On the Parallel Computation Thesis

Nachum Dershowitz and Evgenia Falkovich
School of Computer Science, Tel Aviv University, Tel Aviv, Israel
nachum.dershowitz@cs.tau.ac.il; jenny.derzhavetz@gmail.com

March 16, 2015

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.

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 in this work. We shall refer to the individual objects composing such a system as “cells”.

As Galileo observed in the above epigraph, the “manual” of the universe is written in mathematical language. This conviction means that its evolution and the evolution of its components can be expressed in mathematical terms. It has, furthermore, been convincingly argued by Gurevich [26] (presaged by Post [29]) 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 [12, 24, 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 request values from another cell—a message-based framework. Similarly, one can allow one cell to set values in another cell or to request those changes from the other cell (depending again on one’s viewpoint). Interaction and cooperation have been considered within Gurevich’s framework [2, 3, 4, 5]. We take the shared-memory viewpoint here 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 communication (the “topology”) other than at birth (cf. [21]). 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 [13] proved that alternating polynomial time is equivalent to deterministic polynomial space. In 1977, Allan Borodin [10] suggested that this result may be generalized:

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

This thesis is usually referred to as the Parallel Computational Thesis. In 1978, Steven Fortune [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 [28] showed that a Common PRAM with a bit more than an exponential number of initial processes can compute any NP problem in constant time. 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 considered “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 classical algorithms [17, 18], in which it was shown that all (formally) effective 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]. 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.

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 2) 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 tripartite vocabulary $F \cup F' \cup G$ consisting of private (internal) operations $F$, public (global) $G$, and embryonic $F'$, the latter having the same similarity type as $F$. (There could be any fixed number of embryonic copies $F', F'', \ldots, F^{(k)}$, but let us leave it
simple for now: one child at a time.) The individual cells all run a “classical” (sequential) algorithm in the sense of [26, 15].

Initially, all cells agree on $G$ and their $F^0$ are pristine (completely undefined). A single global step of the algorithm comprises of the following stages.

1. First, each cell $C$ takes one classical step, producing a set of updates (changes of values of state locations) $U$.

2. Cells’ private operations $F$ and embryonic operations $F^0$ are updated per $U$.

3. Then the union of all the cell’s public updates together are applied to every cell’s public $G$. If there is any disagreement between cells regarding updates to $G$ (the same location getting contradictory new values), the whole system aborts. (Abortion could be replaced with nondeterministic behavior, should one prefer.)

4. Assuming there are no conflicts, mitosis takes place as follows: Each cell $C$ in which the values of the operations $F^0$ were modified splits into two, a mother $C$ and daughter $C^0$. The daughter $C^0$ inherits $G$, as updated, from her mother; her $F$ is a copy of her mother’s $F^0$. For both mother and daughter, $F^0$ is reinitialized 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 next state, as when it suffers an internal clash between attempted updates.

3 Parallel Algorithms

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

**Definition 1 (Parallel System)** A parallel system consists of a set (or class) $S$ of states, a (nonempty) subset $S^0 \subseteq S$ of which are initial, a federacy of (countably or uncountably many) identities $I$, localized states $X, i \in S$ for each $X \in S$ and $i \in I$, and a (partial) transition function $\tau : S \rightarrow S$. Whenever $\tau$ is undefined for a state $X \in S$, we will say that $X$ is terminal.

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

3.1 Global States

We need for systems to comprise multiple local processes, what we called “cells”. As explained above, states should be formalized as (first-order) logical structures over some vocabulary, fixed by the algorithm. Since we are dealing with parallel algorithms, with both private and shared memory, each cell has a local state,
which is a structure over a (finite) vocabulary $G \cup F$, where the (current) values of operations in $G$ are stored (conceptually, at least) in global locations, accessible to all cells, while private data is stored as values of operations in its personal copy of $F$.

Each cell has its own unique identity, taken from some index set (or class) $\mathcal{I}$. Suppose $F = \{f^1, \ldots, f^k\}$. Then the $k$ local functions of cell $i \in \mathcal{I}$ are indexed $F_i = \{f^1_i, \ldots, f^k_i\}$, where, for each $j = 1, \ldots, k$, the functions $f^j_i$ have the same arity for all cells $i$. It will be convenient in what follows to denote $F^j = \{f^j_i : i \in \mathcal{I}\}$. The global state of the algorithm will be an algebra over the combined (possibly infinite) vocabulary $V = G \cup F^*$, where $F^* = \bigcup_{i \in \mathcal{I}} F_i = \bigcup_{j=1}^k F^j$.

Let $X$ be a state of $A$ with domain (universe, carrier) $D$. Let $g$ be some function in $V$ (either in $G$ or in $F^*$) of arity $n$ and $\bar{u}$ be an $n$-tuple of elements of that domain. Then, by interpreting $g$, $X$ determines the value $[g]_X(\bar{u})$ of its location $g(\bar{u})$. For any ground term $t$, we write $[t]_X = w$ to mean that the value of $t$ as interpreted in $X$ is $w$.

When state $X$ with transition $\tau$ is not terminal, we say that $g(\bar{u}) \mapsto w$ is an update of $X$ if $\tau$ changes the value of $g(\bar{u})$ to be $w$ in $\tau(X)$, which was not its value in $X$. By $\Delta^\tau(X)$, we denote the set of all updates of $X$ under $\tau$.

Each cell works with only part of the global state. We define the $i$th localization $X_i$ of global state $X$ of federacy $\mathcal{I}$ to be the structure $X$ with its “active” vocabulary restricted to $V_i$, meaning that all other functions $(F^* \setminus F_i)$ are everywhere undefined, taking on the otherwise unused value $\bot$. Note that the localization of a localization $X_i$ is $X_i$ itself. The evolution of the $i$th cell should utilize only the values of the defined functions of the $i$th localization, identifying its private $F$ with the indexed functions $F_i$. Transitions for a cell can only change values of its functions and of its progeny. We say that a localization $X_i$ is empty if every function in $F_i$ is also undefined. These are nascent cells, yet to be born. We call $X_i$ an $i$-state when it’s not empty.

