Enumerated BSP Automata

Gaetan Hains

Abstract Parallel software needs formal descriptions and mathematically verified tools for its core concepts that are data-distribution and inter-process exchanges or synchronizations. Existing formalisms are either non-specific, like process algebras, or unrelated to standard Computer Science, like algorithmic skeletons or parallel design patterns. This has negative effects on undergraduate training, general understanding and mathematically-verified software tools for scalable programming. To fill a part of this gap, we adapt the classical theory of finite automata to bulk-synchronous parallel computing (BSP) by defining BSP words, BSP automata and BSP regular expressions. BSP automata are built from vectors of finite automata, one per computational unit location. In this first model the vector of automata is enumerated, hence the adjective *enumerated* in the title. We also show symbolic (intensional) notations to avoid this enumeration. The resulting definitions and properties have applications in the areas of data-parallel programming and verification, scalable data-structures and scalable algorithm design.

1 Introduction and Background

This paper introduces a new theory of bulk-synchronous parallel computing (BSP), by adapting classical automata theory to BSP. It attempts to provide the simplest possible mathematical description of BSP computations. With maximal reuse of existing Computer Science it is hoped that this theory will find its way into more complex formalisms for parallel programming tools, language designs and software-engineering.

BSP is a theory of parallel computing introduced by Valiant in the late 1980s [20] and developed by McColl [14]. Unlike theories of concurrency that generalize sequential computation, BSP retains the deterministic and predictable behaviour of sequential machines of the Von Neumann type, while taking advantage of concurrent

G. Hains (✉)
Huawei France R&D Center, 20 quai du Point du Jour, 92100 Boulogne-Billancourt, France
e-mail: gaetan.hains@huawei.com

© Springer International Publishing Switzerland 2017

A. Adamatzky (ed.), *Emergent Computation*, Emergence, Complexity
and Computation 24, DOI 10.1007/978-3-319-46376-6_10

execution for accelerating computations. **Concurrency theory is related to BSP in the way microscopes are related to telescopes: they are built from similar components but look in opposite directions** (Fig. 1).

Just as parallel algorithms are a *special case* of sequential algorithms (they realize a sub- complexity class of problems i.e. NC $\subseteq$ P), BSP machines are close relatives of sequential machines whose instruction cycles are built from vectors of asynchronous sequential computations and are called *supersteps*. When the vector's elements terminate, they are globally synchronized, which guarantees determinism and allows predictable performance. This global sequence is called a superstep: launch a vector of asynchronous and independent sequential computations, wait until they all terminate and synchronize them. A *BSP computation* is a sequence of supersteps realized by a BSP computer, that is a vector of Von Neuman sequential computers linked by a global synchronization device. As we will see by adapting finite automata theory to BSP

- BSP automata are special cases of finite-state machines because they are finitely-defined systems, but
- the correspondence is not trivial and both the finite-alphabet hypothesis and the classical theory of product automata have to be adapted to account for the two-level nature of BSP computation.

In the usual definition and application of BSP, the sequential elements also *exchange* data during synchronization and there is a simple linear model of time complexity that estimates the delay for synchronization and data exchange in units of sequential computation. The model defined in the present paper can be extended to represent communication, as in the BSPCCS process algebra [16].

2 Bulk-Synchronous Words and Languages

Automata theory is both an elementary and standard part of Computer Science and an area of advanced research through topics such as tree automata, pattern matching and concurrency theory. It is also universally used by computing system through lexical analysis, text processing and similar operations. We are interested here in the core elementary theory of automata as described for example in [15, 21] or in the initial chapters of graduate textbooks such as [9, 18].

Let Σ be a finite alphabet and $p > 0$ an integer constant. Elements of Σ represent inputs to the automaton, or more generally events it takes part into. Constant p represents the assumption of a vector of parallel computation units executing the events or receiving them as signals. The local sequential computers or computations are indexed by $[p] = \{0, 1, \dots, p-1\}$ and variable i will be assumed to range over $[p]$. In programming systems like BSPlib [7] this variable i is called `pid` for "Processor ID". A value $i \in [p]$ is sometimes called a *processor*, or an *explicit process* [11] or simply a *location*. Throughout the paper, all vectors will be assumed to be indexed by $[p]$.

Concurrent systems & theories	Bulk-synchronous parallelism
Arbitrary asynchronism	Structured asynchronism
High-complexity	Low-complexity
Distributed computing	Parallel computing
Unpredictable performance	Predictable performance
Endless processes like servers	Finite processes like algorithms
Not scalable	Massively scalable
Pairwise synchronizations	Collective synchronizations
Implicit shared memory	Explicit distributed memory
Implicit processes	Explicit processes (pid variable)

Fig. 1 Concurrency versus bulk-synchronous parallelism

Our whole theory of BSP computation is parametrized over this constant p, which is thus "static" and fixed for a given application of the theory. There is now a large body of research about BSP and many generalizations have been studied but for the sake of generality we model here only the standard core of BSP.

Our first definition represents the asynchronous part of supersteps: vectors of sequential computations, which automata theory sees as p-vectors of traces or *word-vectors*.

Definition 1 Elements of $(\Sigma^*)^p$ will be called *word-vectors*. A *BSP word* over Σ is a sequence of word-vectors i.e. a sequence of $((\Sigma^*)^p)^*$. A *BSP language* over Σ is a set of BSP words over Σ.

Remark 1 The word-vector $< \epsilon, \ldots, \epsilon >$ is not equivalent to an empty BSP word ϵ as the former will trigger a global synchronization, while the latter will not. In other words, $< \epsilon, \ldots, \epsilon >$ has length one and ϵ has length zero.

In our examples we will assume that $\Sigma = \{a, b\}$, ϵ is the empty "scalar" word and $p = 4$ without loss of generality.

For example $\mathbf{v}_1 =< ab, a, \epsilon, ba >$ and $\mathbf{v}_2 =< bbb, aa, b, a >$ are word-vector and $\mathbf{w} = \mathbf{v}_1\mathbf{v}_2$ is a BSP word. It is understood that $\mathbf{w}$ represents two successive supersteps and that

$$\mathbf{w} = \mathbf{v}_1\mathbf{v}_2 \neq < abbbb, aaa, b, baa >$$

that is: concatenation of BSP words is not the same as pointwise concatenation of word-vectors. Concatenation of BSP words represents phases of collective communications and barrier synchronizations (see Fig. 2, where vectors are drawn vertically). Concatenation of BSP words accordingly means concatenation of (sequences of) word vectors:

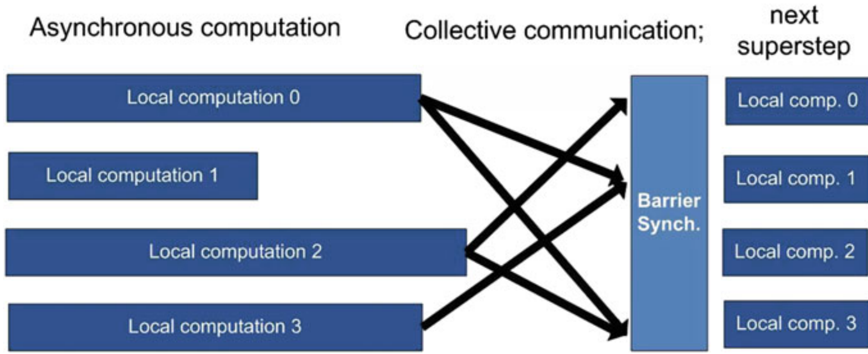

Fig. 2 A BSP superstep

$$\mathbf{w} = \mathbf{v}_1\mathbf{v}_2 =< ab, a, \epsilon, ba >< bbb, aa, b, a > .$$

Let $\mathbf{v}_a =< \epsilon, a, aa, aaa >$ respectively $\mathbf{v}_b =< \epsilon, b, bb, bbb >$ be word-vectors whose local words are a^i and b^i respectively. Then $L_a = \{\mathbf{v}_a\}$, $L_b = \{\mathbf{v}_b\}$ and $L_2 = \{\epsilon, \mathbf{v}_a, \mathbf{v}_b, \mathbf{v}_a\mathbf{v}_b\}$ are finite BSP languages and $L_3 = \{\epsilon, \mathbf{v}_a, \mathbf{v}_a\mathbf{v}_b, \mathbf{v}_a\mathbf{v}_b\mathbf{v}_a, \ldots\}$ is an infinite BSP language.

3 Finite Versus Infinite Alphabet

A BSP word is built from an infinite alphabet: even when Σ is finite, the set of word-vectors will be infinite. This part of the model illustrates the fact that a BSP computer is two-level: it is built from sequential computers, whose computations are finite but of unlimited length. But the infinite-alphabet property is not caused by the (finitely-many) computing elements, it would still hold if $p = 1$. It is rather a consequence of the fact that synchronization barriers are cooperative and not pre-emptive. Individual local computations have to *terminate* before a superstep ends with synchronization.

In his famous paper [19] Turing gives sketches several arguments for the choice of a finite alphabet. One is physical-topological: infinite alphabets realized by a finite physical device would require infinite precision of the device reading a symbol from working memory. He also gives another argument against infinite alphabets:

> compound symbols [such as arabic numerals], if they are too lengthy, cannot be observed at a glance

and even mentions, less convincingly, the case of Chinese ideograms as an attempt

> to have an enumerable infinity of symbols.

So our notion of BSP computation would appear to be incoherent with classic Church-Turing models: it is built from an infinite alphabet of symbols. However that

would only be the case if we chose to use BSP languages as a model for decidability, which they are not intended to be. The BSP model was invented to model parallel *algorithms*, not arbitrary parallel computations. All local computations are therefore assumed to terminate and so is the global sequence of supersteps.

The best point of view on this question of infinite-vs-finite alphabet for BSP is that **BSP languages are sets of traces having a series-parallel structure representing the behaviour of parallel computers that all synchronize periodically**.

4 Bulk-Synchronous Automata

We now define BSP automata as acceptance machines for BSP words.

Definition 2 A BSP automaton **A** is a structure

$$(\{Q^i\}_{i\in[p]}, \Sigma, \{\delta^i\}_{i\in[p]}, \{q^i_0\}_{i\in[p]}, \{F^i\}_{i\in[p]}, \Delta)$$

such that for every i, $(Q^i, \Sigma, \delta^i, q^i_0, F^i)$ is a deterministic finite automaton (DFA),[1] and $\Delta : \mathbf{Q} \to \mathbf{Q}$ is called the *synchronization function* where $\mathbf{Q} = (Q^0 \times ... \times Q^{(p-1)})$ is called the set of *global states*.

In other words a BSP automaton is a vector of sequential automata A^i over the same alphabet Σ, together with a synchronization function that maps state-vectors to state-vectors.

Observe that the synchronization function is finite, like the transition functions, and that its value depends on a whole vector of local states. Because of it, a BSP automaton is more than the *product* [6] of its local automata (see Appendix 1 for an explanation).

Let Q^i be a set of local states at location i, $\delta^i : Q^i \times \Sigma \to Q^i$ a local transition function on those states and $\delta^i * : Q^i \times \Sigma^* \to Q^i$, the extended transition function on Σ-words. Right-application notation is sometimes convenient: $\delta^i * (q, w)$ can be written qw e.g. $qab = \delta(\delta(q, a), b)$.

Define a *transition function* $\boldsymbol{\delta}$ on word-vectors as follows. For $\mathbf{q} \in \mathbf{Q}$ and $\mathbf{w} =< w^0, \ldots, w^{p-1} >$ a word-vector

$$\mathbf{q}w = \Delta(< q^0 w^0, \ldots, q^{p-1} w^{p-1} >) \quad (1)$$

i.e. "synchronization" of the result of application of local transition functions to local words. Function Δ is the model of a synchronization barrier because its local results depend on the whole vector of asynchronous results.

A BSP word is a sequence of word vectors. It is read by a BSP automaton as follows (Fig. 3):

[1] Q^i is the finite set of states, δ the transition function, $q^i \in Q^i$ the initial state and $F^i \subseteq Q^i$ the non-empty set of accepting states.

