

Constructors provide a way of giving a unique name to each domain element. The domain can be identified with the Herbrand universe (free term algebra) over constructors. Destructors provide an inverse operation for constructors. For every constructor c of arity n, we may have destructors $c^1, \ldots, c^n$ to extract each of its arguments $[c^j(c(x_1, \ldots, x_j, \ldots, x_n)) = x]$, plus c^0, which returns an indicator that the root constructor (of a value) is c. Constructors and destructors are the usual way of thinking of domain values of effective computational models. For example, strings over an alphabet {a,b,...} are constructed from a scalar constructor $\varepsilon()$ and unary constructors a$(\cdot)$, b$(\cdot)$, while destructors may read and remove the last letter. Natural numbers in unary (tally) notation are normally constructed from (unary) successor and (scalar) zero, with predecessor as destructor. The positive integers in binary notation are constructed out of (the scalar) ε and (unary) digits 0 and 1, with the constructed string understood as the binary number obtained by prepending the digit 1. The destructors are the last-digit and but-last-digit operations. For Lisp's nested lists (s-expressions): the constructors are (scalar) `nil` (the empty list) and (binary) `cons` (which adds an element to the head of a list); the destructors are `car` (first element of list) and `cdr` (rest of list). To construct 0-1-2 trees, we would have three constructors, $A()$, $B(\cdot)$, and $C(\cdot,\cdot)$, for nodes of out-degree 0 (leaves), 1 (unary), and 2 (binary), respectively. Destructors may choose a child subtree, and also return the degree of the last-added (root) node.

Definition 13 (Effective State [4])**.** A state is *effective* if its domain is isomorphic to a free constructor algebra and its operations all fall into one of the following categories:

1. those free constructors and their corresponding destructors and equality;
2. infinitely-defined operations that can themselves be computed effectively with those same constructors (perhaps using a richer vocabulary); or
3. finitely-many other defined locations, not having the default value, $\perp$.

If there are no infinitely-defined operations (item 2), then the state is deemed *basic*.[12]

Initial states may include constructor operations (item 1 in the above definition) which are certainly effective, as they are just a naming convention for domain values.

Without loss of effectiveness, we can allow any finite amount of nontrivial data (item 3) in initial states, provided that all initial states are provided with the same data. We assume that domains include two distinct truth values, and—furthermore—that we have (scalar) constructors, T and F, for them. Boolean operations are effective finite tables, so we may presume them, too. Initial states need also to contain a finite amount of input, which we discuss below.

Moreover, initial states can be endowed with known effective operations (item 2). The circularity of this definition of effectiveness bottoms-out with operations that are programmable directly from the constructors. Given free constructors, equality of domain values and destructors are also effective (see [4]). We are presuming that constructors are present in states—even if an algorithm avoids their direct use.

Remark. Two other ways of capturing the notion that (initial) states have a finite description, thereby characterizing effectiveness, have been suggested. One alternative [33] characterizes an effective (initial) state as one for which there is a (semi-) decision procedure for equality of terms in the state. That is, there is Turing machine for determining whether a state interprets two terms (given as strings) as the same domain value. A second alternative [18] requires that there exist an (arbitrary) injection from the chosen domain of the algorithm into the natural numbers such that the given base operations (in initial states) are all tracked (under that injection) by partial-recursive functions. This way, an algorithm is effective if there is an injection $\rho_D : D \to \mathbb{N}$ for each domain D of its states, such that the (partial) function $\rho_D(f) : \mathbb{N} \rightharpoonup \mathbb{N}$ is (partial) recursive for every operation f of its initial states.

In contrast to Definition 13, these two alternatives are somewhat circular: the first relies on Turing-machine computability and the second, on recursive functions. All the same, all three characterizations of effectiveness have been shown to lead to one and the same class of effective functions (up to isomorphism) for any sequential computational model of computation over any domain [5]. □

6.2 Effective Algorithms

Obviously, an initial state should also be allowed to include some input, which will differ from initial state to initial state. To handle inputs, we postulate some subset of the templates, named *input terms*, which can contain any possible combination of domain values, and such that all initial states agree on all terms over the vocabulary of the algorithm except for these.

[12] Though item (3) is a special case of the previous item, we want to keep them separate and allow basic states to have the third type, but not the second.

To preclude ineffective information being included in the initial global setup, the number of cells that are active at the outset must be finite and input independent; alternatively, there must be some effective way of setting up the initial configuration from the input values alone.

Postulate VI (Effectiveness)**.** A parallel algorithm is *effective* if it is of one of these two types:

1. its initial global states consist of all states containing exactly one nondormant cell that is in an effective state, and all initial states over the same domain are identical except for the values of input terms; or
2. its initial global states are the terminal states of an effective parallel algorithm of the previous type.

Finiteness of templates, together with commutativity with isomorphism, guarantees that only finitely many locations in a cell can be affected by one transition [26, Lemma 6.1]. That assures that, whenever a state is effective, so is the next state (when there is a next state). This justifies our definition of an effective *algorithm* as having effective *initial* states.

Example 1 (Continued). We can approximate our stopwatch example of Section 2 with an effective algorithm by using the constructible rationals $\mathbb{Q}$ as domain, instead of the reals, letting π be some rational approximation, and starting out with only some finite number of points spread out evenly along the initial line from $(0, 2\pi)$ to $(1, 2\pi)$. In point of fact, Figure 1 was drawn with 30 segments along the minute hand, using four decimal places for arithmetic. □

6.3 Basic Algorithms

When measuring time complexity, we will want to charge more than unit cost for complex, programmed operations, like multiplication or factorial. To capture this distinction between basic unit-cost operations and complex ones, we take advantage of the fact that every effective algorithm can have its defined operations "in-lined," yielding a basic algorithm, whose steps we will count instead.