Identities are just a fiction to distinguish one cell from another; when comparing states, the individual identities should be ignored. Two states are the same if they are the same up to permutation of cell identities. Similarly, two sequences of state transitions are the same if there is a permutation of identities for the states in one of the sequences that makes it identical to the other sequence.

To facilitate state comparison, we define an anonymization operator $\hat{z}$ that wipes off the identifier, that is, it erases the identity-index from function symbols. Thus, the anonymized $X^\hat{z}_i$ is obtained from a localized cell $X_i$ by restricting the vocabulary to $V_i$ and pretending that the remaining symbols $f^j_i$ are just $f^j$, for all $j$. Accordingly, we say that $X_i = Y_k$, for two localizations, if $X^\hat{z}_i = Y^\hat{z}_k$, that is, if they are identical when anonymized. Similarly, we say that transition $\tau$ generates the same updates for $X_i$ and $Y_k$ if the updates to $G$ are the same in both $X_i$ and $Y_k$, the updates to $F_i$ 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. In such a case, we will simply write $\Delta^\tau(X_i) = \Delta^\tau(Y_k)$. We denote by $\Delta^i(X)$ the set of all updates to locations of
To capture the uniform behavior of cells, we introduce templates, which are terms over an unadorned vocabulary $G \cup \{f^1, \ldots, f^k\}$, where $f^j$ is a symbol of the same arity as the $f^i \in F$. For each $i \in I$, the template $t$ induces a critical term $t_i$, obtained by replacing each occurrence of $f^j$ by $f^i$. Given states $X$ and $Y$ from the same transition system, we say that they agree on a set of templates $T$ and indicate $X \equiv_T Y$ if $[t_i]_X = [t_i]_Y$ for every $t \in T$ and $i \in I$. In words, every localized term defined by a template $t$ has the same value in both $X$ and $Y$.

To compare different cells we should again ignore their specific identities. Let $X_i$ and $X_k$ be distinct localizations of global state $X$. We say that $X_i \equiv_T X_k$ (the two states “agree”) if $[t_i]_X = [t_k]_X$ for each $t \in T$. Similarly, we may compare localizations of two distinct global states. We write $X_i \equiv_T Y_k$, for localization $X_i$ of $X$ and $Y_k$ of $Y$, if $[t_i]_X = [t_k]_Y$ for each $t \in T$.

### 3.2 Algorithms

Let $A = (S, S_0, I, \tau)$ be a parallel system of federacy $I$ over vocabulary $V$. We deem it to be “algorithmic” if it satisfies a number of postulates, which we now proceed to explicate.

**Postulate 1 (Abstract State)** A parallel system is abstract if the following properties hold:

1. Its states are structures over a vocabulary $V$.
2. All states share the same vocabulary.
3. The functions in its states are all strict: $f(\ldots, \bot, \ldots) = \bot$ for all $f \in V$.
4. Its states (and also the set of initial states and the set of terminal states) are closed under isomorphism (of first-order structures).
5. Its states are also closed under localizations, that is, if $X$ is a state, then $X_i$ is also a state, for each identity $i$.
6. Isomorphic states are either both terminal or else their next states are isomorphic, via the same isomorphism.
7. Transitions preserve the domain of states.

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.

It must be possible to describe the effect of transitions in terms of the information in the current state. To that end, we use *templates*, which refer to global locations in the current state and to local locations in each cell. Parallel algorithms use these templates to describe state transitions, without referring to cells individually. If every referenced location has the same value in two states, then the behavior of the algorithm must be the same for both of those states. This, the essence of what makes a process algorithmic, is a crucial insight of [26].

Each cell is fully responsible for its local updates. The updates created by an individual cell may not depend on its identity, but only on global and local locations that are available to it.

**Postulate 2 (Localization)** An abstract parallel system with state-transition $\tau$ and identities $I$ is localized if there exists a finite set $T$ of templates such that $\Delta^\tau(X_i) = \Delta^\tau(Y_k)$ and $\Delta^\tau_i(X) = \Delta^\tau_k(Y)$ whenever $X_i \equiv_T Y_k$ for states $X$ and $Y$ and identities $i, k \in I$.

Moreover, all updates of a global state are generated solely by local cells.

**Postulate 3 (Globalization)** An abstract parallel system with state-transition $\tau$ and identities $I$ is globalized if $\Delta^\tau(X) = \bigcup_{i \in I} \Delta^\tau_i(X)$ for all states $X$ and identities $i \in I$.

For ordinary (non-parallel) algorithms, one asks for the following [26, Bounded Exploration Postulate]:

**Definition 2 (Algorithmicity)** An abstract parallel system with state-transition $\tau$ is algorithmic if there exists a finite set $T$ of templates such that $X \equiv_T Y$ implies $\Delta^\tau(X) = \Delta^\tau(Y)$ for all states $X$ and $Y$.

**Proposition 1** Algorithmicity follows from localization and globalization.

**Proof:** Suppose $X \equiv_T Y$ for states $X,Y$ and some finite set of templates $T$. Assume that $\delta = f(u_1, \ldots, u_n) \mapsto u_0$ is an update of $X$. Then, by globalization, there exists a cell $i$ such that $\delta$ is an update of $X_i$. Since $X \equiv_T Y$ there exists a $j$ such that $X_i \equiv_T Y_j$. From localization, we deduce that $\delta$ is an update of $Y_j$, and from globalization that it is an update of $Y$. Hence $\Delta^\tau(X) \subseteq \Delta^\tau(Y)$. Inclusion of $\Delta^\tau(Y)$ in $\Delta^\tau(X)$ is proved in the same way. Hence $\Delta^\tau(X) = \Delta^\tau(Y)$, as required. $\Box$

### 3.3 Childhood

If some localization of $X$ is empty but is not empty for $\tau(X)$, this indicates that a child has been born. We demand that once a cell has been created, no other cell can change its internals.
**Postulate 4 (Fertility)** An abstract parallel system with state-transition $\tau$ and identities $I$ is fertile if there exists a (input-independent) bound $n \in \mathbb{N}$, such that, for any localization $X_i$ of a state $X$ with identity $i \in I$, $\tau(X_i)$ has at most $n$ non-empty localizations.