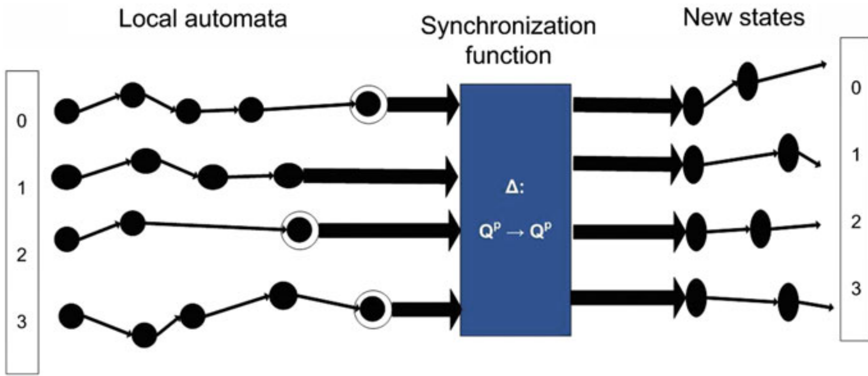

Fig. 3 A BSP automaton

1. If the sequence of word vectors is empty, the vector state remains the vector of local initial states; otherwise continue.
2. If $< w^0, \ldots, w^{p-1} >$ is the first word vector. Local automaton i applies w^i to its initial state and transition function to reach some state q^i, **not necessarily an accepting state**.
3. The synchronization function maps $\Delta :< q^0, \ldots, q^{p-1} > \to < q'^0, \ldots, q'^{p-1} >$.
4. If there are no more word vectors, and $\forall i.\ q'^i \in F^i$, the BSP word is accepted.
5. If there are no more word vectors, and $\exists i.\ q'^i \notin F^i$, the BSP word is rejected.
6. If there are more word vectors, control returns to step 2. but with local automaton i in state q'^i, for every location i.

Finite automata have no explicit notion of variables and values but states can be used to encode them e.g. $q_x 1 = \{(x, 1)\}, q_x 2 = \{(x, 2)\}, \ldots$. As a result, the synchronization function Δ can encode the communication of values between locations i, j, although this is not explicit in the general theory.

Proposition 1 *A BSP automaton is equivalent to a deterministic automaton over (the infinite alphabet of) word-vectors.*

Proof If all p finite automata are deterministic, then the transition function on word-vectors is a total and well-defined function of type $\mathbf{Q} \times (\Sigma^*)^p \to \mathbf{Q}$. The following structure built from $\mathbf{A}$ is a deterministic automaton by construction:

$$(\mathbf{Q}, (\Sigma^*)^p, \boldsymbol{\delta}, < q_0^0, \ldots, q_0^{p-1} >, (F^0 \times \ldots \times F^{p-1})).$$

The automaton is deterministic because $\boldsymbol{\delta}$ is well-defined and total because of 1 and the fact that local automata are deterministic. □

As Proposition 1 states, the BSP automaton is a deterministic automaton but its alphabet is infinite. The synchronization function Δ finite (can be enumerated) but enumerating the transition function $\boldsymbol{\delta}$ is impossible: it is a table over $Q^p \times (\Sigma^*)^p$

whose second component is infinite. So $\boldsymbol{\delta}$ is infinite, but it has an obvious finite representation: the vector of finite transition functions δ^i. As a result, a BSP automaton is practically equivalent to a DFA modulo the above syntactic changes.

Definition 3 As shown in the proof of Proposition 1, a BSP automaton $\mathbf{A}$ is a DFA on word-vectors. A BSP-word $\mathbf{w}$ is *accepted* by $\mathbf{A}$ if the reflexive-transitive closure of $\boldsymbol{\delta}$ takes initial state $\mathbf{q}_0 =< q_0^0, \ldots, q_0^{p-1} >$ to an accepting state of $(F^0 \times \ldots \times F^{p-1})$ when applied to $\mathbf{w}$. The *language* of $\mathbf{A}$ is its set of accepted BSP-words.

We now give BSP automata to recognize BSP languages from Sect. 2.

Let A_a^i and A_b^i be the unique minimal DFA to recognize $\mathbf{v}_a$ and $\mathbf{v}_b$. Define A_a as the BSP automaton $(< A_a^0, A_a^1, A_a^2, A_a^3 >, \Delta)$ where Δ is the identity function. Then, for word-vector $\mathbf{a}$, the local transition functions of A_a will lead to a vector of accepting states, which the synchronization function Δ will leave unchanged. For any other word-vector $\mathbf{w}$, the local transition functions will lead to a vector of non-accepting states, unchanged by synchronization. As a result A_a accepts exactly language $L_a = \{\mathbf{v}_a\}$. A similar construction with letter b gives a BSP automaton A_b to accept L_b.

We now define a BSP automaton to accept

$$L_2 = \{\epsilon, \mathbf{v}_a, \mathbf{v}_b, \mathbf{v}_a\mathbf{v}_b\}.$$

Let A_{a+b}^i be a DFA that accepts language $\{\epsilon, a^i, b^i\}$ with exactly three accepting states: q_0^i initial state for accepting ϵ, q_{Fa}^i for accepting a^i and q_{Fb}^i for accepting b^i. Let $A_{\epsilon+b}^i$ be a DFA that accepts language $\{\epsilon, b^i\}$ with initial (accepting) state q_b^i. Define the BSP automaton as

$$A_{a+b} = (< A_{a+b}^i \cup A_{\epsilon+b}^i : i = 0, 1, 2, 3 >, \Delta)$$

where the local automaton has the union of accepting states and initial state q_0^i. Define also

$$\Delta < q_{Fa}^0, q_{Fa}^1, q_{Fa}^2, q_{Fa}^3 >=< q_b^0, q_b^1, q_b^2, q_b^3 >$$

and Δ is the identity function on all other vector-states. Then A_{a+b} on $\boldsymbol{\epsilon}$ leads to $\Delta(\boldsymbol{\epsilon}) = \boldsymbol{\epsilon}$ which is by definition accepting. Automaton A_{a+b} applied to word-vector $\mathbf{a}$ leads to

$$\Delta < q_{Fa}^0, q_{Fa}^1, q_{Fa}^2, q_{Fa}^3 >=< q_b^0, q_b^1, q_b^2, q_b^3 >$$

an accepting state. Automaton A_{a+b} applied to word-vector $\mathbf{b}$ leads asynchronously to

$$< q_{Fb}^0, q_{Fb}^1, q_{Fb}^2, q_{Fb}^3 >$$

unchanged by Δ and that is an accepting state. Automaton A_{a+b} applied to word-vector $\mathbf{ab}$ leads through $\mathbf{a}$ and synchronization to $< q_b^0, q_b^1, q_b^2, q_b^3 >$ and from there

asynchronously to accepting states of $A^i_{\epsilon+b}$ that the second synchronization preserves. So **ab** is also accepted and it can be checked that any other sequence of word-vectors is not accepted.

5 Non-determinism and Empty Transitions

A non-deterministic finite automaton (NFA) is a finite automaton whose transition function has type $Q \times \Sigma \rightarrow \mathcal{P}(Q)$ i.e. zero, one or more transitions $\delta(q, a)$ can exist for a given symbol a . The closure of its transition function is the union of all possible paths defined by δ for an input word.

A non-deterministic finite automaton with empty transitions (ϵ-NFA) is an NFA over alphabet $\Sigma \cup \{\epsilon\}$ where ϵ does not denote the empty word but a special "internal" symbol that represents "spontaneous" state changes happening without input. The closure of its transition function is the union of all possible NFA transitions on the input word interleaved with an arbitrary number of ϵ symbols.[2]

The languages recognized by NFA and by ϵ-NFA are same regular languages generated by regular expressions and recognized by DFA [15]. This holds because of:

1. a polynomial-time algorithm to remove ϵ-transitions without changing the language, and
2. an exponential-time algorithm to convert an NFA into an equivalent DFA.

The former transformation is called the subset algorithm because it generates a DFA whose states are subsets of the NFA states.

Definition 4 A non-deterministic BSP automaton (NBSPA) is a BSP automaton whose local automata are of type $Q \times \Sigma \rightarrow \mathcal{P}(Q)$ and whose synchronization function has type $\Delta : \mathbf{Q} \rightarrow \mathcal{P}(\mathbf{Q})$.

Definition 5 A non-deterministic BSP automaton with empty transitions (ϵ-NBSPA) is a NBSPA whose local automata are ϵ-NFA (Fig. 4).

Remark that the definition of empty transitions for BSP automata leaves the synchronization function Δ unchanged.

A (standard, deterministic) BSP automaton is by definition a special case of NBSPA and of ϵ-NBSPA but we need to verify whether the latter encode the same class of languages. The answer is positive and given by the next propositions.

[2]This notion of empty transitions is convenient but theoretically delicate. In the case of finite automata it preserves all elementary properties but that is not the case for communicating automata. For example Milner's CCS process algebra [17] uses a spontaneous-transition symbol τ with a similar property, but this changes the so-called bisimulation semantics of *communicating* automata. A more conservative process algebra can be built by replacing τ with an explicit clock-tick symbol Θ [1]. The resulting algebra of processes combines a simple bisimulation semantics with the algebraic simplicity (e.g. distributive law) similar to regular languages.

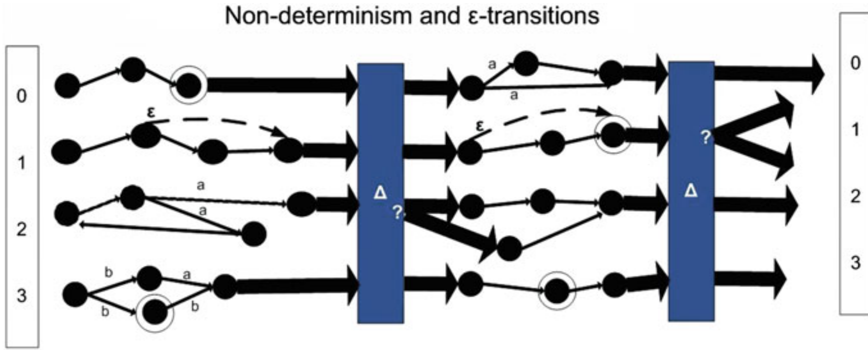

Fig. 4 An ϵ-NBSPA

Proposition 2 *The language of a NBSPA can be accepted by a deterministic BSP automaton.*

Proof Let N be a NBSPA defined by $(< N^0, \ldots, N^{p-1} >, \Delta)$ where the N^i are NFA and $\Delta : \mathbf{Q} \to \mathcal{P}(\mathbf{Q})$. Let Q^i be the set of states of N^i.

By the subset algorithm there exists p DFA D^i accepting the same (scalar) languages as the N^i and whose states are parts of $\mathcal{P}(Q^i)$. Define $\Delta' : \mathcal{P}(\mathbf{Q}) \to \mathcal{P}(\mathbf{Q})$ by

$$\Delta'\{\mathbf{q^1}, \ldots, \mathbf{q^n}\} = \bigcup_{i=1}^{n} \Delta(\mathbf{q^i})$$

so Δ' sends a set of possible vector states to a set of vector states (a non-deterministic choice of synchronization transition). Define D as the deterministic automaton $D = (< D^0, \ldots, D^{p-1} >, \Delta')$. Then we can verify that $L(N) \subseteq L(D)$ by induction on the number of supersteps S in an accepted BSP word.

- $(S = 0)$ If ϵ is accepted by N that is because the initial vector-state in N, $\mathbf{q}^0$ is accepting. By definition of the subset algorithm, the accepting vector state of D is built from local accepting states, and the initial state of D is just $\mathbf{q}^0$. As a result, $\mathbf{q}^0$ is also accepting in D and so D accepts ϵ.
- $(S = 1)$ If a word-vector $\mathbf{w} =< w^0, \ldots, w^{p-1} >$ is accepted by N then one of the paths in N^i applied to w^i leads to a state q'^i such that $\Delta < q'^0, \ldots, q'^{p-1} >$ contains an accepting state-vector. By the subset algorithm, $q'^i \in Q'^i$ where Q'^i is a state of D^i and by the definition of Δ' then $\Delta' < Q'^0, \ldots, Q'^{p-1} >$ contains an accepting state-vector.
- $(S \geq 2)$ If a BSP word $\mathbf{w}^1; \ldots; \mathbf{w}^{n-1}; \mathbf{w}^n$ is accepted by N then N applied to $\mathbf{w}^1; \ldots; \mathbf{w}^{n-1}$ leads to a set of vector-states among which one $\mathbf{Q}$ can be chosen as initial vector state from which N would accept w^n. By construction D contains a vector-state containing $\mathbf{Q}$. Apply then the above one-superstep proof from $\mathbf{Q}$ to show that if N leads to acceptance, so does D. □

Proposition 3 *The language of an ϵ-NBSPA can be recognized by a NBSPA.*

Proof An ϵ-NBSPA N' is simply a non-deterministic BSP automaton built from a vector of ϵ-NFA N'^i. Its synchronization function is non-deterministic but contains no "spontaneous" empty supersteps. Standard automata theory gives us a polynomial-time ϵ-reachability algorithm to convert every N'^i into an equivalent NFA N^i without ϵ-transitions. Define N to be the NBSPA built from the N^i and the same synchronization function as N'. Then $L(N) = L(N')$. □

As a result, non-determinism and ϵ-transitions do not change the languages accepted by BSP automata. Just as with sequential "scalar" automata, those syntactic extensions can be used at an exponential cost in time and number of states. Depending on the complexity of the synchronization functions, the blow-up factor may also depend on p.