Definition 14 (Basic Algorithm)**.** An effective parallel algorithm is *basic* if its initial global states contain exactly one basic nondormant cell (having no infinitely-defined operations) or if they are the terminal states of such a basic algorithm.

Infinitely-defined operations were allowed by item (2) of Definition 13, but are disallowed for basic states.

Basic algorithms are clearly effective, since they operate over finite data only. But they are not expressive enough to emulate all effective functions step-for-step, since the latter may have direct access to complex non-unit-cost operations. For that reason, we allowed an effective state to be equipped with "effective oracle operations," which can be obtained by bootstrapping from a basic algorithm with only constructors.

Definition 15. A basic parallel algorithm is in *normal form* if all its operations (global and local) are nullary or unary, except for one binary constructor.

Lemma 16. *Any effective (basic) parallel algorithm may be emulated by an effective (basic) parallel algorithm in normal form.*

Proof. This argument, albeit for a different case of abstract state machines, was suggested in [19]. The idea is that an operation with n arguments can be considered as an operation with one argument, an n-tuple constructed by $n-1$ applications of pairing. So the emulating algorithm will have a pairing function as one of its constructors. The other operations will have the same names as in the original one, except that operations that were of arity greater than 1 will now have arity 1: instead of appealing to $f(u_1, \ldots, u_n)$, it will appeal to $f(\langle u_1, \langle u_2, \langle u_3, \ldots \langle u_{n-1}, u_n\rangle \ldots\rangle\rangle\rangle)$. Since the vocabulary of the algorithm is finite, this can be done during the same transition. □

7 Measuring Complexity

The prevalent convention measures (asymptotic) complexity as the (maximum) number of operations relative to input size. As we want to count atomic operations, not arbitrarily complex operations, we should count constructor operations. So we have a choice: to count all the operations executed by an effective algorithm, or to count the transition steps of its corresponding basic algorithm. We take the latter route. To measure the time needed for the execution of a basic algorithm, we use—for the time being—the "uniform measure" [35, pp. 10–11], under which every transition is counted as a one time unit. Later, we will address the question of what cost to charge for each transition step.

To handle arbitrary data types, the sensible and honest way is to define the size of a domain element to be the number of basic operations required to build it. See [17, 6].

Definition 17 (Size)**.** The *size* of a domain element is the minimal number of constructor operations required to name that value.

The size $|n|$ of a unary number n, represented as $s^n(0)$, is $n+1$. The size of n in binary is $\lceil \lg n \rceil$; for example, $|5| = 3$, the length of $0(1(\varepsilon))$, the initial 1(for the string 101) being understood. The size of Turing-machine strings is (one more than) the length of its tape, since string constructors are unary (see the basic Turing-machine implementation in [4]). The size of the tree

$$C(B(A(), A()), B(A(), A()), B(A(), A()))$$

is only 3, because subtrees can be reused, and the whole tree can be specified by

$$C(s, s, s) \textbf{ where } s = B(r, r),\ r = A()$$

An effective algorithm is allowed to access effective oracles (e.g. multiplication) in its initial states, which however are required to be programmable (i.e.

algorithmically describable) by a basic algorithm, that is, using constructors and destructors only (usually with a larger vocabulary). In other words, by bootstrapping an effective algorithm, we get a basic one, which is the right one to consider for measuring complexity.

Definition 18 (Complexity)**.** We measure the (time) complexity of an effective algorithm by the number of basic steps (comprising a bounded number of constructor, destructor, and equality operations) required to perform the computation from initial to terminal states, relative to the input size.

In other words, we inline effective sub-algorithms to get a basic one and measure the complexity of the latter. Since any single step comprises a bounded number of operations, counting steps gives the same order of magnitude as counting individual operations.

Example 2. Consider an effective algorithm `rev` to reverse the top-level elements of a Lisp-like list. The domain consists of all nested lists; that is, either an empty list $\langle\rangle$, or else a nonempty list of lists: $\langle\langle\rangle\rangle$, $\langle\langle\rangle\langle\rangle\rangle$, $\langle\langle\rangle\langle\rangle\langle\rangle\rangle$, ..., $\langle\langle\langle\rangle\rangle\rangle$, $\langle\langle\langle\rangle\rangle\langle\rangle\rangle$, ..., $\langle\langle\rangle\langle\langle\rangle\langle\rangle\rangle\rangle$, ... The function `rev`: $\mathcal{L} \to \mathcal{L}$ takes a list $\langle l_1 \dots l_n \rangle$ and returns $\langle l_n \dots l_1 \rangle$, with the sublists l_j unchanged. For instance, $\texttt{rev}(\langle\langle\rangle\langle\rangle\langle\rangle\rangle\rangle) = \langle\langle\rangle\langle\rangle\rangle\langle\rangle\rangle$.

Now, `rev` could be a built-in operation of the Lisp model of computation, which in one fell swoop reverses any list. Clearly, constant cost for `rev` is not what is intended; we want to count the number of basic list operations (the constructor `cons`, destructors `car` and `cdr`, and tests $=$ and $\neq$) needed to reverse a list ℓ of length n. So there is no escape but to take into account how `rev` is implemented internally.

Suppose $\texttt{rev}(\ell)$ works something like this:

> **if** $y = \bot$ **then** [$y := \ell \parallel z := \texttt{nil}$]
> **if** $y \neq \bot$ **&** $y \neq \texttt{nil}$ **then** [$z := \texttt{cons}(\texttt{car}(y), z) \parallel y := \texttt{cdr}(y)$]
> **if** $y = \texttt{nil}$ **then** *answer* $:= z$

The first line initializes the scalar y to the input list ℓ; the second line is iterated to build z element by element; the last line puts the result in location *answer*.