The idea here is that in a single step each cell may participate in the creation of only a bounded number of new processes.

Lastly, each newborn cell has exactly one mother:

**Postulate 5 (Motherhood)** An abstract parallel system with state-transition $\tau$ and identities $I$ is maternal if whenever a localization $X_i$ of a state $X$ with identity $i \in I$ is empty, but is non-empty for $\tau(X)$, there is an identity $k \in I$ such that $\Delta^\tau_i(X) \subseteq \Delta^\tau(X_k)$.

### 3.4 Parallel algorithms

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

**Definition 3 (Parallel Algorithm)** A parallel algorithm is a parallel state-transition system that satisfies Postulates 1–5.

**Proposition 2** Any parallel algorithm over a finite vocabulary may be described by an ordinary algorithm.

**Proof:** Consider an algorithm over a finite vocabulary $V$. Then instead of $V = G \cup F^*$, we may assume that we only have $V = G$ (and we required that $G$ be finite). So for this case, the algorithm is only required to have the abstract state and algorithmic properties, and postulates 2–5 are redundant. Also, our final set of templates $T$ is just a finite set of terms over $V = G$. So what we have is a classical (sequential) algorithm with critical terms $T$, as defined by Gurevich in [26].

### 4 Parallel Programs

The two basic program instructions are assignment and creation.

**Assignment.** An atomic assignment is an instruction of the form $g(t^0, \ldots, t^n) := t^0$, where $t^0, \ldots, t^n$ are templates and $g \in V$ has arity $n$.

Let $X_i$ be a localization of $X$. Assume that $[t^j]_X = u^j_i$ for each $j = 0, \ldots, n$. If $g \in G$, then application of an assignment $a$ on $X$ for $i$ generates an update $\Delta^a(X_i) = \{g(u^1_i, \ldots, u^n_i) \rightarrow u^0_i\}$, with the appropriate index $i$. If $g \in F$, then the application generates an denoted $\Delta^a(X_i) = \{f_i(u^1_i, \ldots, u^n_i) \rightarrow u^0_i\}$, with the appropriate index $i$. If one of the $t^j$ is undefined ($\bot$) in $X_i$, then $\Delta^a(X_i) = \emptyset$.

The application of $a$ to $X$ generates the update set $\Delta^a(X) = \bigcup_{i \in I} \Delta^a(X_i)$. **.
Parallel assignment. More generally, a parallel assignment rule $a$ is a finite set of atomic assignments, written out as $a_1 \parallel a_2 \parallel \cdots \parallel a_k$. The update set generated by this instruction is $\Delta^a(X) = \bigcup_{i=1}^k \Delta^{a_i}(X)$. If $\Delta^a(X)$ includes conflicting updates (different values assigned to the same location), then the rule is not applied.

Creation. This is an instruction denoted $\nu.a$ of the form new $a$, where $a$ is a parallel assignment.

Suppose $a$ is a single assignment $f(t^1, \ldots, t^n) := t^0$, and let $X_i$ be one localization. The transition initializes some empty localization $X_i$, by setting $f_k([t^1_i]_X, \ldots, [t^n_i]_X)$ to be $[[t^0_i]]_X$. Then $\Delta^{\nu.a}(X_i) = \{f_k(u^1_i, \ldots, u^n_i) \mapsto u^0_i\}$, where each $u^n_i = [t^n_i]_X$. However, if any one of the $u^n_i$ is undefined, then $\Delta^{\nu.a}(X_i) = \emptyset$. For each cell $i$, the transition chooses a different daughter cell $k_i$. In general, the update is appended to the total set of updates $\Delta^{\nu.a}(X) = \bigcup_{i \in I} \Delta^{\nu.a}(X_i)$.

Guard. An atomic guard is a condition of the form $s = t$ or $s \neq t$. A guard $t = s$ evaluates to $\tau$ for localization $X_i$ if $[t_i]_X = [s_i]_X$, and $t \neq s$ is $\tau$ 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 $\tau$ for $X_i$ when each $c_j$ is.

Guarded assignment. This is an instruction denoted $c : a$ of the form if $c$ then $a$, where $c$ is a guard and $a$ is a parallel assignment rule. Application of $c : a$ to state $X$ generates the set of updates $\Delta^{c:a}(X) = \bigcup_{i \in I} \{\Delta^{a}(X_i) : i \in I, [c]_{X_i} = \tau\}$.

Guarded creation. This instruction $c : \nu.a$ takes the form if $c$ then new $a$, where $a$ is a parallel assignment and $c$, a guard. The assignments are executed on each $X_i$, for which the guard $c$ evaluates to $\tau$.

Definition 4 (Parallel Abstract State Program) A parallel abstract state program is a finite set $P = \{r_1, \ldots, r_n\}$ of rules as above. To execute $P$ on state $X$, all rules are executed simultaneously, that is, $\Delta^P(X) = \bigcup_{r \in P} \Delta^r(X)$. If $\Delta^P(X)$ has conflicting updates, then no updates are applied at all.

For state $X$, we denote by $P(X)$ the state obtained by application of program $P$ on $X$. If no rule in $P$ applies, then $P$ is not defined for $X$.

Note that for each instance of creation, the program chooses new unused indices from $I$ 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 function symbols), the specific choice is immaterial.
5 Representation Theorem

A parallel program $P$ is a characteristic program of algorithm $A$ if $P(X) = \tau(X)$ for each state $X$ of $A$. We shall presume for simplicity that $A$ is over a vocabulary $G \cup F^i$ only and denote it by $G \cup F$. We will also assume that in Postulate 4 we have at most one child born per step $(n = 2)$. All proofs can be easily extended to the general case.

By globalization, $\Delta^\tau(X) = \bigcup \Delta^\tau(X_i)$. So we start with localized states $X_i$. We prove that the transition of $X_i$ can be described by a rule composed of assignment and creation rules.