6 Sequentialization

Every parallel computation can be simulated sequentially and the theory of BSP automata expresses this fact by a transformation from BSP automata to classical finite automata. A word $u = a_1 \ldots a_n$ of Σ^* is *localized* to i as follows: $u@i = (a_1, i) \ldots (a_n, i)$. A word-vector $\mathbf{w} \in (\Sigma^*)^p$ is *sequentialized* to a word $\mathrm{Seq}(\mathbf{w})$ on alphabet $\Sigma \times [p]$ by the transformation:

$$\mathrm{Seq}(\mathbf{w}) = \mathbf{w}^0 @0 \ldots \mathbf{w}^{p-1} @(p-1).$$

In other words, $\mathrm{Seq}(\mathbf{w})$ concatenates the words of word-vector $\mathbf{w}$ after having labelled them by their locations (any interleaving of the localized words would satisfy our purpose, but ordered concatenation is simpler). For example if $\mathbf{w} =< b, \epsilon, bb, aa >$ then $\mathrm{Seq}(\mathbf{w}) = \mathbf{w}^0 @0 \ldots \mathbf{w}^3 @3 = (b, 0)(b, 2)(b, 2)(a, 3)(a, 3)$.

Definition 6 A BSP word on $(\Sigma^*)^p$ is *sequentialized* to a word on $(\Sigma \times [p]) \cup \{;\}$ as follows (Fig. 5):

$$\begin{aligned} \mathrm{Seq}(\epsilon) &= \epsilon \\ \mathrm{Seq}(\mathbf{v}_1 \ldots \mathbf{v}_n) &= \mathrm{Seq}(\mathbf{v}_1); \ldots ; \mathrm{Seq}(\mathbf{v}_n); \end{aligned}$$

A BSP language L is sequentialized to $\mathrm{Seq}(L)$ by sequentializing every one of its BSP words.

The following remarks should be kept in mind because they are much more than a syntactic detail. 1. The sequentialization of a BSP word is either empty or contains at least one semicolon, and 2. function Seq has one of two possible types.

- To sequentialize word vectors $\mathrm{Seq} : (\Sigma^*)^p \to (\Sigma \times [p])^*$.
- To sequentialize *BSP words* $\mathrm{Seq} : ((\Sigma^*)^p)^* \to ((\Sigma \times [p]) \cup \{;\})^*$.

BSP element: type $\longrightarrow$ local / sequential element
$\epsilon : \Sigma^* \xrightarrow{@i} \epsilon$
$a : \Sigma^* \xrightarrow{@i} (a,i)$
$abaa : \Sigma^* \xrightarrow{@i} (a,i)(b,i)(a,i)(a,i)$
$\boldsymbol{\epsilon} =< \epsilon,\epsilon,\epsilon,\epsilon >: (\Sigma^*)^p \xrightarrow{\text{Seq}} \epsilon$
$\mathbf{v}_1 =< aba, b, bbb, a >: (\Sigma^*)^p \xrightarrow{\text{Seq}} (a,0)(b,0)(a,0)(b,1)(b,2)(b,2)(b,2)(a,3)$
$\mathbf{v}_2 =< a, \epsilon, bbb, \epsilon >: (\Sigma^*)^p \xrightarrow{\text{Seq}} (a,0)(b,2)(b,2)(b,2)$
$\boldsymbol{\epsilon} =< \epsilon,\epsilon,\epsilon,\epsilon >: (\Sigma^*)^p \xrightarrow{\text{Seq}} \epsilon$
$\epsilon : ((\Sigma^*)^p)^* \xrightarrow{\text{Seq}} \epsilon$
$\boldsymbol{\epsilon} =< \epsilon,\epsilon,\epsilon,\epsilon >: ((\Sigma^*)^p)^* \xrightarrow{\text{Seq}} (\epsilon;) \ = \ ;$
$\mathbf{v}_2\ \boldsymbol{\epsilon} : ((\Sigma^*)^p)^* \xrightarrow{\text{Seq}} (a,0)(b,2)(b,2)(b,2);;$
$\boldsymbol{\epsilon}\ \mathbf{v}_2 : ((\Sigma^*)^p)^* \xrightarrow{\text{Seq}} ;(a,0)(b,2)(b,2)(b,2);$
$< \epsilon, a, \epsilon, a >< b, b, b, b >: ((\Sigma^*)^p)^* \xrightarrow{\text{Seq}} (a,1)(a,3);(b,0)(b,1)(b,2)(b,3);$

Fig. 5 Localization and sequentialization

For simplicity we denote both by the same symbol Seq but the first one is only an auxiliary part of the definition of the second one.

So if $\boldsymbol{\epsilon} =< \epsilon, \ldots, \epsilon >$ is considered to be a word-vector, then it is sequentialized to the empty word. But as a BSP word it is sequentialized to the one-symbol word ";" (Definition 6). This is the theoretical representation that even an "empty" BSP algorithm (whose every local process has an empty execution trace) must end by a synchronization barrier that propagates the coherent information "end execution" to every location. In terms of BSP automata this means that even if Δ is the identity function, and it follows a vector of empty computations, it still must be applied once.

A finite automaton $A = (Q, \Sigma, \delta, q_0, F)$ of alphabet Σ can be *localized* to $i \in [p]$ and becomes automaton $A@i$ by the transformation

$$A@i = (Q \times \{i\}, \Sigma \times \{i\}, \delta@i, (q_0, i), F \times \{i\})$$

where $(\delta@i)((q,i),(a,i)) = (\delta(q,a), i)$.

Proposition 4 *For any BSP automaton A on Σ, there exists a finite automaton $Seq(A)$ on $(\Sigma \times [p]) \cup \{;\}$ such that $Seq(L(A)) = L(Seq(A))$.*

Proof Let $A = (< A^0, \ldots, A^{p-1} >, \Delta)$ with $A^i = (Q^i, \Sigma, \delta^i, q_0^i, F^i)$.
Define vector states $Q = \prod_{i=0}^{p-1} Q^i$ for the sequential automaton i.e. all vectors of local states.
Define localized transition function $\delta_a(\mathbf{q}, a) = \mathbf{q}[i{:=}\delta^i(\mathbf{q}_i, a)]$ i.e. the local asynchronous transition at i for any letter a localized at i.
Define a vector of initial state $q_0 =< (q_0^0, 0), (q_0^1, 1) \ldots, (q_0^{p-1}, p-1) >$ with the

local initial states.
Define also the set of unanimously-accepting vector states $F = \prod_{i=0}^{p-1} F^i$.
Then $A_a = (Q, \Sigma \times [p], \delta_a, q_0, F)$ is a DFA that can simulate the application of A to any word-vector $\mathbf{w} =< w^0, \ldots, w^{p-1} >$ as follows.
Let $w = \text{Seq}(\mathbf{w})$ then $\delta_a(q_0, w) =< \delta^0(q_0^0, \mathbf{w}^0), \ldots, \delta^{p-1}(q_0^{p-1}, \mathbf{w}^{p-1}) >$.

As a result, the asynchronous automaton A_a simulates A in the absence of synchronizations. That covers the trivial case of accepting the empty BSP word whose sequentialization is $\text{Seq}(\epsilon) = \epsilon$. Indeed if $\epsilon \in L(A)$ that is because $\forall i. q_0^i \in F^i$ and then by definition $q_0 \in F$ so $\epsilon \in L(A_a)$. But even a single word-vector (single superstep) involves the synchronization function when it is considered as a BSP word.

To simulate its effect with the sequential automaton, transform A_a to a DFA $A_;$ on $(\Sigma \times [p]) \cup \{;\}$ as follows.
Let $\delta_;$ be the extension of δ_a with transitions on symbol semicolon ; that simulate the effect of the synchronization function Δ. For any state vector $\mathbf{q}$ define $\delta_;(\mathbf{q}, ;) = \Delta(\mathbf{q})$. Since the synchronization function is total, this ensures that $\delta_;$ is a total function and that $A_;$ is a DFA. Consider a non-empty BSP word of length one i.e. a word vector $\mathbf{v}$ (which *could* be a vector of empty words). The effect of $A_;$ on $\text{Seq}(\mathbf{v}) = (\text{Seq}(\mathbf{v}));$ is the same as the effect of A on $\mathbf{v}$. Therefore $\mathbf{v} \in L(A)$ iff $\text{Seq}(\mathbf{v}) \in L(A_;)$.

A trivial induction argument shows that this is also the case for a BSP word of any length. We therefore define $\text{Seq}(A) = A_;$ and conclude that

$$\mathbf{v}_1 \ldots \mathbf{v}_n \in L(A) \quad \Leftrightarrow \quad \text{Seq}(\mathbf{v}_1); \ldots ; \text{Seq}(\mathbf{v}_n); \in L(A_;)$$

i.e. $\text{Seq}(L(A)) = L(A_;) = L(\text{Seq}(A))$. □

7 Parallelization

We have seen in Sect. 6 the sequentialization of word-vectors by localization of their words, one symbol at a time $\text{Seq} : (\Sigma^*)^p \to (\Sigma \times [p])^*$. It is easy to invert this transformation and define $\text{Par} : (\Sigma \times [p])^* \to (\Sigma^*)^p$ so that $\text{Par}(\text{Seq}(\mathbf{w})) = \mathbf{w}$.

Let $\boldsymbol{\epsilon}[i{:=}u]$ be the word vector that is empty everywhere except for word u at position i. Let $\mathbf{u} \cdot \mathbf{v}$ be the pointwise concatenation of word-vectors i.e.

$$< u^0, \ldots, u^{p-1} > \cdot < v^0, \ldots, v^{p-1} > =< u^0 v^0, \ldots, u^{p-1} v^{p-1} > .$$

Define $\text{Par} : \Sigma \times [p] \to (\Sigma^*)^p$ by

$$\text{Par}(a, i) = \boldsymbol{\epsilon}[i{:=}a]$$

so that for example $\text{Par}(u@i) = \boldsymbol{\epsilon}[i{:=}u]$. Define then Par on sequentialized words of $(\Sigma \times [p])^*$ by

$$\text{Par}((a, i)(b, j) \ldots) = \text{Par}(a, i) \cdot \text{Par}(b, j) \ldots$$

and in particular $\mathrm{Par}(\epsilon) = \boldsymbol{\epsilon}$ the vector of empty words (or "empty-word vector" not to be confused with the empty BSP word $\epsilon \in (\Sigma^*)^p)^*$). For example

$$\mathrm{Par}((a,0)(b,1)(b,0)(b,3)) =< ab, b, \epsilon, b > .$$

The following follows directly from the definition of Seq on word-vectors.

Lemma 1 *Parallelization is the left-inverse of sequentialization on word-vectors* $(\Sigma^*)^p$*:*

$$Par(Seq(\mathbf{v})) = \mathbf{v}.$$

In fact, any permutation π of $\mathrm{Seq}(\mathbf{w})$ that does not reorder co-located letters would also preserve the parallelization $\mathrm{Par}(\pi(\mathrm{Seq}(\mathbf{w})))$ but we will not expand on this for it is not essential to our developments.

Function Par has one of three possible types.

- To parallelize localized letters $\mathrm{Par} : (\Sigma \times [p]) \rightarrow (\Sigma^*)^p$.
- To parallelize semicolon-free words $\mathrm{Par} : (\Sigma \times [p])^* \rightarrow (\Sigma^*)^p$.
- To parallelize localized words with semicolons $\mathrm{Par} : (\Sigma \times [p]) \cup \{;\})^* \rightarrow ((\Sigma^*)^p)^*$.

Again, this can lead to ambiguity if the input type is unknown: the semicolon-free word is mapped to the empty-word vector, but the empty general word of type $((\Sigma \times [p]) \cup \{;\})^*)$ is mapped to the empty BSP word (Fig. 6). This ambiguity is of course only a convenience for notation but, as we have seen earlier, the difference between empty-word vector and empty BSP word in fundamental.

The following straightforward consequence of our definitions shows that Par is a non-injective function.