The number of operations executed by this implementation is less than $10(n + 2)$, because the maximum number of list and other (Boolean, assignment) operations in a single iteration is 10 and there are always $n+2$ iterations. So, any application of `rev` in an algorithm devolves into $O(n)$ basic operations.

Note that any straightforward (even multi-tape) Turing machine would require arbitrarily more steps, in general, proportional to the overall *size* of the input ℓ, which depends on the sizes of all of the individual elements in ℓ, rather than just on the number n of elements at its top level, as is the case with the above list-based algorithm. The length of $\langle\langle\rangle\langle\langle\rangle\langle\rangle\rangle\rangle$ is 2, but its size is 6.

In any RAM implementation, each list is represented by some natural number; what encoding is chosen is immaterial, as long as all operations perform consistently. Regardless of what number is used to represent the list $l = \langle\langle\rangle\langle\langle\rangle\langle\rangle\rangle\rangle$, $\texttt{car}(\texttt{rev}(\texttt{car}(\texttt{rev}(l))))$ should return the number that represents $\langle\langle\rangle\rangle$. □

8 Parallel Random Access Machines

For the definition of RAMs, we take the set of instructions suggested by Cook and Reckhow in [12] and use the classification of RAM machines by van Emde Boas in [35]. RAMs operate over the natural numbers $\mathbb{N}$.

8.1 RAMs

A random access machine has a fixed number of registers, which we refer to as `X`, `Y`, etc., plus a monolithic "random access" memory, indexed by natural numbers.

For *basic RAMs*, the following operations are considered to each take "unit" time:

1. $\texttt{X} \leftarrow \texttt{C}$ places the constant `C` in register `X`;
2. $\texttt{X} \leftarrow [\texttt{Y}]$ loads register `X` with $[\texttt{Y}]$, the contents of the memory location indexed by `Y`;
3. $[\texttt{Y}] \leftarrow \texttt{X}$ stores the register value in memory location $[\texttt{Y}]$;
4. `TRA m if X` > 0 transfers control to the `m`-th line of the program if $\texttt{X} > 0$;
5. `READ X` puts the next input value in `X`;
6. `PRINT X` prints the number in `X` on the output tape.

Successor RAMs are an extension of basic RAMs with successor/predecessor operations:

7. `INC X` increments the value of register `X` by 1;
8. `DEC X` decrements `X` by 1.

Arithmetic RAMs are the model originally defined by Cook and Reckhow in [12]; they extend basic RAMs with addition and subtraction:

7. $\texttt{X} \leftarrow \texttt{Y} + \texttt{Z}$;
8. $\texttt{X} \leftarrow \texttt{Y} - \texttt{Z}$.

Multiplication RAMs extend Arithmetic RAMs with multiplication and division:

9. $\texttt{X} \leftarrow \texttt{Y} \times \texttt{Z}$;
10. $\texttt{X} \leftarrow \texttt{Y} \div \texttt{Z}$.

A *multidimensional RAM* operates with a multidimensional memory, rather than the classical one-dimensional array. Thus a memory address is given by a tuple of natural numbers:

11. $\texttt{X} \leftarrow [\texttt{Y}, ..., \texttt{Z}]$;
12. $[\texttt{Y}, ..., \texttt{Z}] \leftarrow \texttt{X}$.

8.2 PRAMs

A parallel RAM (PRAM) consists of some number of independent sequential processors, each with its own private memory and communicating with one another via a shared global memory. In one unit of time, each processor can execute a single RAM operation and write to one global or local memory location. All processors execute the same RAM program. PRAMs are classified by the kind of unit-time operations they are equipped with. For example, in one step of a basic PRAM, each process can execute one basic RAM instruction. The same holds for arithmetic and multiplication PRAMs. In addition to this, each process may create a child process, using the FORK command. The child process will run the same program as her parent and will receive from her parent the label of the "first command to execute."

We use FORK in the way it was pioneered in [23]:

13. `fork` *label*. Create a child process that begins execution from command number *label*. The contents of one fixed register are copied from parent to child.

A *multidimensional PRAM* is a PRAM that has multidimensional memories, both global and local.

Another important classification of parallel machines is by restrictions on shared-memory access. In a single step of a PRAM, each process can access a location in shared memory for either reading or writing. And each type of access can be either exclusive (only one process is granted access) or common (multiple process access) under some restriction. The *exclusive read/write* restriction prevents reading from/writing to the same global memory location simultaneously by two distinct processors. These options are denoted R(ead), W(rite), E(xclusive), and C(ommon). So *CREW PRAM* stands for ***C**ommon-**R**ead **E**xclusive-**W**rite **P**arallel **R**andom-**A**ccess-**M**achine*. In this model, any process may read any shared memory location at any step. But, in a single step, a memory location may be written to by at most one process.

A Common-Write machine should include in its description a policy for conflict resolution—for the case when multiple processors request a write to the same global memory location. Some commonly used methods are:

- the COMMON model—all processors writing to the same location are required to write the same value;
- the ARBITRARY model—any process participating in common write may succeed and the algorithm is obliged to work correctly regardless of which is the winner; and
- the PRIORITY model—there is a linear order of processors and the one with highest priority is the one that succeeds.

The above PRAM models do not differ much in computational power. A PRIORITY PRAM (the strongest) can be simulated by an EREW PRAM (the

weakest) with the same number P of processors and with only order $\log P$ time overhead [36, 28].