By our simplifying assumption, the algorithm has only one local function. 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 $\tau(X_i)$ are over $G \cup \{f_i, f_k\}$ for some $k \in I$. So we may treat $X_i$ and $\tau(X_i)$ as ordinary states of an ordinary (non-parallel) algorithm over finite vocabulary $G \cup \{f_i, f_k\}$ with critical terms $T_i \cup T_k$.

Let $\delta = h(u_1, \ldots, u_n) \mapsto w$ be an update in $\Delta^\tau(X_i)$. According to [30, Lemma 5] (or [26, Lemma 6.2]), for each value $u_j = 0, \ldots, n$ there exists a term $t^i \in T_i \cup T_k$ such that $[t^i]_{X_i} = u_j$. Let $\alpha_\delta$ be the ordinary assignment rule $h(t^i, \ldots, t^n) := t^0$. We have $\Delta^{\alpha_\delta}(X_i) = \{\delta\}$. Denote by $\alpha_i$ the assignment obtained as a parallel composition of $\alpha_\delta$ for all $\delta \in \Delta^\tau(X_i)$. Obviously, $\Delta^{\alpha_i}(X_i) = \Delta^\tau(X_i)$.

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 $\tau(X_i)$ are over $G \cup \{f_i, f_k\}$, we may conclude that $h \in G \cup \{f_i, f_k\}$. Accordingly, we partition $\alpha_i$ into two parallel assignment rules: $a_i$ are all those rules with $h \in G \cup \{f_i\}$ and $n_i$ are for rules with $h = f_k$. Obviously, $a_i \parallel n_i$. We may call the characteristic assignment of $X_i$.

Let $a_i^F$ be obtained from $a_i$ by replacing $f_i$ with $f$. Then $a_i^F$ is an assignment rule over the templates $T$. From the definition of parallel assignment, we obtain that $\Delta^{a_i^F}(X_i) = \Delta^{a_i}(X_i)$. Let $n_i^F$ 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 $\Delta^{n_i^F}(X_i) = \Delta^{a_i}(X_i)$. Define the program $\alpha_i^F = a_i^F \parallel n_i^F$. Then $\Delta^{\alpha_i^F}(X_i) = \Delta^{a_i^F}(X_i) \cup \Delta^{n_i^F}(X_i) = \Delta^{a_i}(X_i)$.

**Lemma 1** Let $X_i$ be a localized state of a parallel algorithm with identity $i$, and $\alpha_i$ the characteristic assignment for $X_i$ and $\tau$. Then $\alpha_i^F(X_i) = \alpha_i(X_i) = \tau(X_i)$.

**Proof:** That $\alpha_i^F(X_i)$ is $\tau(X_i)$ follows from the above discussion. That $\alpha_i(X_i)$ is $\tau(X_i)$ follows from [30, Lemma 11].

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

**Lemma 2** Let $X_i$ be a localized state of a parallel algorithm with identity $i$, and $\alpha_i$ the characteristic assignment for $X_i$ and $\tau$. If $Y_i$ is a localized state with the same identity $i$ and $X_i \equiv X_i Y_i$, then $\alpha_i^F(Y_i) = \alpha_i(Y_i) = \tau(Y_i)$.
**Proof:** Since $\alpha^I$ is a rule over $T$ it will contain updates based on the values of $T$ only. Considering that $X_i \equiv_T Y_i$, we will have $\Delta^\alpha^I(X_i) = \Delta^\alpha^I(Y_i)$. It follows from the previous lemma that $\alpha^I(X_i) = \tau(X_i)$. According to the localization postulate, we have $\Delta^\tau(Y_i) = \Delta^\tau(X_i)$, again since $X_i \equiv_T Y_i$. Combining all together, we conclude that $\alpha^I(Y_i) = \tau(Y_i)$, as claimed.

Every localized state $X_i$ induces an equivalence relation $\sim_{X_i}$ on templates $T$ according to which $s \sim_{X_i} t$ iff $\llbracket s_i \rrbracket_{X_i} = \llbracket t_i \rrbracket_{X_i}$. We show next that update commands for localization $X_i$ are determined by this relation.

**Lemma 3** Let $X_i$ be an $i$-state of an algorithm, $Y_k$ an $k$-state, and $\alpha_k$ the characteristic assignment for $Y_k$ and $\tau$. If $\sim_{X_i} = \sim_{Y_k}$, then $\alpha^I(X_i) = \alpha_k(X_i) = \tau(X_i)$.

**Proof:** We may treat $X_i$ and $Y_k$ as ordinary states over finite vocabularies, as we did at the start of this section. We are given that $\llbracket s_i \rrbracket_{X_i} = \llbracket t_i \rrbracket_{X_i}$ iff $\llbracket s_k \rrbracket_{Y_k} = \llbracket t_k \rrbracket_{Y_k}$ for any templates $s, t \in T$. By [30, Lemma 13], we get $\alpha_i(X_i) = \tau(X_i)$. By Lemma 1, we may conclude that $\alpha^I(X_i) = \tau(X_i)$. Recall that we consider states to be equal if they are equal up to a permutation of identities.

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

**Theorem 1 (Representation)** For each parallel algorithm, there exists a characteristic parallel abstract program.

**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 localized 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 $R_{\sim} = \mathbf{if} \; c_{\sim} \; \mathbf{then} \; \alpha^I_{X_{\sim}}$, where $\alpha_{\sim}$ is the characteristic assignment for $X_{\sim}$. Obviously $c_{\sim}$ evaluates to $T$ on $X_{\sim}$, and hence $R_{\sim}(X_{\sim}) = \alpha^I_{X_{\sim}}(X_{\sim})$.

Define $P$ to be the parallel program consisting of rules $R_{\sim}$ for all possible equivalence relations $\sim$ of $T$, for which there is at least one state $X_{\sim}$. Since $T$ is finite, 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, $P(X) = \tau(X)$ for any state $X$.

Consider some localized state $X_i$ satisfying the relation $\sim_i$ on templates. By Lemma 3, $\alpha_{\sim_i}(X_i) = \tau(X_i)$. Exactly one guard in $P$ applies to $X_i$ and that is $c_{\sim_i}$. So $P(X_i) = P_{\sim_i}(X_i) = \alpha^I_{X_{\sim_i}}(X_i) = \alpha_{\sim_i}(X_i) = \tau(X_i)$.