Proposition 5 *If w is a word of $(\Sigma \times [p])^*$ and π is a permutation of w that does not exchange co-located letters, then $Par(\pi(w)) = Par(w)$.*

For this reason, Seq is not the left-inverse of Par:

$$\exists w \in (\Sigma \times [p])^*. \ \mathrm{Seq}(\mathrm{Par}(w)) \neq w.$$

local / sequential element: type	$\longrightarrow$	vector/BSP element: type
$(a,1) : \Sigma \times [p]$	$\xrightarrow{\mathrm{Par}}$	$< \epsilon, a, \epsilon, \epsilon >: (\Sigma^*)^p$
$\epsilon : (\Sigma \times [p])^*$	$\xrightarrow{\mathrm{Par}}$	$< \epsilon, \epsilon, \epsilon, \epsilon >: (\Sigma^*)^p$
$(a,1)(b,3)(a,1) : (\Sigma \times [p])^*$	$\xrightarrow{\mathrm{Par}}$	$< \epsilon, aa, \epsilon, b >: (\Sigma^*)^p$
$(a,0)(b,0)(a,0)(b,1)(b,2)(b,2)(b,2)(a,3)$	$\xrightarrow{\mathrm{Par}}$	$< aba, b, bbb, a >: (\Sigma^*)^p$
$(a,0)(b,0)(b,2)(a,3); (a,0)(b,1)(b,2)(b,2);$	$\xrightarrow{\mathrm{Par}}$	$< ab, \epsilon, b, a >< a, b, bb, \epsilon >: ((\Sigma^*)^p)^*$

Fig. 6 Parallelization

For example if $w = (a, 0)(b, 3)(a, 1)$ then $\text{Par}(w) =< aa, \epsilon, \epsilon, b >$ and $\text{Seq}(Par(w)) = (a, 0)(a, 1)(b, 3) \neq w$. But Seq ∘ Par is clearly a normal form for words of $(\Sigma \times [p])^*$: it sorts their letters in increasing order of locations.

Proposition 6 *Reduction to normal form $\cong=$ Seq ∘ Par is a congruence for concatenation on $(\Sigma \times [p])^*$ and $(\Sigma \times [p])^* / \cong$ is isomorphic to $(\Sigma^*)^p$.*

Proof Taking the normal form by $\cong$ preserves the value of Par, and Par is surjective. Taking the i-subword of any $w \in (\Sigma \times [p])^*$ is a homomorphism for concatenation. Therefore Par is a homomorphism from word concatenation to word-vector concatenation. As a result Par is injective on $(\Sigma \times [p])^* / \cong$, surjective and homomorphic. □

Concurrency theories like process algebras [17] ignore the notion of localization and simply consider interleavings π that forget the locations i. That is why they are models of *shared-memory* computers and that was one of the reasons for inventing theories like BSP that do not abstract from distributed-memory.

As the semicolon symbol ; encodes synchronization barriers i.e. the end of supersteps, it is natural to extend parallelization to all words on $(\Sigma \times [p]) \cup \{; \}$.

Definition 7 Let $\alpha = \alpha_0; \ldots; \alpha_n;$ where $\alpha_i \in (\Sigma \times [p])^*$.
Then $\text{Par}(\alpha) = \text{Par}(\alpha_0) \ldots \text{Par}(\alpha_n)$.

For example $\text{Par}((a, 0)(b, 1)(b, 0)(b, 3); (a, 2)(a, 2)(b, 3); ; (a, 0))$ is the BSP word:

$$< ab, b, \epsilon, b >< \epsilon, \epsilon, aa, b >< \epsilon, \epsilon, \epsilon, \epsilon >< a, \epsilon, \epsilon, \epsilon > .$$

The inversion property on word-vectors then follows from our definitions.

Lemma 2 *Parallelization is the left-inverse of sequentialization on BSP words $((\Sigma^*)^p)^*$:*

$$Par(Seq(\mathbf{w})) = \mathbf{w}.$$

The reasoning in the other direction, about Seq ∘ Par applies to BSP words identically and yields the same result as for word vectors (individual BSP supersteps): $\cong=$ Seq ∘ Par sorts inter-semicolon sequences in increasing order of location, it is a congruence for concatenation on $((\Sigma \times [p]) \cup \{; \})^*$ and leads to a parallel-sequential isomorphism.

Reduction to normal form $\cong=$ Seq ∘ Par is a congruence for concatenation on sequential words but with the important exclusion of non-empty semicolon-free sequential words that are meaningless for BSP.

Definition 8 $\Sigma_{p;} = (((\Sigma \times [p])^*);)$

In other words, $\Sigma_{p;}^*$ is the set of sequential localized words, without non-empty semicolon-free words. We find that $\Sigma_{p;}^* / \cong$ is isomorphic to the BSP words $((\Sigma^*)^p)^*$.

Proposition 7 *Reduction to normal form $\cong = Seq \circ Par$ is a congruence for concatenation on $\Sigma^*_{p;}$ and $\Sigma^*_{p;}/\cong$ is isomorphic to $((\Sigma^*)^p)^*$.*

Proof The proof is almost identical to that of Proposition 6. The only (key) difference is that Par would not be a bijection if applied to the whole of $((\Sigma \times [p]) \cup \{;\})^*$. □

Definition 9 The parallelization $\text{Par}(L)$ of a language on $(\Sigma \times [p]) \cup \{;\}$ is $\{\text{Par}(\alpha) : \alpha \in L\}$ and the sequentialization $\text{Seq}(L')$ of a BSP language is $\{\text{Seq}(\mathbf{w}) : \mathbf{w} \in L'\}$.

We now give results about inverting the sequentialization of BSP automata.

The first result is about inverting the sequentialization of "incomplete superstep" BSP words. Such words correspond to sequentialized words on $\Sigma_{p;}$ i.e. words of the form $((\Sigma \times [p])^*);)$. It would appear that such words contain all the necessary information to be recognized by a BSP automaton. One word at a time this is true, but it does not hold of regular languages of this type. Take for example the regular language of expression $((a, 0) + (b, 1))^*$ then its parallelized language can be recognized by a BSP automaton whose language is $< a^*, b^*, \epsilon, \epsilon >$, essentially because the number of a events is independent from the number of b events. But a language like that of $((a, 0)(b, 1))^*$ is parallelized to language $\{< a^n, b^n, \epsilon, \epsilon > \mid n \geq 0\}$ which cannot be recognized by a BSP automaton because the local automata at locations 0 and 1 would need to keep synchronised without the help of the synchronization function. However if the sequentialized language is given extra synchronization semicolons, then it can be recognized by a BSP automaton. In the above example, the language of expression $((a, 0)(b, 1);)^*$ is parallelized to language $\{< a, b, \epsilon, \epsilon >^n \mid n \geq 0\} =< a, b, \epsilon, \epsilon >^*$ for which a BSP automaton exists. The process of adding semicolons to a sequential word or language will be called *over-synchronization*.

Definition 10 For $w \in ((\Sigma \times [p])^*) \cup \{;\}$, we say that w' over-sychronizes w and write $w \leq_; w'$ if w' is obtained by interleaving w with a word of the form $;^*$. A language L' over-synchronizes language L, written $L \leq_; L'$, if there is a bijection from L to L' which is an over-synchronization. An automaton A' over-synchronizes automaton A, written $A \leq_; A'$ if $L(A) \leq_; L(A')$.

Lemma 3 *For any automaton A on $(\Sigma \times [p])$ there is a sequential automaton $A' \geq_;$ A, and a BSP automaton $Par(A)$ on Σ such that $L(Par(A)) = Par(L(A'))$.*

Proof Let r_A be a regular expression such that $L(r_A) = L(A)$ (Appendix 2). We show by induction on the syntax of r_a that there exists a BSP automaton to recognize the parallelization of an over-synchronization of $L(A)$.

If $r_A = \emptyset$ then $\text{Par}(L(r_A)) = \emptyset$ and the BSP automaton can be any one that has empty sets of accepting states. If $r_A = \epsilon$ then $L(r_A) = \{\epsilon\}$ and $\text{Par}(L(r_A)) = \{\epsilon\}$ so the BSP automaton should recognize nothing but the empty BSP word. To obtain this, define its local automata as accepting $\{\epsilon\}$ and the synchronization function is the identity.

If $r_A = r_0^*$ then by induction there is a BSP automaton A_0 to recognize $\mathrm{Par}(L(r_0))$. Add new unique accepting states q_F^i to its local automata and ϵ-transitions from their (previously) accepting states to the q_F^i. Add to A_0's synchronization function the mapping from $< q_F^0, \ldots, q_F^{p-1} >$ to the initial states of all finite automata. Call this new ϵ-NBSPA A_1. Then $L(A_1) = \mathrm{Par}(L((r_0;)^*))$ i.e. the over-sychronization $(r_0;)^*$ has A_1 as accepting BSP automaton.

If $r_A = r_1 + r_2$ then by induction there are BSP automata A_j and $r_j' \geq_; r_j$ such that $L(A_j) = \mathrm{Par}(L(r_j'))$ for $j = 1, 2$. Then build an NBSPA A_0 whose local automata have: the union of local states from A_1, A_2 with an added new initial state with an ϵ-transition leaving it to each of the (previously) initial states from A_1^i and A_2^i, transition function that are the union of local transition functions from A_1, A_2 and a new final state q_F^i. The synchronization function of A_0 is the union of synchronization functions of A_1, A_2 with the added mappings from all state vectors that are uniformly $(\forall i)$ accepting for r_1' or uniformly accepting for r_2' to $< q_F^0, \ldots, q_F^{p-1} >$. Then $L(A_0) = L(A_1) \cup L(A_2) = \mathrm{Par}((r_1' + r_2');)$ which is an over-synchronization of r_A.

If $r_A = r_1 r_2$ then a similar construction leads to a BSP automaton accepting the parallelization of $r_1; r_2$. □

The second result follows about inverting the sequentialization of all BSP words.

Theorem 1 *For any automaton A on* $(\Sigma \times [p]) \cup \{;\}$ *there is a sequential automaton* $A' \geq_; A$, *and a BSP automaton* $Par(A)$ *on* Σ *such that* $L(Par(A)) = Par(L(A'))$.

Proof Let r_A be a regular expression such that $L(r_A) = L(A)$ (Appendix 2). We will show by induction on the syntax of r_A that there exists a BSP automaton $\mathrm{Par}(A)$ to recognize $\mathrm{Par}(L(r_A)) = \mathrm{Par}(L(A))$.

If $r_A = \emptyset$ then $\mathrm{Par}(L(r_A)) = \emptyset$ and the BSP automaton can be any one that has empty sets of accepting states.

If $r_A = \epsilon$ then $L(r_A) = \{\epsilon\}$ and $\mathrm{Par}(L(r_A)) = \{\epsilon\}$ so the BSP automaton should recognize nothing but the empty BSP word. To obtain this, define its local automata as accepting $\{\epsilon\}$ and the synchronization function can be arbitrary because it does not get applied on the empty BSP word.

It is not possible to have $r_A = (a, i)$ then it is easy to build a BSP automaton to recognize $\mathrm{Par}(\{(a, i);\}$.

If $r_A =;$ then $L(r_A) = \{;\} = \{\epsilon;\}$ so $\mathrm{Par}(L(r_A)) = \{\mathrm{Par}(\epsilon)\} = \{\boldsymbol{\epsilon}\}$. Define then A^i as a finite automaton with initial state q_0^i, a single accepting state equal to q_0^i and transition function to accept only the empty word. Let $\mathbf{q}_0 =< q_0^0, \ldots, q_0^{p-1} >$ and define $\Delta(\mathbf{q}_0) = \mathbf{q}_0$ and a different value of $\Delta(\mathbf{q})$ for all other $\mathbf{q}$. Define the BSP automaton $\mathrm{Par}(A) = (< A^0, \ldots, A^{p-1} >, \Delta)$. Then applying $\mathrm{Par}(A)$ to $\boldsymbol{\epsilon}$ leads to $\mathbf{q}_0$ vacuously on the local automata and then by one application of Δ, so $\boldsymbol{\epsilon}$ is accepted. Applying $\mathrm{Par}(A)$ to any other BSP word leads to non-acceptance so $L(\mathrm{Par}(A)) = \{\boldsymbol{\epsilon}\}$.

If $r_A = r_0^*$, $r_A = r_1 + r_2$ or $r_A = r_1 r_2$ then the corresponding induction steps used in the proof of Lemma 3 apply directly. □

Moreover, as seen in Sect. 5 there exists a deterministic BSP automaton A equivalent to the ϵ-NBSPA constructed in the proof of Theorem 1.

8 Bulk-Synchronous Regular Expressions

In this section it is shown how to adapt regular expressions (Appendix 2) to BSP languages.

A BSP regular expression is an expression R from the following grammar:

$$R \ ::= \ \emptyset \mid \epsilon \mid < r^0, \ldots, r^{p-1} > \mid R; R \mid R* \mid R + R$$

where r^i is any (scalar) regular expression. The set of BSP regular expressions is BSPRE and the language any BSP regular expression is defined by $L : \text{BSPRE} \to \mathcal{P}(((\Sigma^*)^p)^*)$ as:

R	$L(R)$
$\emptyset$	$\{\ \}$
ϵ	$\{\epsilon\}$
$< r^0, \ldots, r^{p-1} >$	$L(r^0) \times \ldots \times L(r^{p-1})$
$R_1; R_2$	$L(R_1)L(R_2)$
R^*	$L(R)^*$
$R_1 + R_2$	$L(R_1) \cup L(R_2)$

We now show that Kleene's equivalence theorems (Appendix 2 and [10]) can be adapted to the two-level BSP regular expressions and automata.