8.3 PRAMs are Parallel Algorithms

To see how PRAMs meet the requirements we laid out for parallel algorithms, we need to understand what the states would look like from the point of view of our postulates. The domain of the states of a PRAM algorithm is the natural numbers. The states are all endowed with the arithmetic capabilities of PRAMs and the associated vocabulary. The global PRAM memory resides in a global operation from our point of view; the local memories are local; the registers are global or local, as the case may be. The templates are the various registers and expressions appearing in the PRAM program. Forking, however, requires copying all local information to the global area, creating a new cell, and then copying the local information into its proper place.

The effective parallel algorithm corresponding to a PRAM has to first create some input-dependent number of cells and supply them each with local data, per option (2) of Postulate VI. The individual cells can also be assigned an identifier—as a number—by the parent when created. Only after setting up such an initial state, from the PRAM's point of view, would one start running the PRAM program proper.

9 Simulation of Parallel Algorithms

To reach our main conclusion, namely, the Parallel Computation Thesis (Theorem 23), we compose several polynomial simulation steps:

1. Every basic parallel algorithm can be emulated by a basic PAM machine (Lemma 12).
2. For every basic PAM, there is one in normal form (Lemma 16).
3. Every basic normal PAM may be simulated by a multidimensional Successor PRAM endowed with an encoding operation σ to represent domain elements as numbers (Lemma 19).
4. There is a suitable encoding σ that can be computed by a Multiplication PRAM (Lemma 20).
5. Every Multiplication PRAM can be simulated by a Common Arithmetic PRAM (Lemma 21).
6. Every Common Arithmetic PRAM can be simulated by an Exclusive Arithmetic PRAM (Lemma 22).

The first two steps have already been taken. The third step is next:

Lemma 19. *Any effective parallel algorithm in normal form can be simulated by a three-dimensional Successor Common PRAM with oracle access to some injection $\sigma : \mathbb{N}^3 \to \mathbb{N}$ and with word size big enough to accommodate single-step processing of desired σ values. The overhead in running time is some constant (multiplicative) factor that depends on the simulated algorithm. The number of required processors is equal to the number of cells.*

Such a σ will be suggested shortly (in the proof of Lemma 20).

Proof. Let $\mathcal{A}$ be an effective parallel algorithm with global operations $G = \{g^1, \ldots\}$ and local operations $F = \{f^1, \ldots\}$. Let X be a state of $\mathcal{A}$, D be the domain of X, and $C = \{c^1, \ldots, c^\ell\} \subseteq G$ be the constructors of D in some order. Recall that we identify D with a free term algebra over C.

The proof proceeds in four steps: (1) the representation of domain elements of $\mathcal{A}$ as numbers; (2) the representation of operations of $\mathcal{A}$ as arrays of numbers; (3) the representation of states of $\mathcal{A}$ as states of a PRAM; (4) the implementation of transitions of $\mathcal{A}$ as transitions of the PRAM.

(1) *Domain simulation.* We first define injections $\tau : D \to \mathbb{N}^3$ and $\rho : D \to \mathbb{N}$ in the following mutually recursive way:

- $\tau : \bot \mapsto \langle 0, 0, 0 \rangle$;
- $\tau : c^j() \mapsto \langle j, 0, 0 \rangle$, when c^j is a nullary constructor (i.e. a scalar constant);
- $\tau : c^j(u) \mapsto \langle j, \rho(u), 0 \rangle$, when c^j is a unary constructor and $u \in D$ is any domain element;
- $\tau : c^j(u, v) \mapsto \langle j, \rho(u), \rho(v) \rangle$, when c^j is the unique binary constructor of the normal form and $u, v \in D$ are any domain elements;
- $\rho : u \mapsto \sigma(\tau(u))$, where $\sigma : \mathbb{N}^3 \to \mathbb{N}$ is given.

The function τ is an injection since we identified D with a free term algebra; ρ is an injection since it is the composition of two injections.

(2) *Algebra simulation.* We describe next a multidimensional PRAM state X^P that corresponds to a PAM state X via domain injection $\rho : D \to \mathbb{N}$. State X^P includes the following:

- a number of processors equal to the number of cells in X;
- a three-dimensional shared memory, G, to maintain values $g^j(\cdot, \cdot)$ in $G[j, \cdot, \cdot]$;
- a two-dimensional local memory F for each processor i to maintain values $f^j(\cdot)$ in $F[j, \cdot]$.

To each local cell X_i in X we allocate one processor, which we refer to as p_i. In the shared memory G of X^P, we store global values of X. The local memory F of each processor p_i stores local values for the local state X_i. The isomorphism between PAM states and PRAM memories is as follows:

- $G[j, 0, 0] = \rho([\![g^j]\!]_X)$ if g^j is a global scalar.
- $G[j, \rho(u), 0] = \rho([\![g^j]\!]_X(u))$, for each $u \in D$, if $g^j(\cdot)$ is a global (unary) operation.
- $G[j, \rho(u), \rho(v)] = \rho([\![g^j]\!]_X(u, v))$, for all $u, v \in D$, for the (unique) binary constructor $g^j(\cdot, \cdot)$.
- $F[j, 0] = \rho([\![f^j]\!]_{X_i})$ in processor p_i if f^j is a local scalar.
- $F[j, \rho(u)] = \rho([\![f^j]\!]_{X_i}(u))$ in processor p_i, for each $u \in D$, if $f^j(\cdot)$ is a local (unary) operation.
- All other entries of shared and local memories are 0.

(3) *State simulation.* The only information that X^P is missing to simulate X are the values of the templates T. For convenience, we take T to be its closure under the subterm relation, so each local state X_i has readily available the local values of its critical terms T_i and all of their subterms. Hence, each processor p_i should keep a pointer for each of the values of terms in T_i. To this end, for each term $t_i \in T_i$, the process p_i will store a scalar, which we will refer to as t, in its local memory. And if $[\![t_i]\!]_{X_i} = u$ then p_i should store $\rho(u)$ as the value of its local t. With this information, X^P simulates X via ρ.