Assume finally that $X$ is a general state of the algorithm. By **globalization**, the update of $X$ is a union of updates of all its localizations $X_i$, that is, $\Delta^\tau(X) = \bigcup_{i \in I} \Delta^\tau(X_i)$. By the **abstract state** axiom, $X_i$ is also a state. According to **localization**, 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 $\Delta^\tau(X_i) = \Delta^\tau(X_i)$ for all $i \in I$, 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. Only a global state that can be described finitely can be considered effective.

6.1 Effective State

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 could easily require 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 non-trivial relations, which might hide non-computable information.

Definition 5 (Effective State [8]) 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); and
3. finitely-many other defined locations, not having the default value, \( \bot \).

Furthermore, the number of non-empty cells in a global state is finite and input-independent.

Constructors provide a way to give a unique name for any domain element, and 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 \( [c_i(c(x_1, \ldots, x_i, \ldots, x_n)) = x] \), 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, \ldots\} are constructed from a scalar (nullary) constructor \( \epsilon() \) and unary constructors \( a() \), \( b() \), 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) \( \varepsilon \) 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) \texttt{nil} (the empty list) and (binary) \texttt{cons} (which adds an element to the head of a list); the destructors are \texttt{car} (first element of list) and \texttt{cdr} (rest of list). To construct 0-1-2 trees, we would have three constructors, \( A() \), \( B() \), and \( C(\_\_\_\_\, \_\_\, \_\_\,) \), 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.

Initial states may include constructor operations, which are certainly effective (they are just a naming convention for domain values). Given free constructors, equality of domain values and destructors are also effective (see [8]). We may assume that domains include two distinct truth values, and—furthermore—that we have (scalar) constructors, \texttt{T} and \texttt{F}. Boolean operations are effective finite tables, so we may presume them.

Without loss of effectiveness, we can allow any finite amount of non-trivial data, provided that—except for the input values—all initial states are the same. Otherwise, initial states could hide uncomputable outputs. Moreover, initial states can have effective operations. The circularity of this definition of effectiveness “bottoms-out” with operations that are programmable directly from the constructors. We are presuming that constructors are present in states—even if the algorithm avoids their direct use.

To preclude ineffective information in the initial setup, the number of cells that are active at the outset is bounded. Those cells can create new cells whose number may depend on the input.

### 6.2 Effective Algorithm

Obviously, an initial state should also be allowed to include some input. To handle inputs, we postulate some subset of the templates, namely the \textit{input terms}, for which every possible combination of domain values occurs in some initial state, and such that all initial states agree on all terms over the vocabulary of the algorithm except these.

**Postulate 6 (Effectiveness)** An algorithm is effective if all its initial states are effective and states over the same domain are all identical except for inputs.

Two other ways of capturing the notion that initial states have a finite description, thereby characterizing effectiveness, have been suggested. One alternative [31] 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 a Turing machine for determining whether a state interprets two terms (given as strings) as the same domain value. A second alternative [19] requires that there exist an (arbitrary) injection from the chosen domain of the algorithm into the natural numbers such that the given base functions (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 \( \rho_D(f) : \mathbb{N} \to \mathbb{N} \) is (partial) recursive for every operation \( f \) of its initial states.

In contrast to Definition 5, 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 computational model over any domain [9].

6.3 Basic Algorithms

Definition 6 (Basic) An effective algorithm is basic if its initial states have no infinitely-defined operations.

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 bigger operations, such as multiplication. Therefore, we have allowed an effective state to be equipped with “effective oracles”, which can be obtained by bootstrapping from a basic algorithm with the chosen constructors. Still, when measuring time complexity, we will want to charge more than unit cost for such programmed operations. To get that, we take advantage of the fact that every effective algorithm can have its defined operations “in-lined”, yielding a basic algorithm.

Finiteness of templates, together with commutativity with isomorphism, guarantees that only finitely many locations 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.

Definition 7 We say that a parallel algorithm is basic (effective, resp.) if its initial state is basic (effective, resp.).

Definition 8 A basic parallel algorithm is in function-normal form if all its functions (global and local) are nullary or unary, except for one binary constructor.

Lemma 4 Any effective (basic) parallel algorithm may be emulated by an effective (basic) parallel algorithm in function-normal form.

Proof: This proof for a different case of abstract state machines was suggested in [20]. The idea is that a function with \( n \) arguments can be considered as a function 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 functions will have the same names as in the original one, except that functions 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 \( f((u_1, \langle u_2, \langle u_3, \ldots, \langle u_{n-1}, u_n \rangle \rangle, \ldots)) \)). Since the vocabulary of the algorithm is finite, this can be done during the same transition. \( \square \)
7 Measuring Complexity

The common approach 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” [33, pp. 10–11], under which every transition is counted as a one time unit. Later, we will address the question of what cost to assign to each transition step.

To handle arbitrary data types, the only sensible and honest way is to define the size of a domain element to be the number of basic operations required to build it:

Definition 9 (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 \([\log n]\); for example, \(|5| = 3\), the length of \(0(11)\), 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 [8]). The size of the tree \(C(B(A(A()),A()),B(A(A()),A()))\) is only 3, because subtrees can be reused, and the whole tree can be specified by

\[C(s, s, s) \text{ where } s = B(r, r), r = A().\]

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 10 (Complexity) We measure the (time) complexity of an effective algorithm by the number of basic operations (constructors, destructors, equality) required to perform the computation from initial to final 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.