Theorem 2 *For $R \in BSPRE$ there exists a BSP automaton A_R such that $L(A_R) = L(R)$.*

Proof We proceed by induction on the syntax of R. If $R = \emptyset$ the BSP automaton simply needs to have empty (local) sets of accepting states. If $R = \epsilon$ the BSP automata should have as unique accepted BSP word the empty one. That is obtained by having accepting (local) start states and all transitions leading to different (non-accepting states), with an indentity synchronization function.

If $R =< r^0, \ldots, r^{p-1} >$ then there exist classical automata A^i on Σ such that $L(A^i) = L(r^i)$ (Appendix 2). The BSP automaton is then simply the collection of those automata with identity synchronization function.

If $R = R_1; R_2$ then by induction there exists BSP automata A_1, A_2 such that $L(A_j) = L(R_j)$ for $j = 1, 2$. Define the BSP automaton A whose states is the disjoint union of those of the A_j, whose accepting states are those of A_2, whose initial vector state is that of A_1, whose (partial) Δ_A is the union of the synchronization functions of the A_j with an added ϵ-transition from all accepting state vectors in A_1 to the (previously) initial state vector of A_2. The resulting A is an ϵ-NBSPA accepting language $L(A_1)L(A_2) = L(R)$.

If is of the form $R = R_0^*$ or $R = R_1 + R_2$, similar constructions lead to ϵ-NBSPA whose language is R. □

Theorem 3 *For A a BSP automaton there exists $R_A \in BSPRE$ such that $L(R_A) = L(A)$.*

Proof Assume

$$A = (\{Q^i\}_{i\in[p]}, \Sigma, \{\delta^i\}_{i\in[p]}, \{q_0^i\}_{i\in[p]}, \{F^i\}_{i\in[p]}, \Delta) \text{ and } A^i = (Q^i, \Sigma, \delta^i, q_0^i, F^i).$$

Let $Q = \bigcup_i Q^i$ be the union of all states in the local automata, then the states $\mathbf{Q}$ of A are all in Q^p. Similarly, let $\mathbf{F} \subseteq \mathbf{Q}$ be the accepting states of Q.

Let $q_1, q_2 \in Q$ be any local states. Then by Kleene's theorem there exists $r(q_1, q_2)^i \in$ RE such that $L(r(q_1, q_2)^i)$ is the set of Σ words that lead from q_1 to q_2 with δ^i. Let RE_A be the *finite set* of all such regular expressions over all pairs of states and all location $i \in [p]$. Let $\Sigma_A = (\text{RE}_A)^p \cup \{;\}$, a large but finite "alphabet" for the following construction of a "vector-automaton" $\mathbf{A}$ equivalent to A. Define $\Delta_A : Q \times \Sigma_A \to Q$ by $\Delta_A(\mathbf{q}_1, \mathbf{r}) = \mathbf{q}_2$ where $\mathbf{q}_1, \mathbf{q}_2$ are vectors of states linked at every location by the local projection of $\mathbf{r}$. Define also $\Delta_A(\mathbf{q}_1, ;) = \mathbf{q}_2$ iff $\Delta(\mathbf{q}_1) = \mathbf{q}_2$. Define at last $\mathbf{A} = (\mathbf{Q}, \Sigma_A, \Delta_A, \mathbf{F})$.

This NFA can be applied to BSP words by applying the vectors of regular expressions to the word vectors pointwise, and traversing any semicolon-edge when there is a change of word. By defining transition in this manner, $\mathbf{A}$ is an acceptance mechanism for BSP word whose accepted language is precisely $L(A)$.

By Kleene's theorem applied to $\mathbf{A}$, there is a (normal) regular expression built from alphabet $\Sigma_A = (\text{RE}_A)^p \cup \{;\}$ whose language is $L(\mathbf{A}) = L(A)$. By construction, such a regular expression is precisely a BSPRE whose language is that of A. □

It is convenient to write $r = r'$ in *RE* (respectively $R = R'$ in BSPRE) when the two regular expressions (resp. BSP reg. expr.) have the same language.

Proposition 8 *(Sect. 9.3.1 of [15]) For $r, r_1, r_2, r_3 \in RE$:*
$\epsilon r = r\epsilon = r \quad r_1(r_2r_1)^* = (r_1r_2)^*r_1$
$\emptyset r = r\emptyset = \emptyset \quad (r_1 \cup r_2)^* = (r_1^*r_2^*)^*$
$\epsilon^* = \emptyset^* = \epsilon \quad r_1(r_2 \cup r_3) = r_1r_2 \cup r_1r_3$

Classical equivalences such as the above hold also for BSP regular expressions R, R_1, R_2, R_3 because they involve no interactions between the two levels of BSP syntax.

9 Minimization

For a given DFA A there exists a so-called *minimal* DFA M_A [21]: $L(M_A) = L(A)$, the number of states of M_A is minimal amongst all automata of equal language. Moreover the minimal automaton M_A is unique: it is isomorphic to any other M'_A of equal language and of the same size. The computation $A \mapsto M_A$ is called *minimization* and can be realized by sequential algorithms of worst-case quadratic time in the number of states of A.

Let us recall the state congruence relation used for the invariant of those algorithms, and its very compact formulation by Benzaken (Chap. 2, Sect. 6.3 of [2]).

Definition 11 Let $A = (Q, \Sigma, q_0, \delta, F)$ be a DFA and $k \geq 0$ an integer. For $q \in Q$, define A_q to be the language accepted by A starting from q i.e. $A_q = L(A[q_0{:=}q])$. For $p, q \in Q$ define $p \simeq_k q$ or "p, q are k-equivalent" to mean $L_k(A_p) = L_k(A_q)$ where $L_k(_)$ denotes the sub-language of words no longer than k.

Then k-equivalence $\simeq_k$ is clearly an equivalence relation on Q and:

- $\simeq_0$ has only two equivalence classes of states, namely F and $Q - F$: $L_0(p) = L_0(q)$ precisely when the empty word is accepted from either state. That is true when both are in F and false otherwise.
- $p \simeq_{k+1} q$ iff $(p \simeq_0 q)$ and $\forall a \in \Sigma.(\delta(p, a) \simeq_k \delta(q, a))$: two states define the same language of length $\leq k + 1$ iff 1 they are both accepting/non-accepting and 2 any pair of transitions $\delta(p, a)$ and $\delta(q, a)$ leaving them on the same symbol a, leads to k-equivalent states.

By definition, $(k + 1)$-equivalence is a (non-strict) refinement of k-equivalence, so its equivalence classes $Q/\simeq_{k+1}$ are obtained by splitting some equivalence classes of $Q/\simeq_k$. Moreover, if for some k we have $Q/\simeq_{k+1} = Q/\simeq_k$ then all $Q/\simeq_i$ are equal for $i = k, k + 1, k + 2 \ldots$. Observe then that in the series of partitions $Q/\simeq_0, Q/\simeq_1, \ldots Q/\simeq_i \ldots$, the number of equivalence classes is non-decreasing, yet by definition it cannot be greater than the number of states $\mid Q \mid$. It follows that $A_p = A_q$ iff $p \simeq_{|Q|} q$ and it can be proved that $A/\simeq_{|Q|}$ is the unique minimal DFA equivalent to A. Sequential algorithms for computing it can be derived from the above construction, among them Hopcroft's algorithm [8] of time complexity $O(n \log n)$ where $n = \mid Q \mid$.

The above ideas have been generalized by D'Antoni and Veanes to so-called symbolic finite automata (SFA) whose alphabets are logical formulae rather than elementary letters [5]. They generalize the above DFA minimization method to SFA and find that the key requirement is to check rapidly for satisfiability of $\phi \wedge \psi$ when considering transitions of the form $\delta(p, \phi)$ and $\delta(q, \psi)$. From the construction used in the proof of Theorem 3 it appears that our BSP automata are a special case of SFA and that those results [5] apply to them. But we will not make use of this general result for the sake of simplicity and to keep this paper self-contained. Efficient algorithms for BSP automata will also benefit from our elementary presentation that only builds from DFAs, REs and vectors, as there is no guarantee that excessively general methods lead to efficient algorithms for the class of BSP automata.

Let Min be the minimization function on DFA that results from applying Hopcroft's algorithm. Observe that it is not sufficient to minimize a BSP automaton by minimizing its local automata: we must account for the synchronization function. The states of a BSP automaton are the state-vectors of $\prod_i Q^i$. But if we apply the classical method to state-vectors and alphabet $\Sigma_{p;}$ then all minimization properties and methods apply.

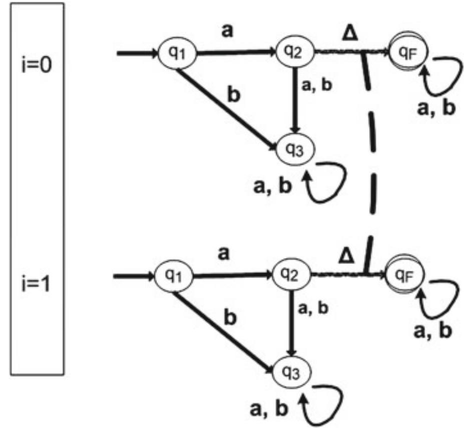

Fig. 7 Automaton $\mathbf{A}_a$

Proposition 9 *If A is a deterministic BSP automaton on Σ then there exists a sequential automaton Min(Seq(A)) that accepts the same Seq(L(A)) and is of minimal size.*

Proof Consider A as a special notation for Seq(A): an automaton on $(\Sigma \times [p]) \cup \{;\}$ i.e. with vector-states but single-symbol local transitions or global transitions on ; defined by the synchronization function Δ. Then clearly A is deterministic so it is a DFA that can be minimized. Apply sequential minimization to obtain the result. □

Minimizing BSP automata is considerably more complex than minimizing DFA. The reason is that pointwise minimization of the local automata, without reference to the synchronization function, may change the accepted BSP language. Let us illustrate this property by an example.

Example 1 Let A_a be the DFA with four states q_1, q_2, q_3, q_F, initial state q_1, unique accepting state q_F, and transitions as shown in Fig. 7 (ignoring Δ). Then clearly $L(A_a) = \emptyset$ because q_F is unreachable from q_1.

Define also the BSP automaton $\mathbf{A}_a = (< A_a, \ldots, A_a >, \Delta)$ with the following synchronization function:

$\mathbf{q}$		$\Delta(\mathbf{q})$
$\mathbf{q}_2 =< q_2, \ldots, q_2 >$	$\xrightarrow{\Delta}$	$\mathbf{q}_F =< q_F, \ldots, q_F >$
$\mathbf{q}_F =< q_F, \ldots, q_F >$	$\xrightarrow{\rightarrow}$	$\mathbf{q}_{1F} =< q_1, q_F, \ldots, q_F >$
any other $\mathbf{q}$	$\xrightarrow{\rightarrow}$	$\mathbf{q}_{1F} =< q_1, q_F, \ldots, q_F >$

Since $\mathbf{q}_F$ is the only accepting vector state for $\mathbf{A}_a$, and since the initial state is $< q_1, \ldots, q_1 >$ it follows that the empty BSP word is not accepted by $\mathbf{A}_a$. Any BSP word of $L(\mathbf{A}_a)$ is therefore of length one or more, so must trigger one or more applications of Δ. By definition, the only such application leading to acceptance is $\Delta(\mathbf{q}_2)$. By definition of A_a, the only word-vectors leading to $\mathbf{q}_2$ is $\mathbf{a} =< a, a, \ldots, a >$. So the BSP word $\mathbf{a} \in L(\mathbf{A}_a)$.

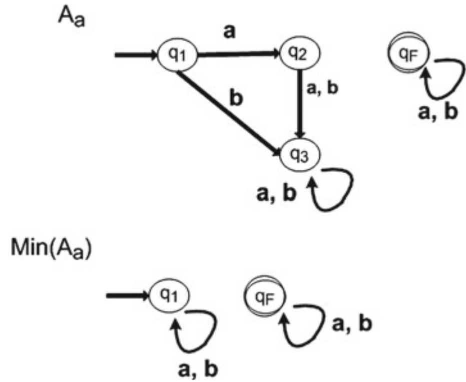

Fig. 8 Locally minimal automaton $\mathrm{Min}(A_a)$

Any longer BSP words are not accepted, because 1 by definition of A_a, local transitions will only lead from $\mathbf{q}_F$ to itself and 2 synchronization Δ will then lead to $\mathbf{q}_{1F}$ which is not accepted, and similarly for a BSP word of length more than two.

As a result $L(\mathbf{A}_a) = \mathbf{a} =< a, a, \ldots, a >$.