(4) *Transition simulation.* Finally, we show that there exists a program P for a multidimensional PRAM with oracle access to σ such that if $\eta(X) = Y$ then $P(X^P) = Y^P$, where η is the transition function of algorithm $\mathcal{A}$.

Let X be a state of $\mathcal{A}$. A transition from X may be viewed as the union of the transitions of all local states X_i of X, by Proposition 4. By **locality**, updates of X_i are the same, whether it is a global state with just one cell or a local cell of a bigger state, that is, $\eta(X) = \bigcup_i \eta(X_i)$. Hence, it is enough to provide a program P such that $P(X_i) = Y_i$ for all $i \in \mathcal{I}$. More generally, it is enough to prove that $P(X^P) = Y^P$ for any local state X.

So let X be any *localized* state, and Y be the next state, $\eta(X)$. Let X^P be a PRAM state simulating X, as described above. Then X^P has only one processor. Let T be the templates of $\mathcal{A}$, and let P be a characteristic parallel program of $\mathcal{A}$, as described in Theorem 12. For each transition, P performs a bounded number of basic operations on critical terms: comparisons, assignments, and creation commands.

We explain now how a PRAM can simulate each basic operation:

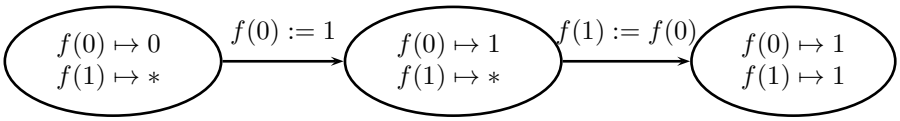
- Basic comparison operations ask to compare the values of two critical terms. Since, as we assumed, X^P has those values in special local constants, the unique processor should only compare the values of those two constants. This is done in one single operation, since we assume that a processor may perform any memory access in one step.

- A basic assignment command $h(s) := t$ applied to state X creates one update $h([\![s]\!]_X) \mapsto [\![t]\!]_X$. We assumed that T is closed under the subterm relation; hence X^P records local values for terms s and t as well.

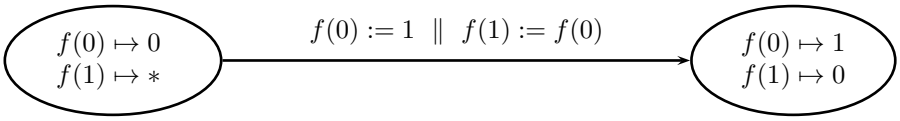
  If h is some global operation g^j, an assignment is simulated by a shared-memory write command: $G[j, \rho([\![s]\!]_X), 0] \leftarrow \rho([\![t]\!]_X)$. If h is some local operation f^j, the assignment is simulated by a local memory write command: $F[j, \rho([\![s]\!]_X)] \leftarrow \rho([\![t]\!]_X)$. This again can be done in one operation, since we assumed that a processor may operate on any memory location in a single step.

- The **new** command is simulated by the FORK command of PRAMs. Initial information that a mother passes to her child should be placed by the mother in shared memory. (This requires some bookkeeping, since there are many processes working in parallel.) A mother should wait for the child to copy this information to its local memory and then she should clean up. Only after that may she move on to the next step. Since, per **motherhood**, a mother may perform only a bounded number of operations for her children, the number of steps required to complete this task is also uniformly bounded. A mother can be programmed to "sleep" (repeating increment/decrement instructions) while its child copies the data. So this action may be simulated in a constant (algorithm-dependent) number of PRAM steps.

Note that a PAM performs a multiple number of assignments in one step. This is not equivalent to sequentially performing the same assignment statements (like a PRAM does). As an example, assume that we have the local value $f(0) = 0$. Consider two assignment statements: $f(0) := 1$ and $f(1) := f(0)$. Sequential application will result in:



whereas simultaneous application results instead in:



To avoid such problems, before a PRAM starts to construct the updates of X^P, it must make a copy of all critical-term values in X^P and then refer to those stored values.

Algorithm 1 The parallel RAM simulates one step of a basic parallel program.

1. Create a local copy of all critical term values.
2. Perform all assignment commands.
   - Stay put for exactly m operations (sleep if required).
3. Create initial information for children in (a specified place in) shared memory.
   - Stay put for exactly $n \cdot d$ operations (sleep if required).
4. FORK the required number of times and wait for children to update their initial information.
   - Stay put for exactly n operations (sleep if required).
5. Clean children's information from shared memory.
   - Stay put for exactly $n \cdot d$ operations (sleep if required).
6. Update critical term values for the next step.

By **algorithmicity**, only a bounded number of assignments may be executed in one step. Assume that this bound for our algorithm is m. According to **fertility**, only a bounded number of children can be born by one mother in one step. Assume that this bound for our algorithm is n. In addition, according to **motherhood**, only a bounded amount of data can be passed from mother to child. Assume that this bound for our algorithm is d.

So to simulate one transition of a PAM (and thus of a parallel algorithm), a PRAM process should do as described in Algorithm 1. Sleep pauses are inserted to synchronize the actions of distinct processes. □

Lemma 20. *Any basic parallel algorithm can be simulated by a Multiplication Common PRAM with only constant factor increase in running time and with the same number of processors, with the PRAM operating on words of logarithmic size.*

Proof. To prove that a Multiplication Common PRAM may simulate a basic parallel program in normal form, according to Lemma 19, we only have to show that we can compute some bijection $\sigma : \mathbb{N}^3 \to \mathbb{N}$, preserving the logarithmic size. This may, for instance, be accomplished by the Cantor pairing function:

$$\begin{aligned} \sigma'(x, y) &= \tfrac{1}{2}(x + y)(x + y + 1) + y \\ \sigma(x, y, z) &= \sigma'(\sigma'(x, y), z) \end{aligned}$$

which may be computed using a bounded number of arithmetic operations (multiplication, addition, and halving).