Consider an effective algorithm \(\text{rev}\) 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: \(\langle\langle\rangle\rangle\), \(\langle\langle\langle\rangle\rangle\rangle\),..., \(\langle\langle\langle\langle\langle\rangle\rangle\rangle\rangle\rangle\),..., \(\langle\langle\langle\langle\langle\langle\langle\rangle\rangle\rangle\rangle\rangle\rangle\rangle\rangle\),.... The function \(\text{rev}: \mathcal{L} \to \mathcal{L}\) takes a list \(\langle l_1...l_n\rangle\) and returns \(\langle l_n...l_1\rangle\), with the sublists \(l_i\) unchanged. For instance, \(\text{rev}(\langle\langle\langle\langle\langle\langle\rangle\rangle\rangle\rangle\rangle\rangle))=(\langle\langle\langle\langle\langle\langle\rangle\rangle\rangle\rangle\rangle\rangle)\).
Now, \texttt{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 \texttt{rev} is not what is intended; we want to count the number of basic list operations needed to reverse a list of length \(n\). So there is no escape but to take into account how \texttt{rev} is implemented internally. Suppose \texttt{rev}(x) is something like this:

\[
y := x; z := \text{nil} \\
\text{repeat} \\
\text{if } y = \text{nil} \\
\quad \text{then return } z \\
\text{else } [z := \text{cons}(\text{car}(y), z); y := \text{cdr}(y)]
\]

We want to count the operations executed by this implementation, which is \(cn\) for some constant \(c\) that is the (maximum) number of (constructor and destructor) operations in a single iteration. Note that any straightforward Turing machine would require many more steps, quadratic in the size of the input \(x\), rather than the number of elements at the top level, as in this list-based algorithm. 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 for the list \(l = \langle \langle \rangle \langle \rangle \rangle\), \texttt{car(\texttt{rev(car(\texttt{rev}(l))))}) should return the number that represents \(\langle \rangle\).

8 Parallel Random Access Machines

For the definition of RAMs, we take the set of instructions suggested by Cook and Reckhow in [14] and use the classification of RAM machines suggested by Boas in [33].

8.1 RAMs

For basic RAMs, the following operations are considered to each take “unit time”:

1. \(X \leftarrow C\), where \(X\) is a register and \(C\) is a constant.

2. \(X \leftarrow [Y]\), where \([Y]\) denotes the contents of the memory location indexed by \(Y\).

3. \([Y] \leftarrow X\).

4. \texttt{TRA m if } X > 0: \text{ Transfer control to the } m\text{-th line of the program if } X > 0.

5. \texttt{READ } X. \text{ Get next input value.}

6. \texttt{PRINT } X. \text{ Print to the output tape.}

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

7. \texttt{INC } X. \text{ Increase the value of register } X \text{ by 1.}
8. **DEC X.** Decrease the value of register X by 1.

Arithmetic RAMs are the model originally defined by Cook and Reckhow in [14]; they extend basic RAMs with addition and subtraction:

7. \( X \leftarrow Y + Z \).
8. \( X \leftarrow Y - Z \).

Multiplication RAMs extend Arithmetic RAMs with multiplication and division:

9. \( X \leftarrow Y \times Z \).
10. \( X \leftarrow Y \div Z \).

A multidimensional RAM operates with a multidimensional memory, rather than the classical one dimensional array. Thus an entry address is defined by a tuple of natural numbers.

### 8.2 PRAMs

A parallel RAM (PRAM) consists of several independent sequential processors, each with its own private memory and communicating with one another through 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 a type of RAM unit time operations, that is, in one step of 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 does and will receive from parent the label for the “first command to execute”. We use `fork` in the way it was pioneered in [23]:

11. `fork label`. Create a child process which starts execution from `label`.

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

Another important classification is by restriction on shared memory access. In a single step of a PRAM, each process can access an entry in shared memory for either reading or writing. And each type of access can be either exclusive (one process access) or common (multiple process access) under some restriction. The exclusive read/write restriction prevents reading from, writing to the same global memory cell simultaneously by two distinct processors. We denote these options by \( R(\text{ead}), W(\text{rite}), E(\text{xclusive}), C(\text{ommon}) \). So CREW PRAM stands for “common-read exclusive-write parallel random-access-machine”. In this model, any process may read any shared memory entry in any step. But in a single step any entry may be written to by at most one process.
A common write machine should have in its description a restriction for conflict resolution, for a case when multiple processors call for a write to the same global memory cell. Some commonly used methods are: (a) COMMON model—all processors writing to the same location write the same value; (b) ARBITRARY model—any process participating in common write may succeed and algorithm should work correctly, regardless of winner; (c) PRIORITY model—there is a linear order on processors and the one with minimal priority 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 of processors and with only \(O(\log P)\) time overhead, where \(P\) is the number of processors [22].

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 is the integers (and whatever is isomorphic to the integers). The states are all endowed with the arithmetic capabilities of PRAMs. The global PRAM memory is a global function; 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 to its proper place.

As we do not allow an effective parallel algorithm to start with initial non-trivial cells, the effective parallel algorithm corresponding to a PRAM would have to first create some input-dependent number of cells and supply them with their local data. The individual cells can also be told what their id is when they are 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

Lemma 5 Any effective parallel algorithm in function-normal form can be simulated by a three-dimensional Successor Common PRAM with oracle access to some injection \(H : \mathbb{N}^3 \rightarrow \mathbb{N}\) and with word size big enough for one step processing of desired \(H\) values.

The overhead in running time is some constant multiplicant, which depends on the simulated algorithm. The number of required processors is equal to the number of cells.

Proof: Let \(A\) be an effective parallel algorithm with global functions \(G = \{g_1, \ldots, g_k\}\) and local functions \(F = \{f_1, \ldots, f_l\}\). Let \(X\) be a state of \(A\). Let \(D\) be the domain of \(X\). Let \(C \subseteq G\) be the constructors of \(D\). We choose some order on \(C\), that is, \(C = \{c_1, \ldots, c_k\}\). Recall that we identify \(D\) with a free-term algebra over \(C\).
**Domain simulation.** We first define an injection \( E : D \to \mathbb{N}^3 \) in the following recursive way:

- \( E : \bot \mapsto (0, 0, 0) \)
- \( E : c_i() \mapsto (i, 0, 0) \), when \( c_i \) is a nullary constructor (i.e. a scalar constant);
- \( E : c_i(u) \mapsto (i, J(u), 0) \), when \( c_i \) is a unary constructor, and \( u \in D \) is any domain element;
- \( E : c_i(u, u') \mapsto (i, J(u), J(u')) \), when \( c_i \) is the unique binary constructor, and \( u, u' \in D \) are any domain elements.