Consider now *local* minimization of the BSP automaton $\mathbf{A}_a$ of Example 1. That yields the BSP automaton $(< \mathrm{Min}(A_a), \ldots, \mathrm{Min}(A_a) >, \Delta)$ where $\mathrm{Min}(A_a)$ is the minimal DFA for accepting the empty language i.e. the two-state DFA of Fig. 8. Local state q_1 in $\mathrm{Min}(A_a)$ is actually the equivalence class $\{q_1, q_2, q_3\}$ in A_a so the synchronization function would send $< q_1, \ldots, q_1 >$ to $\Delta(q_2, \ldots, q_2 >) = \mathbf{q}_F$ so that any BSP word of length one would be accepted. The result would then be a BSP automaton whose language is $< (a+b)^*, \ldots, (a+b)^* > \neq L(\mathbf{A}_a)$. The above remarks show that local minimization alone does not preserve the BSP language.

The application of Min $\circ$ Seq as in Proposition 9 has a disadvantage: it produces an automaton whose parallelization is not obvious. Sequentialization can then be reversed but only at the cost of over-synchronization (by Theorem 1).

In other words, if we apply Min $\circ$ Seq and then Par $\circ \leq_;$ in the hope of minimization, the resulting BSP automaton may have a reduced number of states but an increased number of synchronizations. In practical terms that means that the BSP automaton's implementation will consume less space, and process BSP words in the same number of local transitions, but require an increased number of global barriers. Proposition 9 is thus a first but insufficient step towards BSP automata minimization.

Figure 9 illustrates the minimization of $\mathrm{Seq}(\mathbf{A}_a)$ (with $p = 2$, sufficient for illustrating the computation). The circled groups of state vectors are provisional congruence classes to be refined until the Min algorithm reaches its fixed point. They strongly depend on the structure of Δ and as we have seen, the resulting (sequential) automaton on $\Sigma_{p;}$ can only be re-parallelized at the expense of extra synchronizations.

More important for our purpose, the objective of bulk-sychronous parallelism is to provide realistic and predictable *parallel speed-up*. BSP theory includes a cost-model that relates sequential time, the number of processes p and global synchronization

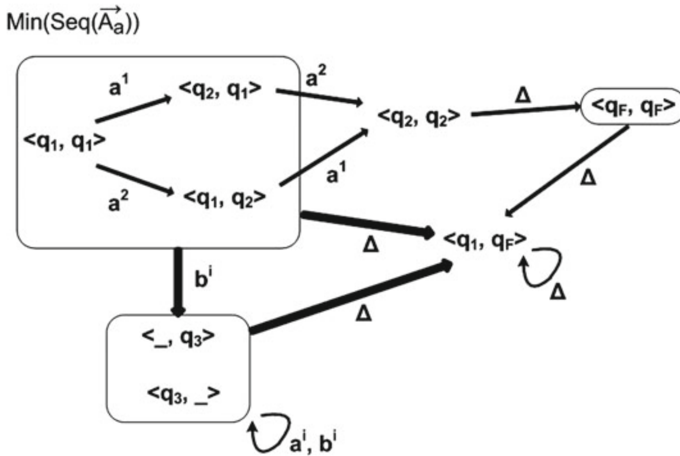

Fig. 9 Sequential minimization of BSP automaton $\mathbf{A}_a$

(and communication) delays. Automaton minimization is directly related to space complexity, memory consumption, but as seen in this section it can lead to higher synchronization costs hence more time complexity. As a step in this direction, we now adapt the cost-model of BSP theory to BSP automata and show how it can be used as objective function in the search for fast parallel versions of sequential automata.

10 Cost-Model

Words from regular languages can be recognized by (classical) finite automata in time proportional to their length. Being models of parallel algorithms, BSP automata are meant to accelerate this process. Ideally a word from a regular language could be recognized p times faster by a BSP automaton. This is certainly possible but, in general, parallel recognition requires more than one superstep so that the BSP automaton's operations require a BSP word of length more than one. Moreover, BSP theory and systems have show that the synchronization function Δ's implementation incurs costs that may be larger than the speed-up of parallelism.

In this section we show how to accelerate the recognition of regular languages, and define a detailed version of the BSP cost model to quantify the time-space cost of doing this.

The first auxiliary notion is concatenation-factorization on sequential words.

Definition 12 A *factorization function* on Σ words is a function $\Phi : \Sigma^* \to (\Sigma^+)^*$ such that

$$\begin{array}{l} \Phi(\epsilon) = \epsilon \\ |w| > 0 \Rightarrow |\Phi(w)| > 0 \\ \Phi(w) = w_1, w_2 \ldots, w_n \Rightarrow w_1 w_2 \ldots w_n = w \end{array}$$

By definition, a factorization function sends the empty word to itself and sends a non-empty w to a non-empty sequence of non-empty words whose concatenation is w itself.

Next we define the distribution of sequential words to (BSP) locations. Recall that $(\Sigma_{p;})^*$ is the set of sequentialized BSP words $((\Sigma \times [p])^*;)^*$.

Definition 13 Given a factorization function Φ on Σ words, a *distribution function* based on Φ is a $D_\Phi : \Sigma^* \to (\Sigma_{p;})^*$ such that

$$\begin{array}{l} D_\Phi(\epsilon) = \epsilon \\ \Phi(w) = w_1, w_2 \ldots, w_n \Rightarrow D_\Phi(w) = w'_1; w'_2; \ldots w'_n; \\ w_t = a_1 \ldots a_k \Rightarrow w'_t = (a_1, i_1) \ldots (a_k, i_k) \\ i_1, \ldots, i_k \in [p] \end{array}$$

The distribution of a language on Σ is the set of distributions of its words i.e. $D_\Phi(L) = \{D_\Phi(w) \mid w \in L\}$.

This definition is such that a distribution $D(w)$ is the sequential image of a BSP word and $\text{Seq}(\text{Par}(D_\Phi(w))) \cong D_\Phi(w)$ (Sect. 7).

For example if

$$w = aaabba$$

one possible factorization is

$$\Phi(w) = aaab, ba$$

and one possible associated distribution is

$$D_\Phi(w) = (a, 3)(a, 2)(a, 2)(b, 0); (b, 1)(a, 1);$$

with

$$\text{Par}(D_\Phi(w)) =< b, \epsilon, aa, a >< \epsilon, ba, \epsilon, \epsilon >$$

and

$$\text{Seq}(\text{Par}(D_\Phi(w))) = (b, 0)(a, 2)(a, 2)(a, 3); (b, 1)(a, 1); \cong D_\Phi.$$

The definition of distribution function is flexible enough to allow any word and any language to be distributed to a BSP word or BSP language. The existence of distributions is a trivial fact of no interest in itself. What matters is optimization: the discovery of distributions with minimal parallel execution time. To define this we need to define the cost of a BSP automaton's computations. The synchronization

cost is an experimental constant that depends on the physical machine executing one of our BSP automata.

Definition 14 Let $\mathbf{v} \in (\Sigma^*)^p$ be a word vector. Its *BSP cost* $\text{cost}(\mathbf{v}) = \max_i \left|v^i\right|$ is the length of its longest element. Define also $l \in \mathbf{N}^+$, the *barrier synchronization cost* constant. For a BSP word $w = \mathbf{v}_1 \ldots \mathbf{v}_S \in ((\Sigma^*)^p)^*$, its BSP cost is

$$\text{cost}(w) = \Sigma_{t=1}^{S}(\text{cost}(\mathbf{v}_t) + l) = Sl + \Sigma_{t=1}^{S}\text{cost}(\mathbf{v}_t).$$

The reader familiar with BSP theory will have noticed that our cost function covers local sequential computation and global synchronization but not communication. This is indeed a simplification and assumes, not that communication is "free" but that an implementation always uses all-to-all communications and that its usual BSP cost of $p \times g$ is here hidden in the l constant.

More detailed presentations of BSP automata will refine this, for example by taking into account the actual dependencies in the synchronization function: a purely local Δ actually costs less than one whose values (output states) depend on all the input states. The above-defined cost model is a pessimistic upper-bound for this. We now explain how BSP automata encode the elements of BSP algorithm design namely load balancing and minimal synchronization.

Definition 15 For a given distribution function D_Φ of factorization Φ, the BSP cost of a sequential word $w \in \Sigma^*$ with respect to D_Φ is defined as the BSP cost of the parallelization of its distribution:

$$\text{cost}_{D_\Phi}(w) = \text{cost}(\text{Par}(D_\Phi(w)))$$

For example in the above example with $w = aaabba$ we had

$$\text{Par}(D_\Phi(w)) =< b, \epsilon, aa, a >< \epsilon, ba, \epsilon, \epsilon >$$

and so

$$\text{cost}_{D_\Phi}(w) = \text{cost}(\text{Par}(D_\Phi(w))) = 2 + l + 2 + l = 4 + 2l.$$

A direct consequence of the cost model is that the cost of a word with respect to D_Φ is least if the factorization $\Phi(w)$ produces a minimal number of factors (hence minimal number of BSP supersteps) while the distribution of each factor $D_\Phi(w_t)$ has the least maximal local length (hence the most balanced distribution). This bi-objective cost function is the basis of BSP algorithm design: for a given amount of parallelism, balance the lengths of local computations while minimizing the number of supersteps.

Problem 1 BSP-PARALLELIZE-WORDWISE
Input: *A regular language L given by a regular expression r or DFA A.*
Goal: *Find a distribution* D_Φ *and BSP automaton* A_D *such that* $L(A) = Par(D_\Phi(L))$ *and* $|A_D| \in O(|A|)$.

Subject to: $\forall w \in \Sigma^*.\ cost_{D_\Phi}(w)$ *is minimal over* $\{(\Phi, D_\Phi, A_D) \mid L(A) = Par(D_\Phi(L))\}$.

Minimization for every individual w is not a standard formulation. A better one is:

Problem 2 BSP-PARALLELIZE
Input: *A regular language L given by a regular expression or DFA.*
Goal: *Find a distribution* D_Φ *and BSP automaton* A_D *such that* $L(A) = Par(D_\Phi(L))$ *and* $|A_D| \in O(|A|)$.
Subject to: $T_{D_\Phi}(n) = \max\{cost_{D_\Phi}(w) \mid |w| = n\}$ *is minimal over* $\{(\Phi, D_\Phi, A_D) \mid L(A) = Par(D_\Phi(L))\}$*, for all* $n \geq 0$.

Theoretical work can concentrate on $\lim_{n\to\infty} T_{D_\Phi}(n)$ while certain applications could consider only fixed-size input words i.e. a single value of n. The former is clearly a general algorithm-design problem and the latter is more likely to have an algorithmic solution. The present formulation of BSP automata leave open both theoretical and practical explorations: depending on the space of factorization and distribution functions that is considered, the BSP-PARALLELIZE problem could have widely different complexities.

In the next section we explore an important subproblem: finding BSP automata parallelizations for the block-wise distribution function $D_{\div p}$. The cost is then equal to l times the number of supersteps and BSP-PARALLELIZE amounts to minimizing the number of supersteps. But as specified in the problem definition ($|A_D| \in O(|A|)$) this should not be at the cost of an explosion in the number of states. We present elements of both lower- and upper-bound for this parameter.

11 Parallel Acceleration

Problem BSP-PARALLELIZE sets the goal of finding the fastest possible tuple (factorization, distribution, BSP automaton) of dimension p to recognize a given regular language L. *Fastest* refers to the cost of the BSP words once they are factorized and distributed for the BSP automaton. As a first step towards such optimal solutions, we will adapt the experimental notion of parallel speedup and show some parallelizations measured thus.

Definition 16 Let L be a regular language and (Φ, D_Φ, A_D) a factorization, distribution and BSP automaton for L i.e. $\text{Par}(D_\Phi(L))$. The parallel *speedup* obtained by (Φ, D_Φ, A_D) on a given word size n is the ratio

$$\text{speedup}(\Phi, D_\Phi, A_D, n) = \min\{n/\text{cost}_{D_\Phi}(w) \mid |w| = n\}$$

The n term in the denominator is $|w|$, the cost of sequential recognition by a DFA. On first inspection, the definition of speedup does not appear to depend on the language L being recognized. But it actually does. A speedup value is only possible by virtue

of a BSP automaton recognizing L with the given factorization (supersteps) and distribution (data placement).

We take three examples of simple regular languages to parallelize.

- $L_1 = L(a^*)$
- $L_2 = L(a^*b^*)$
- $L_3 = L((a+b)^*bbb(a+b)^*)$

Example 2 **Parallel recognition of** L_1. Sequential recognition of a^* amounts to reading a word $w \in \Sigma^*$ sequentially with a DFA for this language. The simplest DFA A_{a^*} has two states q_1, q_2, starts from q_1, accepting states $F = \{q_1\}$, and transitions $\delta(q_2, x) = q_2, \delta(q_1, a) = q_1, \delta(q_1, b) = q_2$.