Now we need only map the three-dimensional memory to a single dimension. And that too can be accomplished via σ. □

n.b. There being no prevailing notation for the verbose expression $\log\log x$ (or $\log\log\log x$, etc.), we propose to use $\operatorname{logg} x$ (and $\operatorname{loggg} x$, resp.) in what follows.[13]

Lemma 21. *Any Multiplication Common PRAM with time complexity $T(n)$ and with $P(n)$ processors can be simulated be an Arithmetic Common PRAM in order*

$$T(n) \cdot \log T'(n)$$

time and with order

$$P(n) \cdot T'(n) \cdot \log T'(n) \cdot \operatorname{logg} T'(n)$$

processors, where $T'(n) = T(n) + \log n$.

Proof. It was shown in [34] that multiplication of n-bit numbers can be done by circuits of bounded fan-in with depth $O(\log n)$ and number of agents $O(n \cdot \log n \cdot \operatorname{logg} n)$.[14] It was shown in [27] that a bounded fan-in circuit can be transformed into a circuit with bounded fan-in and bounded fan-out with only a constant factor increase in the number of gates and in depth. The latter can be simulated by an Arithmetic EREW PRAM, where gates are simulated by processes and time is equivalent to depth. Obviously, an Arithmetic EREW PRAM may be considered as a special case of an Arithmetic Common PRAM.

Combining the above, an Arithmetic Common PRAM may perform a multiplication of n-bit numbers with an extra $O(n \cdot \log n \cdot \operatorname{logg} n)$ processes and in $O(\log n)$ time.

In one single step, an Arithmetic PRAM can at most double the maximum number it already has in its memory. So starting with input n, the maximal value it may attain over $T(n)$ steps is $n2^{T(n)}$, which can be stored in memory using $\log_2(n2^{T(n)}) = T'(n)$ bits. According to the above, multiplication of numbers with $T'(n)$ bits can be done in $O(\log T'(n))$. To do so, each process may require order $T''(n) = T'(n) \cdot \log T'(n) \cdot \operatorname{logg} T'(n)$ extra processes. The total number of processors used will be order $P(n) \cdot T''(n)$.

Recall that we are simulating a multiplication PRAM. Hence a processor that desires to perform multiplication will have to create its helpers by itself. Thus, it will have to invoke FORK $O(T''(n))$ times. And then each of those helpers performs multiplications in $O(\log T'(n))$ steps. As we may FORK from child processes also, until we have enough processes, creating k processes requires $\log k$ steps, so order $T''(n)$ processes need $O(\log T''(n)) = O(\log T'(n))$ steps. It follows that the overall time for one multiplication is still order $\log T'(n)$. □

[13] Cf. the suggested use of l_2, l_3, etc. in [37].

[14] The construction is logspace uniform, that is, there exists a Turing machine that, on input of size n, generates in logspace a program executed by each processor.

Lemma 22. *Any basic algorithm with time complexity $T(n)$ and with $P(n)$ processors can be simulated by an Arithmetic EREW PRAM in order*

$$T(n) \cdot \log T'(n) \cdot \big(\log P(n) + \log T'(n)\big)^2$$

time and with order

$$P(n) \cdot T'(n) \cdot \log T'(n) \cdot \operatorname{logg} T'(n)$$

processors, where $T'(n) = T(n) + \log n$.

Proof. An Arithmetic Common PRAM can be simulated by an Arithmetic EREW PRAM of the same type with a time-overhead factor that is logarithmic squared in the number of processors [21, 36]. Applying this to the combined result of Lemmas 20 and 21, we arrive at a processor overhead factor that is order $P'(n) = T'(n) \cdot \log T'(n) \cdot \operatorname{logg} T'(n)$, as in Lemma 21, and an additional factor of $\log^2(P(n) \cdot P'(n)) = O((\log P(n) + \log T'(n))^2)$ in time overhead. □

With the previous lemma in place, we have finally arrived at the formal substantiation of the Parallel Computation Thesis:

Theorem 23 (Parallel Computation)**.** *Polynomial time for basic (effective) parallel algorithms—with a number of cells that is no more than exponential in running time—is equivalent to polynomial space for Turing machines.*

Proof. It was shown in [23, Thm. 1] that polytime PRAM equals PSPACE, provided that $T(n) \geq \log n$ and the number of processors of the PRAM is no more than exponential in (parallel) running time. The theorem follows from this and the previous lemma. □

10 Discussion

The starting point for this research was the desire to characterize parallel computation in as generic a form as possible, with an eye especially towards the effective special case. Blass and Gurevich [2] successfully characterized parallel algorithms within the abstract-state-machine framework, but their approach is not easily restricted to be effective. In particular, in their setup, an unbounded number of children may be created by a single cell in a single step.

Analogous to prior work on effectiveness for classical algorithms [18, 5], we have characterized what makes a parallel algorithm effective, demanding that the initial global state be effectively describable. This decomposes into two main requirements: (a) that each cell itself be an effective classical algorithm; (b) that the initial setup of cells be producible by an effective algorithm. This formalization allowed us to establish the veracity of an "invariance" thesis for parallel algorithms, as has recently been achieved for classical sequential algorithms [15, 17]: *All effective parallel models of computation can be polynomially*

simulated by a standard model (PRAM). The Parallel Computation Thesis follows: no reasonable parallel model can do more in polynomial time than can a Turing machine with polynomial space.