\( J : D \to \mathbb{N} \) is defined by \( J(u) = H(E(u)) \).

\( E \) is an injection since we identified \( D \) with a free-term algebra; \( J \) is an injection since it is the composition of two injections.

**Algebra simulation.** We are going to describe a multidimensional PRAM state \( X_H \) that will simulate \( X \) via domain injection \( J \). State \( X_H \) includes the following:

- a number of processors equal to the number of cells in \( X \),
- a three-dimensional shared memory, referred to by \( G \),
- a two-dimensional local memory for each processor, referred to by \( F \).

To each local cell \( X_i \) in \( X \) we allocate one processor in \( X_H \), which we refer to it as \( p_i \). In the shared memory of \( X_H \), we will store global values of \( X \). The local memory of each processor \( p_i \) will store local values of cell \( X_i \). Isomorphism of states is defined as follows:

- \( G[i, 0, 0] = J([g_i]) \) if \( g_i \) is a global constant and 0 otherwise;
- \( G[i, J([u]), 0] = J([g_i(u)]) \) if \( g_i() \) is a global function and 0 otherwise;
- \( G[i, J([u]), J([v])] = J([g_i(u, v)]) \) for the unique binary constructor \( g_i(\cdot, \cdot) \) and 0 otherwise;
- \( F[i, 0] = J([f_{i_k}]) \) for some processor \( p_k \) if \( f_i \) is a local constant and 0 otherwise;
- \( F[i, J([u])] = J([f_{i_k}(u)]) \) if \( f_i() \) is a local function and 0 otherwise;
- All other entries of shared and local memories are 0.
State simulation. The only information that $X_H$ is missing to simulate $X$ via injection $J$ are the values of the templates $T$. For further convenience, we assume that $T$ is closed under the subterm relation (otherwise we take the closure). Then each local cell $X_i$ knows the values of its critical terms $T_i$. Hence, each processor $p_i$ should keep a pointer for the values of terms in $T_i$. For this, for each term $t_i \in T_i$ the process $p_i$ will store a constant named $t_i$ in its local memory. And if $[t_i] = u$ at $X_i$ then $p_i$ should store $J(u)$ as the value of its local constant $t_i$. With this information, $X_H$ simulates $X$ via injection $J$.

Transition simulation. We next show that there exists a program $P_H$ for a multidimensional PRAM with oracle access to $H$ such that if $\tau(X) = Y$ then $P_H(X_H) = Y_H$, where $\tau$ is a transition function of algorithm $A$.

Let $X$ be a state of $A$. By globalization, a transition from $X$ may be viewed as union of transitions of all local cells $X_i$ of $X$. By localization, updates of $X_i$ are the same, weather it is a global state with just one cell or a local cell of a bigger state, that is, $\tau(X) = \bigcup_i \tau(X_i)$. Hence it is enough to provide a program $P_H$ such that $P_H(X_{i,n}) = Y_{i,n}$ for any $i \in I$. In more general way, it is enough to prove that $P_H(X_H) = Y_H$ for any localized state $X$.

So let $X$ be a localized state for some $i$. Let $Y := \tau(X)$. Let $X_H$ be a PRAM state simulating $X$, as we described above. Then $X_H$ has only one processor. Let $T$ be templates of $A$. Let $P$ be a characteristic parallel program of $A$, as described in Theorem 1. For each transition, $P$ performs a bounded number of basic operations on critical terms: comparisons, assignments, and creation commands. We should explain how a PRAM may simulate each single basic operation:

- Basic comparison operations ask to compare the values of two critical terms. Since, as we assumed, $X_H$ 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 data entry in one step.

- A basic assignment command $h(s) := t$ applied on $X$ creates one update $\delta = h([s]_X) \rightarrow [t]_X$. We assumed that $T$ is closed under the subterm relation. Hence at $X_H$ we have local values for all terms $s$ and $t$.

If $h$ is some global function $g_i$, an assignment is simulated by a shared memory write command: $G(i, J([s]), 0) \leftarrow J([t])$. If $h$ is some local function $f_i$, the assignment is then simulated by a local memory write command: $F(i, J([s])) \leftarrow J([t])$. This again can be done in one operation, since we assumed that a processor may operate any data entry in one step.

- The new command is simulated by the fork command of PRAMs. An application of this command returns 0 for a child and the child’s process id (pid) for a mother. This provides a way for a process to know that it is a “newborn” one. Some initial information that a mother passes to her child should be created by the mother in shared memory. A mother
should wait for the child to copy this information to its local memory and then the mother should clean it. Only after that may the mother move to the next step. Since, per *motherhood*, a mother may create only a bounded number of data entries for a child, 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 (depending on the algorithm) number of PRAM steps.

Note that a PRAM 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:

\[
\begin{align*}
&f(0) = 0 \\
&f(0) := 1 \\
&f(0) = 1 \\
&f(1) = f(0)
\end{align*}
\]

whereas simultaneous application results in:

\[
\begin{align*}
&f(0) = 0 \\
&f(0) := 1 \text{ || } f(1) = f(0) \\
&f(0) = 1 \\
&f(1) = 0
\end{align*}
\]

To avoid this, before a PRAM starts to construct the updates of \( X_H \), it should make a copy of all critical term values in \( X_H \) and use them as reference. 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 number of data units may be passed from mother to child. Assume that this bound for our algorithm is \( d \).

So to simulate one transition of a parallel program (and thus of parallel algorithm), a PRAM process should do as described in Algorithm 1.

Sleep pauses are inserted to synchronize the actions of distinct processes.