A simple and efficient parallelization for L_1 is $(\Phi_1, D_\circ, A_1)$ defined as follows. The factorization function keeps the input word into a single superstep word: $\Phi_1(w) = (w)$.
The "remainder p" distribution function sends letters to locations in cyclic fashion: $D_\circ(u_0 \dots u_{n-1}) = (u_0,\ 0 \mod p) \dots (u_{n-1}, (n-1) \mod p)$;
The BSP automaton $A_1 = (< A_{a^*}, \dots, A_{a^*} >, \mathrm{Id})$ has a copy of the DFA for accepting a^* at every location so any input word containing letter b will put one location into non-accepting local state. The synchronization function is the identity on state vectors. As a result, the BSP words accepted are those that only contain letter a, i.e. $L(A_1) = \mathrm{Par}(D_\circ(L))$.

By construction $\mathrm{cost}_{D_\circ}(w) = \mathrm{cost}(w) + l$ i.e. the cost of the distributed word vector + the cost of one barrier. Cyclic distribution is known to have cost n/p or $(n/p)+1$ because no location receives more than that many of the letters. As a result the speedup is $\frac{n}{(n/p)+l}$ which tends to p (ideal speedup) for large input sizes.

The above construction is an asymptotic solution to BSP-PARALLELIZE for this language because any BSP automaton costs one l term on non-empty input, and processing the whole input word is both necessary and requires at least parallel cost n/p.

Example 3 **Parallel recognition of** L_2. Sequential recognition of a^*b^* is done by a DFA having states q_0, q_1, q_2 of which q_0 is initial, accepting states q_0, q_1 and transition function $q_0 \xrightarrow{a} q_0, q_0 \xrightarrow{b} q_1$,
$q_1 \xrightarrow{b} q_1, q_1 \xrightarrow{a} q_2$,
$q_2 \xrightarrow{x} q_2$.
Let us call this automaton A_2.

Consider again parallelization with factorization function Φ_1 i.e. BSP words of length one superstep. Take as distribution function the "div-p" function that sends to each location a block of length $k \geq p$ except one possibly shorter block at the end: $D_{\div p}(u_0u_1 \dots u_{n-1}) = (u_0, 0/p)(u_1, 1/p) \dots (u_{n-1}, (n-1)/p)$
For example

$$D_{\div 4}(u_0u_1 \dots u_8) = (u_0, 0)(u_1, 0)(u_2, 0)(u_3, 0)(u_4, 1)(u_5, 1)(u_6, 1)(u_7, 1)(u_8, 2).$$

We now show that a BSP automaton can be built for Φ_1 and $D_{\div p}$ to accept L_2. Its parallel speedup will then be the same as in Example 2. Consider the BSPRE
$R = R_0 + R_1 + R_2 + R_3$ where
$R_3 =< a^*, a^*, a^*, a^*b^* >$,
$R_2 =< a^*, a^*, a^*b^*, b^* >$,
$R_1 =< a^*, a^*b^*, b^*, b^* >$,
$R_0 =< a^*b^*, b^*, b^*, b^* >$.
By construction $L(R) = \mathrm{Par}(D_{\div p}(L_2)$ because the words of L_2 split into four equal-length blocks and parallelized are precisely of one of the four forms specified by R.

It is not sufficient for our purpose to apply Theorem 2 to R because the constructive proof given there introduces unnecessary synchronization. To obtain a one-superstep BSP automaton for L_2 based on R proceed as follows. Build a DFA A'_2 with states q_0, q_1, q_2, q_3 such that q_0, q_1, q_3 are accepting, state q_0 accepts $L(a^+)$, state q_1 accepts $L(a^+b^*)$, state q_3 accepts $L(b^*)$ and words leading to q_2 are not from the union of those languages (which equals L_2). Let the BSP automaton have A'_2 as local automaton at every location and define the synchronization function as follows:
$\Delta(< (q_0 + q_1), (q_0 + q_1), (q_0 + q_1), (q_0 + q_1) >)$ = an accepting vector state; to accept R_3,
$\Delta(< (q_0 + q_1), (q_0 + q_1), (q_0 + q_1), q_3 >)$ = an accepting vector state; to accept R_2,
$\Delta(< (q_0 + q_1), (q_0 + q_1), q_3, q_3 >)$ = an accepting vector state; to accept R_1,
$\Delta(< (q_0 + q_1), q_3, q_3, q_3 >)$ = an accepting vector state; to accept R_0,
Δ sends any other state vector to a non-accepting vector state.
It then follows that the BSP automaton accepts L_2 in one superstep for the given distribution. This completes the example.

Example 4 **Parallel recognition of** L_3. Sequential recognition of $(a + b)^*bbb(a + b)^*$ amounts to searching for the first sequence bbb in a given word. A simple manner of obtaining a DFA for this is to start for a NFA with a sequence of 4 states from initial to accepting, each one related to the next by a unique $\delta(q_j, b) = q_{j+1}$ transition, and then apply the NFA-to-DFA transformation. Another method is to retain the four states and add all missing transitions to obtain a DFA. Let us call it A_3.

A_3, and thus L_3 can be parallelized to a one-superstep BSP automaton by a construction similar to that of Example 3 above. The parallelization uses factorization Φ_1 and distribution $D_{\div p}$: it sends the first $n/p = |w|/p$ elements of w to location 0, the next n/p to location 1, etc. with a single superstep symbol ; at the end.

To do this we consider the three factors $w = w_1w_2$ of any $w \in L_3$ where $w_1 \in L((a + b)^*) - \{bbb\}$, $w_2 \in L(bbb(a + b)^*)$ i.e. w_2 begins with the first occurrence of bbb in w. Then we consider all the p possible positions for the first letter of w_2. Each one corresponds through $D_{\div p}$ to a BSPRE. For example
$|w_1bbb| \le n/p$ iff $\mathrm{Par}(D_{\div p}(w)) \in L(< ((a + b)^*) - \{bbb\})bbb, (a + b)^*, (a + b)^*, \ldots >)$. A BSP automaton A_0 can be derived from this BSP regular expression: by definition it operates in one superstep. Similar BSP automata A_i can be derived from the hypothesis that the first b symbol of the first bbb sequence in w starts at a certain

point in w. It follows that $A_0 + A_1 + A_2 + \ldots$ is a BSP automaton for L_3. Moreover it is possible to combine those BSP automata by a purely local process: add (create the disjuction) of all local DFA, and then build the combined synchronization function Δ by operating independently on the accepting states every local part A_j^i. The resulting BSP automaton accepts L_3 in a single superstep. Its speedup is the same as for Example 3.

Warning In our parallelization examples above it is assumed that an input word is split into regular blocks before being input to a custom-built BSPA. If processing time is understood as the time required to accept/refuse a given input word in each language, then our constructions indeed provide a $p\times$ speedup over the initial "sequential" DFA. But the reader should be aware that the BSPA are in general non-deterministic (NBSPA) and that to obtain this speedup in practice requires to transform them into equivalent deterministic BSPA. This pre-processing is amortized over the whole language but may have an exponential cost in space and time.

The construction of Examples 3 and 4 can clearly be applied to the general word recognition problem: for any given $x \in L((a+b)^*)$, one can construct a one-superstep BSP automaton A (i.e. based on Φ_1 and $D_{\div p}$) that paralellizes the language $(a+b)^*x(a+b)^*$. This A is the sum (language union) of $O(\max(p, |x|))$ BSP automata whose local DFA are minor modifications of A_x, the minimal DFA for accepting x.

All examples shown above provide candidate solutions to BSP-PARALLELIZE: they parallelize the given regular language L_j in one superstep with a BSP automaton whose size is linear in the size of a minimal DFA for L_j. All three examples are regular language of star-height one, and in general it is not clear whether such a parallelization is always possible.

Problem 3 *OPEN PROBLEM: does every instance of BSP-PARALLELIZE have a one-superstep solution?*

The answer would be positive if the number of states in the BSP automaton solution were allowed to grow exponentially. However the construction for showing this is very different from that of our above examples.

Proposition 10 *Every regular language L of regular expression r has a one-superstep parallelization $(\Phi_1, D_{\div p}, A)$ that can be constructed in time exponential in $|r|$ and such that $|A|$ is also exponential in $|r|$.*

Proof We show how $L = L(r)$ can be parallelized to a 1-superstep BSP automaton. Define $L_n = L \cap L((a+b)^n)$ and apply the following steps to build $(\Phi_1, D_{\div p}, A)$ such that $L(A) = \mathrm{Par}(D_{\div p}(L))$. Assume without loss of generality that $p = 2$ (if $p > 2$ the construction can be extended by induction).

1. Compute $L^0 = L \cap (a+b)^{n/2}$. Those words are the ones location 0 should accept in A: the first half of L_n's words i.e. L's words for a given length input length n. Let A^0 be a DFA and r^0 a regular expression for L^0.
2. Compute $L_n = L \cap (a+b)^n$ as a regular expression r_n.

3. For every one of the 2^n words $x \in L_{n/2}$, compute the Brzozowski differential $D_x(r_n)$ whose language is known to equal $x \backslash L_n = \{y \mid xy \in L_n\}$. This computation is a simple but exponential time-size converging normalization on the regular expression [4, 21].
4. Let $L' = \sum_{x \in (a+b)^{n/2}} (x \cap L_{n/2}) \backslash L_n$. Let A' be a DFA for accepting L'.
5. Define $A = (< A^0, A' >, \Delta)$ with Δ mapping to an accepting state vector, only those pairs of accepting states that correspond to the same x prefix.

By construction A will accept at location 0 precisely the first halves of words in L_n, and at location 1 their corresponding suffixes. The local automata are a sum (union) of all such possibilities and the synchronization function Δ recombines them in the correct way. □

In this section we have begun exploring parallelizations of regular languages. We have only shown one-superstep examples because there are trivial n-supersteps parallelizations that are of no interest either theoretically or practically. On simple examples of star-height one, space-efficient one-supersteps parallelizations have been constructed. It has also been proved that any regular language can have a one-superstep parallelization if exponential space (number of states) is allowed. It remains to explore intermediate solutions and how their complexity relates to star-height of the input regular language.

12 Intensional Notation and Application to Programming

In the theory presented up to this point, parallel vectors are enumerated but this is not a scalable point of view on parallel programming. It is more usual and convenient to represent vectors as functions from position i to the local element. This was the basis for the λ-calculus in [12] whose primitives are now implemented in BSML (BSP-OCaml). We show how to improve our theory of BSP automata in this manner so that vectors are not enumerated but defined by a simple symbolic notation.

Assume that the locations $i \in [p]$ are written in binary notation 0, 1, 10, 11 .... Define a binary regular expression (BRE) by the following grammar:

$$b ::= \emptyset \mid 0 \mid 1 \mid bb \mid b + b \mid b^*$$

Notice that BRE cannot encode the empty word. This notation is used to encode sets of locations. For example $b_1 = (0+1)^*1$ is the set of odd-rank locations, $b_2 = 0(0+1)(0+1)$ represents the first four locations when $p = 8$, and $b_3 = 010$ $(0+1)(0+1)(0+1)$ the third 8-position block of positions when $p = 32$ i.e. positions 16–23 over 6 binary digits. It would be possible to make this notation symbolic over p but that would require additional syntax and here we only explain how to make it symbolic over the position integers for a fixed p.

To avoid enumerating BSP vectors, replace the enumeration $< r_0, r_1, \ldots, r_{p-1} >$ by a grammar clause for *intensional vectors* of regular expressions:

$$R ::= \ < r@b >$$

where $r \in \mathrm{RE}$ and $b \in \mathrm{BRE}$. The meaning of $< r@b >$ is the vector of regular expressions whose local value is r at locations pid $\in L(b)$ and $\emptyset$ at other locations. For example if $p = 8$, the BSPRE $< (a+b)^{+}@b_2 >$ represents, in enumerated form,

$$< (a+b)^{+},\ (a+b)^{+},\ (a+b)^{+},\ (a+b)^{+},\ \emptyset,\ \emptyset,\ \emptyset,\ \emptyset >$$

i.e. the BSP language of one-superstep BSP words with non-empty local traces at positions 0–3 but empty traces at positions 4–7.