Our model is simpler than that of Blass and Gurevich for those cases we consider. As we do not have message passing, algorithms need not deal at all with process ids. Though we do bound the number of new cells created by a cell in one step, which makes perfect sense in the effective case, an infinite number of initial cells for a non-effective parallel algorithm poses no problem, as in our stopwatch example of Section 2.

Postulate IV does not allow for a mother to prepare an unbounded quantity of local data in a daughter cell before giving birth; she has access to her daughter's memory for the duration of only one step. The PRAM model is similar in this respect. Were we to allow unbounded preparation (with creation commands $\mathbf{new}_k$, for each child k, that are executed only after all assignments to a child's local state have been completed), then the PRAM would need an unbounded number of steps to copy the data back and forth from global memory. This could double the cost of simulation.

In this work, we have only considered discrete-time systems, where all cells progress in lockstep with each other. This line of work may be expanded in at least two directions:

- *Distributed systems, where cells each progress at their own rate.* This will require a sense of identity for cells and a means of communication between them. To accommodate cells that are aware of one another and can refer to each other, the framework needs to be modified. Cf. [3, 14].
- *Systems that evolve in continuous time.* These require a more complex notion of state evolution. See [8, 14, 9].

References

[1] Andreas Blass and Yuri Gurevich. Ordinary interactive small-step algorithms, Parts I–III. *ACM Transactions on Computational Logic*, vol. 7, pp. 363–419 and vol. 8, articles 15–16, 2006–7. Available at `http://research.microsoft.com/~gurevich/Opera/166.pdf`, `http://research.microsoft.com/~gurevich/opera/170.pdf`, `http://research.microsoft.com/~gurevich/opera/171.pdf` (accessed on January 13, 2016).

[2] Andreas Blass and Yuri Gurevich. Abstract state machines capture parallel algorithms: Correction and extension. *ACM Transactions on Computation Logic*, 9(3):article 19, June 2008. Available at `http://research.microsoft.com/~gurevich/Opera/157-2.pdf` (accessed on January 13, 2016).

[3] Andreas Blass, Yuri Gurevich, Dean Rosenzweig, and Benjamin Rossman. Interactive small-step algorithms, Parts I–II. *Logical Methods in Com-*

puter Science, vol. 3(4), paper 3 and vol. 4(4), paper 4, 2007. Available at `http://research.microsoft.com/~gurevich/Opera/176.pdf`, `http://arxiv.org/pdf/0707.3789v2` (accessed on January 13, 2016).

[4] Udi Boker and Nachum Dershowitz. The Church-Turing thesis over arbitrary domains. In Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich, editors, *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, volume 4800 of *Lecture Notes in Computer Science*, pages 199–229. Springer, Berlin, 2008. Available at `http://nachum.org/papers/ArbitraryDomains.pdf` (accessed on January 13, 2016).

[5] Udi Boker and Nachum Dershowitz. Three paths to effectiveness. In Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig, editors, *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, volume 6300 of *Lecture Notes in Computer Science*, pages 36–47, Berlin, August 2010. Springer. Available at `http://nachum.org/papers/ThreePathsToEffectiveness.pdf` (accessed on January 13, 2016).

[6] Udi Boker and Nachum Dershowitz. Honest computability and complexity. In *Martin Davis on Computability, Computational Logic, and Mathematical Foundations*, Outstanding Contributions to Logic Series. Springer, 2016. To appear. Available at `http://nachum.org/papers/HonestComplexity.pdf` (accessed on July 13, 2015).

[7] Allan Borodin. On relating time and space to size and depth. *SIAM Journal of Computing*, 6:733–744, 1977. Available at `http://www.cs.toronto.edu/~bor/Papers/relating-time-space-size-depth.pdf` (accessed on January 13, 2016).

[8] Olivier Bournez, Nachum Dershowitz, and Evgenia Falkovich. Towards an axiomatization of simple analog algorithms. In Manindra Agrawal, S. Barry Cooper, and Angsheng Li, editors, *Proceedings of the 9th Annual Conference on Theory and Applications of Models of Computation (TAMC 2012, Beijing, China)*, volume 7287 of *Lecture Notes in Computer Science*, pages 525–536, Berlin, May 2012. Springer. Available at `http://nachum.org/papers/SimpleAnalog.pdf` (accessed on January 13, 2016).

[9] Olivier Bournez, Nachum Dershowitz, and Pierre Néron. An axiomatization of analog algorithms. Submitted. Available at `http://nachum.org/papers/AxiomatizationAnalog.pdf` (accessed on January 13, 2016).

[10] Ashok K. Chandra and David Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[11] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. of the Association of Computing Machinery*, 28(1):114–133, 1981.

[12] Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. *Journal of Computer Systems Science*, 7:354–375, 1973. Available at `http://www.cs.utoronto.ca/~sacook/homepage/rams.pdf` (accessed on January 13, 2016).

[13] Nachum Dershowitz. The generic model of computation. In Elham Kashefi, Jean Krivine, and Femke van Raamsdonk, editors, *Proceedings of the Seventh International Workshop on Developments in Computational Models (DCM 2011, July 2011, Zürich, Switzerland)*, Electronic Proceedings in Theoretical Computer Science, pages 59–71, July 2012. Available at `http://arxiv.org/pdf/1208.2585.pdf` (accessed on December 27, 2015).

[14] Nachum Dershowitz. *Res Publica*: The universal model of computation. In Simona Ronchi della Rocca, editor, *Proceedings of the 22nd EACSL Conference on Computer Science Logic (CSL), Torino, Italy*, volume 23 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5–10, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. Available at `http://nachum.org/papers/ResPublica.pdf` (accessed on January 13, 2016).