**Lemma 6** Any basic effective parallel algorithm may be simulated by a Multiplication Common PRAM with only constant multiplicand cost 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 function-normal form, according to Lemma 5, we only have
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 instructions.
   - Stay here for exactly \( m \) operations (sleep if required).
3. Create initial information for a children in shared memory.
   - Stay here for exactly \( nd \) operations (sleep if required).
4. FORK the required number of times and wait for children to update their initial information.
   - Stay here for exactly \( n \) operations (sleep if required).
5. Clean children's information from shared memory.
   - Stay here for exactly \( nd \) operations (sleep if required).
6. Update critical term values for the next step.

to show that we may compute some bijection \( H : \mathbb{N}^3 \rightarrow \mathbb{N} \), preserving the logarithmic size. And this may be computed by the Cantor tuple function:

\[
\pi(x_1, x_2) = \frac{1}{2}(x_1 + x_2) \cdot (x_1 + x_2 + 1) + x_2
\]
\[
\pi^3(x_1, x_2, x_3) = \pi(\pi(x_1, x_2), x_3)
\]

This may be computed with 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 can be done again by \( \pi^3 \). \(\square\)

Lemma 7 Any Multiplication Common PRAM with time complexity \( T(n) \) and with \( P(n) \) processors may be simulated by an Arithmetic Common PRAM with

\[
O(\log(nT(n)) \cdot \log \log^2(nT(n)) \cdot \log \log \log(nT(n)))
\]

\(\text{time overhead and with }\)
\[
P \cdot \log(nT(n)) \cdot \log \log(nT(n)) \cdot \log \log \log(nT(n))
\]

processors.

Proof: It was proved in [32] that multiplication of \( n \)-bit numbers can be done by circuits of bounded fan-in with depth \( O(\log n) \) and number of agents \( O(n \cdot \log n \cdot \log \log n) \). (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.) It was proved 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 multiplicand 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 $O(n \log n \cdot \log \log n)$ 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 $n$, during $T(n)$ steps the maximal value it may attain is $n2^{T(n)}$, which can be stored in memory using $\log(n2^{T(n)}) = \log n + T(n)$ bits. According to the above, multiplication of numbers with $\log n + T(n)$ bits can be done in $O(\log(n + T(n)))$. To do so, one process may require an extra order 

$$(\log n + T(n)) \cdot \log(\log n + T(n)) \cdot \log \log(n + T(n))$$

processes.

Recall that we simulate 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 an order of 

$$(\log n + T(n)) \cdot \log(\log n + T(n)) \cdot \log \log(n + T(n))$$

times. And then all those helpers may perform multiplications in $O(\log(\log n + T(n)))$ steps. We may FORK from child processes also, until we have enough processes. Hence, creating $n$ processes will require $\log n$ steps. Creating order 

$$(\log n + T(n)) \cdot \log(\log n + T(n)) \cdot \log \log(n + T(n))$$

processes may be done in 

$$O(\log((\log n + T(n)) \cdot \log n + T(n)) \cdot \log \log(n + T(n))) = O(\log(n + T(n)))$$

steps. It follows that the overall time for one multiplication is still $O(\log(\log n + T(n)))$ steps. The total number of processors used will be:

$$O(P(n) \cdot (\log n + T(n)) \cdot \log(\log n + T(n)) \cdot \log \log(n + T(n)))$$

given that $P(n)$ is the number of processors in initial multiplication PRAM. □

**Lemma 8** Any basic effective algorithm with time complexity $T(n)$ and with $P(n)$ processors can be simulated by an Arithmetic EREW PRAM with time complexity $T(n) \cdot \text{polylog}(nT(n)) \cdot \text{polylog} P(n)$ and with $P(n) \cdot \text{polylog}(nT(n))$ processors.
Proof: Let \( A \) be a basic effective algorithm with time complexity \( T(n) \) and with \( P(n) \) processors. It may be simulated by an Arithmetic Common PRAM with time complexity \( T(n) \cdot \text{polylog}(T(n)) \) and with \( P(n) \cdot \text{polylog}(T(n)) \) number of processors, as proved in Lemma 7. An Arithmetic Common PRAM in turn may be simulated by an EREW PRAM of the same type with only \( \log P(n) \) time overhead, where \( P(n) \) is the number of processors, as was proved in [22, 34]. Hence, any effective algorithm with time complexity \( T(n) \) and with \( P(n) \) processors may be simulated by an Arithmetic EREW PRAM with time complexity \( T(n) \cdot \text{polylog}(nT(n)) \cdot \text{polylog} P(n) \) and with \( P(n) \cdot \text{polylog}(nT(n)) \) processors.

We have arrived at a substantiation of the Parallel Computation Thesis:

Theorem 2 (Parallel Computation) Polynomial time for effective parallel algorithms—with the number of cells no more than exponential in running time—is equivalent to polynomial space of Turing machines.

Proof: It was proved in [23, Thm.1] that

\[
\bigcup_{k=1}^{\infty} T(n)^k\cdot\text{time-Arithmetic-PRAM} = \bigcup_{k=1}^{\infty} T(n)^k\cdot\text{TM-space}
\]

provided that \( T(n) \geq \log n \) and the number of processors of the PRAM is no more than exponential in the (parallel) running time. The theorem thus follows from this fact and Lemma 8.

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 [1, 9] successfully characterized parallel algorithms within the abstract-state-machine framework, but their approach is not easily restricted to be effective. 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 [19, 9], we have characterized what makes a parallel algorithm effective, demanding that the initial global state be finitely describable. This decomposes into two main requirements: (a) that each cell itself be an effective classical algorithm; (b) that there be only finitely many cells initially, though their number may depend on the input value. This allowed us to establish the veracity of an “invariance” thesis for parallel algorithms, as has recently been done for classical algorithms [17, 18]: all effective parallel models of computation can be polynomially simulated by a standard model (PRAM). The Parallel Computation Thesis follows: any reasonable parallel model cannot do more in polynomial time than can a Turing machine with polynomial space.

Our model is simpler than that of Blass and Gurevich for the 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. For example, one can imagine a cell for each of uncountably many points on a line segment in three-dimensional space and an algorithm that applies, in parallel, an affine transformation to the coordinates of each point cell, resulting in a translated segment.

In this work, we have only considered discrete-time systems, where all cells progress in lockstep with each other. We plan to expand this line of work in the following directions:

- Distributed systems, where cells each progress at their own rate, require separate treatment. This will require a sense of identity for cells and a means of communication between them. Cf. [6, 7, 16].
- Systems that evolve in continuous time; see [11, 16].

References