It is also possible to create BSP vectors by superposition $\parallel$ of multiple $r@b$ expressions. For example if $p = 4$, the BSPRE $< a@(0+1)^{*}0 \parallel b@(0+1)^{*}1 >$ corresponds to the enumeration

$$< a, b, a, b > .$$

With this new notation, redefine the BSP regular expressions:

$$R ::= \emptyset \mid \epsilon \mid < V > \mid R; R \mid R* \mid R + R.$$

using a new sub-grammar for BSP vectors:

$$V ::= r@b \ \mid V \parallel V$$

where $r \in \mathrm{RE}$ and $b \in \mathrm{BRE}$. The language of those intensional BSP regular expressions is defined with new rules for intensional vectors:

R	$L(R)$
$\emptyset$	$\{\ \}$
ϵ	$\{\epsilon\}$
$< r@b >$	$\prod_{i=0}^{i=p-1} \begin{cases} L(r) \text{ if } i \in L(b) \\ \{\emptyset\} \quad \text{else} \end{cases}$
$< r_1@b_1 \parallel \ldots \parallel r_k@b_k >$	$\prod_{i=0}^{i=p-1} \bigcup\{L(r_j) \mid i \in L(b_j), 1 \leq j \leq k\}$
$R_1; R_2$	$L(R_1)L(R_2)$
R^{*}	$L(R)^{*}$
$R_1 + R_2$	$L(R_1) \cup L(R_2)$

This new notation is “scalable” in the sense that its parallel implementations can slice it into local parts of the $< r@b >$ sub-expressions and simply combine their local values as (regular) functions of the location number pid. This is similar to what data-parallel programming languages provide. But its restriction to regular expressions has a major advantage: one location can compute the set of locations that hold a certain value. For a parallel implementation this amounts to inverting the communication relation, without specific source-code information to that effect.

We illustrate this kind of application on a simple but meaningful example: converting "get" requests for remote data into "put" operations for sending data. Assume we are programming a one-million core machine $p = 2^{20}$ in a high-level BSP language and a global parallel instruction (purely functional, for simplicity) of the form

get datavector from indexvector

whose input types are a

$$\text{datavector : float}^p; \qquad \text{indexvector : (int set)}^p$$

and whose output type is

$$(\text{float set})^p.$$

Let `datavector` be $< d_0, \ldots, d_{p-1} >$ and `indexvector` be $< I_0, \ldots, I_{p-1} >$ and assume that the get-from instruction realizes a global BSP operation whose resulting value is the vector $< A_0, \ldots, A_{p-1} >$ whose local values are

$$A_i = \{d_j \mid j \in I_i\}$$

. In other words get-from moves the elements of `datavector` as if every processor i sends a request for local data to processors whose positions j are listed in the local table I_i of `indexvector`. Consider now three successively improved data-parallel implementations for this operation.

12.1 2-Phases Implementation

A straightforward implementation is to use two BSP supersteps. The first one sends a set of requests from every processor i to processors $j \in I_i$. The second superstep sends back the requested data i.e. processor j communicates back with all requesting processors $\{i' \mid j \in I_{i'}\}$. The disadvantage of this scheme is that its BSP costs includes two global barriers (i.e. twice the global latency) and implementors wish to avoid it by "converting get into put" using one of the two following methods.

12.2 1-Phase $O(p)$ Inversion

The SPMD paradigm for data-parallel programming ensures that the source program is common to every local processor and thus the code for our instruction is known at every position i, only data d_i and I_i are local. We can consider the I_i to be (finite) languages of 20-bit words ($\log p$-bit words) and improve the get-from instruction's syntax as follows: `indexvector` is given as a BSPRE e.g. $< r@b >$ where r

encodes the I_i. As a result of this language construct, every processor j can directly compute its set of target processors for sending data $\{i' \mid j \in I_{i'}\}$ by simply running every 20-bit word i' through a finite automaton for r: if accepted and if $j \in L(b)$ then processor j should send its d_j to processor i'. This can be done in time proportional to $2^{20} = p$. Moreover it does not require two BSP supersteps but only one: the "get" is implemented directly by a "send", thanks to the simplicity of the sub-language on integer sets for I_i.

12.3 1-Phase $O(\log P)$ Inversion

An even more efficient implementation of the 1-phase implementation is possible due to the simplicity of BSPRE. Every processor j can simply enumerate $L(r)$ because it is a regular language. This can be done in time proportional to the size of this set times the length of the words in it: that is $O(\log p)$ time the number of messages [13]. In our example, if processor j has a small number of requests to satisfy e.g. 3, that would prevent it from executing p or one million instructions.

12.4 Other Intensional Notations

All p-indexed vectors in the theory of BSP automata can be manipulated with similar regular-indexing notations. For example the factorization Φ and distribution D functions on sequential regular languages can likewise be restricted to intensional notations. The result would be to automate the inversion of D, and from there compute a BSPRE directly from a sequential regular expression.

Moreover, partition, distribution and synchronization are enumerated functions whose implementation may not be obvious. Defining a regular notation for those functions improves their ease of programming, makes expressions "scalable" (parametric on p) and leads to useful inversion algorithms e.g. inverse distribution. For example BSPLib [7] and many other "SPMD" data-parallel programming systems present the local code (which corresponds to our local sequential automata) as a function of the location number called `pid`.

All the advantages of an intensional notation can be obtained by an extended notation for BSPRE that we define below. Moreover as we will now explain, the low complexity of regular languages allows us to automate the inversion of the (location $\rightarrow$ value) map, a useful operation for parallel algorithms that is rarely provided by parallel languages.

13 Conclusions and Future Work

We have defined and begun exploring a BSP variant of elementary automata theory. Some key observations are that BSP automata are more than product automata, their natural alphabet is the set of regular expressions, and their state-space is exponential in the number of parallel locations. BSP automata and BSP languages preserve all the classical closure properties: non-determinism, ϵ-transitions and determinization, but break the classical properties of minimization. The interaction between state-minimization and BSP cost optimization remains to be understood. Compact symbolic notations can be designed for the parallel-vector components of BSP automata and BSP regular-expressions. BSP automata can help automate bulk-synchronous parallel programming e.g. as a declarative language for connection supersteps, defining communication structures and cost optimization.

Future work will explore (a) BSP regular grammars and their generalization to BSP context-free languages, (b) the application of BSP automata to parallel text processing and parsing, (c) applications to pattern matching and to parallel data structure (tries etc.) (d) generalizations of BSP automata to heterogeneous and hierarchical architectures.

BSP automata constitute a clear and easily-understood basis for teaching, specifying and writing parallel programs. They can be used to combine the control- and communication structure of BSP programs, analyze or optimize that structure. BSP regular expressions are useful for declarative programming of parallel operations with explicit data placement and synchronizations.

Acknowledgments The author thanks Frédéric Loulergue, Thibaut Tachon, Arnaud Lallouet and Chong Li for their insightful remarks and help with proofreading. This work has been supported by the author's position at Huawei/CSI-FRC.

Appendix 1

1.1 BSP Automaton Versus Product of Automata

The theory of products of automata is developed by Gécseg in [6]. It describes decompositions of finite automata as products of simpler ones, and is closely related to the theory of semigroup decompositions.

A BSP computation is more than a vector of sequential computations, and this is reflected by the fact that a BSP automaton is more than a vector of DFA. This is relatively obvious but we make it here completely explicit by comparing that definition (Definition 2) with that of a product automaton.

Let $A^i = (Q^i, \Sigma, \delta^i, q_0^i, F^i), i \in [p]$ be a vector of DFA and $A = (A^0, \dots, A^{p-1}, \Delta)$ a BSP automaton built from them. Gécseg's definition (Definition 4.2 in [6]) of *machine product* applies to Mealy machines i.e. DFA with an output function

added. For the purpose of language recognition the output functions are not necessary, so we consider the machine product $\prod_i A^i$ as the Gécseg without outputs. According to Definition 4.2 in [6], $\prod_i A^i$ is a state machine with vector states $\prod_i Q^i$, just like the BSP automaton, a new externally-defined alphabet X, and a special transition function δ_ψ based on the externally-given function

$$\phi : (\prod_i Q^i) \times X \to \Sigma^p$$

such that

$$\delta_\psi(< q^0, \ldots, q^{p-1} >, x) =< \delta^0(q^0, x^0), \ldots, \delta^{p-1}(q^{p-1}, x^{p-1}) >$$

where $< x^0, \ldots, x^{p-1} >= \phi((< q^0, \ldots, q^{p-1} >, x)$.

It is trivial to show that the automaton product can simulate the asynchronous parts of a BSP computation. But the structure of δ_ψ is not the same as a synchronization function Δ. The product automaton could simulate the BSP automaton but at the expense of an unnatural encoding e.g. $X = (\Sigma^*)^p \times \prod_i Q^i$ to let ϕ distinguish asynchronous versus synchronous applications of Δ.

But an alphabet which contains states and trace histories is hardly a natural (and low-complexity) encoding. Following this theoretical direction would defeat the purpose of BSP automata that is not the study of algebraic decompositions, or decidability, but rather to investigate programming notations having BSP implementations.

Appendix 2

2.1 Regular Expressions

Regular expressions are a well-known notation for the languages of finite automata. The definitions and properties we state below can be found in every textbook on finite automata for example Chap. 3 of [21]. The languages denoted by regular expressions are called regular, and that class of languages is the same as those recognized by a DFA or its equivalent, non-deterministic variants.

A regular expression is an expression r from the following grammar:

$$r \ ::= \ \emptyset \mid \epsilon \mid a \mid rr \mid r* \mid r + r$$

where $a \in \Sigma$ is any symbol from the alphabet. We write RE for the set of regular expressions . The language of a regular expression is defined by $L : \text{RE} \to \mathcal{P}(\Sigma^*)$ where function L translates $\emptyset$ to the empty language, ϵ (resp. a) to a singleton empty word (resp. singleton one-symbol word), $r_1 r_2$ to the concatenation of languages, r^* to $L(r)^* = \bigcup_{n\geq 0} L(r)^n$ and $r_1 + r_2$ to the union of the two languages. The union,

concatenation, *-closure of two regular languages is regular. The complement of a regular language is also regular [15].

For $r \in RE$, there exists a NFA A such that $L(A) = L(r)$. A time-optimal quadratic time algorithm for this transformation is described in [3]. It has been improved to a linear-space and parallelisable algorithm in [22]. Both use the *Glushkov* automation of r whose states are the positions in r's syntax tree.

Inversely, for A a finite automaton, there exists $r \in RE$ such that $L(r) = L(A)$. The two equivalence properties are called Kleene's theorems [10].

References

1. Anantharaman, S., Hains, G.: A synchronous bisimulation-based approach for information flow analysis. In: Third Workshop on Automated Verification of Critical Systems: (AVOCS'03), Southampton, (UK) (2003)
2. Benzaken, C.: Systèmes formels: introduction à la logique et à la théorie des langages. Masson (1991)
3. Brüggemann-Klein, A.: Regular expressions into finite automata. Theoret. Comput. Sci. **120**(2), 197–213 (1993)
4. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM **11**(4), 481–494 (1964)
5. D'Antoni, L., Veanes, M.: Minimization of symbolic automata. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, pp. 541–553. ACM (2014)
6. Gécseg, F.: Products of Automata. Springer (1986)
7. Hill, J.M.D., McColl, B., Stefanescu, D.C., Goudreau, M.W., Lang, K., Rao, S.B., Suel, T., Tsantilas, T., Bisseling, R.H.: BSPlib: The BSP programming library. Parallel Comput. **24**(14) (1998)
8. Hopcroft, J.: An n log n algorithm for minimizing states in a finite automaton. Technical Report No. STAN-CS-70-190, Stanford University, Department of Computer Science (1971)
9. John, M.: Howie. Automata and languages. Clarendon Press, Oxford (1991)
10. Kleene, S.C.: Representation of events in nerve nets and finite automata. Automata Stud., 3–41 (1956)
11. Loulergue, F., Hains, G.: Functional parallel programming with explicit processes: Beyond SPMD. Lecture Notes in Computer Science, vol. 1300. Springer (1997)
12. Loulergue, F., Hains, G., Foisy, Ch.: A calculus of recursive-parallel BSP programs. Sci. Comput. Programm. (2000)
13. Mäkinen, E.: On lexicographic enumeration of regular and context-free languages. Acta Cybern. **13**(1), 55–61 (1997)
14. McColl, W.F.: Scalable computing. In: van Leeuwen, J. (ed.) Computer Science Today. LNCS, vol. 1000. Springer (1995)
15. McNaughton, Robert: Elementary Computability, Formal Languages and Automata. Prentice-Hall, Englewood Cliffs, NJ (1982)
16. Merlin, A., Hains, G.: A bulk-synchronous parallel process algebra. Comput. Lang. Syst. Struct. **33**(3), 111–133 (2007)
17. Milner, R.: Communication and Concurrency. Prentice Hall, New York (1989)
18. Pin, J.-É.: Variétés de Langages Formels. Masson, Paris (1984)
19. Turing, A.: On computable numbers with an application to the entscheidungs problem. Proc. Lond. Math. Soc. **2**(42), 230–265 (1936)
20. Valiant, L.G.: A bridging model for parallel computation. CACM **33**(8), 103 (1990)
21. Wood, D.: Theory of Computation. Wiley (1987)
22. Ziadi, D., Ponty, J.-L., Champarnaud, J.-M.: Passage d'une expression rationnelle à un automate fini non-déterministe. Bull. Belgian Math. Soc. Simon Stevin **4**(1), 177 (1997)