[15] Nachum Dershowitz and Evgenia Falkovich. A formalization and proof of the Extended Church-Turing Thesis [extended abstract]. In Elham Kashefi, Jean Krivine, and Femke van Raamsdonk, editors, *Proceedings of the Seventh International Workshop on Developments in Computational Models (DCM 2011, July 2011, Zürich, Switzerland)*, volume 88 of *Electronic Proceedings in Theoretical Computer Science*, pages 72–78, July 2012. Available at `http://arxiv.org/pdf/1207.7148v1.pdf` (accessed on December 27, 2015).

[16] Nachum Dershowitz and Evgenia Falkovich. Generic parallel algorithms. In Arnold Beckmann, Ersébet Csuhaj-Varjú, and Klaus Meer, editors, *Proceedings of Computability in Europe (CiE) 2014: Language, Life, Limits (Budapest, Hungary)*, volume 8493 of *Lecture Notes in Computer Science*, pages 133–142, Switzerland, 2014. Springer. Available at `http://nachum.org/papers/GenericParallel.pdf` (accessed on December 27, 2015).

[17] Nachum Dershowitz and Evgenia Falkovich. The Invariance Thesis. 2015. Submitted. Available at `http://nachum.org/papers/InvarianceThesis.pdf` (accessed on December 27, 2015).

[18] Nachum Dershowitz and Yuri Gurevich. A natural axiomatization of computability and proof of Church's Thesis. *Bulletin of Symbolic Logic*, 14(3):299–350, September 2008. Available at `http://nachum.org/papers/Church.pdf` (accessed on January 13, 2016).

[19] Scott Dexter, Patrick Doyle, and Yuri Gurevich. Gurevich abstract state machines and Schoenhage storage modification machines. *Springer J. of Universal Computer Science*, 3:279–303, 1997.

[20] Gilles Dowek. Around the physical Church-Turing thesis: Cellular automata, formal languages, and the principles of quantum theory. In *Proceedings 6th International Conference on Language and Automata Theory and Applications (LATA 2012, A Coruña, Spain)*, volume 7183 of *Lecture Notes in Computer Science*, pages 21–37. Springer Verlag, Berlin, 2012. Available at `https://who.rocq.inria.fr/Gilles.Dowek/Publi/lata.pdf` (accessed on January 13, 2016).

[21] Denise M. Eckstein. Simultaneous memory access. Technical report, Computer Science Department, University of Iowa, 1979.

[22] Evgenia Falkovich. *On Generic Computational Models*. Ph.D. thesis, School of Computer Science, Tel Aviv University, Tel Aviv, Israel, October 2014. Available at `http://www.cs.tau.ac.il/thesis/thesis/Falkovich.Evgenia-PhD.pdf` (accessed on October 9, 2015).

[23] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings 10th Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978. Available at `https://ecommons.cornell.edu/bitstream/handle/1813/7454/78-334.pdf` (accessed on January 13, 2016).

[24] Robin Gandy. Church's thesis and principles for mechanisms. In *The Kleene Symposium*, volume 101 of *Studies in Logic and the Foundations of Mathematics*, pages 123–148. North-Holland, 1980.

[25] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, Oxford, 1995. Available at `http://research.microsoft.com/~gurevich/opera/103.pdf` (accessed on January 13, 2016).

[26] Yuri Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000. Available at `http://research.microsoft.com/~gurevich/opera/141.pdf` (accessed on January 13, 2016).

[27] H. James Hoover, Maria M. Klawe, and Nicholas J. Pippenger. Bounding fan-out in logical networks. *Journal of the ACM*, 31:13–18, 1984.

[28] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 869–942. North-Holland, Amsterdam, 1990. Draft available at `http://www.eecs.berkeley.edu/Pubs/TechRpts/1988/CSD-88-408.pdf` (accessed on January 7, 2016).

[29] Donald E. Knuth. Algorithm and program: Information and date (letter to the editor). *Communications of the ACM*, 9:654, 1966.

[30] Ian Parberry. Parallel speedup of sequential machines: A defense of parallel computation thesis. *SIGACT News*, 18(1):54–67, March 1986.

[31] Emil L. Post. Absolutely unsolvable problems and relatively undecidable propositions: Account of an anticipation. In Martin Davis, editor, *Solvability, Provability, Definability: The Collected Works of Emil L. Post*, pages 375–441. Birkhaüser, Boston, MA, 1994. Unpublished paper, 1941.

[32] Wolfgang Reisig. On Gurevich's theorem on sequential algorithms. *Acta Informatica*, 39(4):273–305, April 2003. Available at `http://www2.informatik.hu-berlin.de/top/download/publications/Reisig2003_ai395.pdf` (accessed on January 13, 2016).

[33] Wolfgang Reisig. The computable kernel of Abstract State Machines. *Theoretical Computer Science*, 409(1):126–136, December 2008. Draft available at `http://www2.informatik.hu-berlin.de/top/download/publications/Reisig2004_hub_tr177.pdf` (accessed on January 13, 2016).

[34] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3):281–292, 1971.

[35] Peter van Emde Boas. Machine models and simulations. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 1–66. North-Holland, Amsterdam, 1990. Draft available at `https://www.illc.uva.nl/Research/Publications/Reports/CT-1989-02.text.pdf` (accessed on January 13, 2016).

[36] Uzi Vishkin. Implementation of simultaneous memory address access in models that forbid it. *J. of Algorithms*, 4(1):45–50, March 1983.

[37] Dan E. Willard. On the application of sheared retrieval to orthogonal range queries. In *Proceedings of the Second ACM SIGACT/SIGGRAPH Annual Symposium on Computational Geometry*, pages 80–89, 1986.