Algebraic Laws for BSP Programming

He Jifeng, Quentin Miller, Lei Chen*

Oxford University Computing Laboratory,
Wolfson Building, Parks Road,
Oxford OX1 3QD, U.K.

Abstract. The Bulk-Synchronous Parallel (BSP) model of computation reflects the capabilities and costs of communication on a wide range of general-purpose parallel architectuers. Thus it allows general-purpose parallel software and hardware to be developed independently of one another; much as the von Neumann model provides the same facility for sequential computation. This paper presents a simple programming notation for shared-memory programming, based upon BSP constructs. The notation is defined formally by its effect upon process states. Algebraic laws are given which allow program derivation. A further set of laws allow transformation of finite programs to a normal form.

1 Introduction

Mathematical models and their algebraic laws provide a theoretical basis for correctness-preserving program transformations, either from a specification to an efficiently executable code or from an existing program to a program that is efficiently compilable to another architecture. As an effort towards a foundation of these methods, we present in this paper a mathematical model and some algebraic laws for BSP programming.

The *Bulk-Synchronous Programming* (BSP) model was originally proposed by [6, 4, 1] as a machine model for general-purpose parallel architectures and a unified cost model for measuring the performance of both parallel computers and parallel programs with BSP structure. However, one can also view it as a programming model (see [2, 5, 7]), in which every program has the following two basic features:

- *Supersteps:* The execution of a program proceeds in supersteps, in which concurrent components (normally called processes) compute asynchronously, with a global synchronisation at the end of each superstep.
- *Global communication:* Processes communicate with one another asynchronously. All communication operations take effect at the synchronisation point.

* {jifeng,quentin,lchen}@comlab.ox.ac.uk

2 The Mathematical Model

We begin by characterising properties of a BSP process. An *observation* of a process is the record of its behaviours during execution. A process can be specified by the set of all possible observations that can be made of it. If two processes have the same set of observations, we say that they are *equivalent* — this provides a semantic basis for developing sound transformation rules.

In a BSP program, a process interacts with its environment through communications which take effect at synchronisation points. Thus, the behaviour of a process can be fully captured in terms of the trace of synchronisation points, the record of pending communication operations, and the values of program variables.

A synchronisation point is represented by a pair $\langle out, gs\rangle$, where out describes all the outputs taking place between the previous synchronisation and the current one, and gs stands for the values of global variables after the current synchronisation.

Definition. The state of a BSP process consists of the following components:

ls :	$LVar \rightarrow Val$	the state of local variables
gs :	$PVar \rightarrow Val$	the state of global variables
out :	$PVar \rightarrow \mathrm{Bag}(Val)$	the pending output operations taking place up to the moment
tr :	$\mathrm{Seq}(SYN)$	the sequence of synchronisation points occuring up to the moment

where $SYN \mathrel{\widehat{=}} (PVar \rightarrow \mathrm{Bag}(Val)) \times (PVar \rightarrow Val)$.

We write s for an initial state and s' for a final state of an execution of a BSP process. (We also write all the values related to initial state and final state in the same style.) To capture the meaning of termination we introduce a boolean variable ok. The equation $ok = \mathbf{true}$ means a program starts its execution properly, and $ok' = \mathbf{true}$ indicates that the process terminates successfully. We define an observation of a process as a tuple $\langle ok, s, s', ok'\rangle$.

An obvious property of observations of BSP processes is that the sequence of synchronisation points of the initial state is always a prefix of that of the final state (i.e. $tr \leq tr'$). The set OBS consists of all possible observations of a BSP process.

$$OBS \mathrel{\widehat{=}} \{\langle ok, s, s', ok'\rangle \mid tr \leq tr'\}$$

For convenience, we identify a process P as a predicate $P(ok, s, s'ok')$, which is satisfied if and only if $\langle ok, s, s', ok'\rangle$ is a possible observation of P.

We wish to embed our programming language in the relational calculus, where the meaning of a program is defined by a pair of predicates $\langle pre, post\rangle$. If the program starts in an initial state s satisfying the precondition pre, it will terminate in a final state s' satisfying the postcondition $post$.

$$pre \vdash post \mathrel{\widehat{=}} (ok \wedge pre \Rightarrow ok' \wedge post) \wedge (tr \leq tr')$$

In the following section we will associate each program construct of our language with such a pair of predicates.

3 Semantics and Algebraic Laws

In this section, we give formal definitions of program constructs of a programming language, and examine the algebraic laws which hold for the programming constructs. Proofs that the laws are sound with respect to the semantics are straightforward and have been omitted. The laws for sequential program constructs and recursion have been studied by [3]. We will add laws for synchronisations, input/output operations and parallel constructs.

3.1 A BSP Language

In order to formulate laws, we introduce the following programming notation.

II	skip
CHAOS	abort
$P \lhd b \rhd Q$	conditional
$P; Q$, $P \| Q$	sequential, parallel composition
var x	variable declaration
$px := x$	assignment
synch	synchronisation
get(g), put(g)	input, output
par P endpar	parallel block

Processes which are connected to one another by parallel composition are called *parallel partners*. In the program text, a group of parallel partners is delimited by a parallel block. Parallel partners communicate with one another by sharing the variables of their parent. We call variables of the parent *global variables*.

3.2 Notational Conventions

Let P be a process and e be an expression. var(P) and var(e) denote the set of variables used by P and e, respectively. We use f, g to denote functions, and dom(f), dom(g) to denote their domains.

Let f_1 and f_2 be two functions. $f_1 \oplus f_2$ is the function which behaves like f_2 in the domain of f_2, and behaves like f_1 otherwise.

$$f_1 \oplus f_2(x) \;\widehat{=}\; \begin{cases} f_2(x) & \text{if } x \in \mathrm{dom}(f_2) \\ f_1(x) & \text{if } x \in (\mathrm{dom}(f_1) - \mathrm{dom}(f_2)) \end{cases}$$

The operator $\uplus$ is used to denote the union operation of bags. Let f_1 and f_2 be two functions with the same domain S and mapping each element of S to a bag. We will write $f_1 \uplus f_2$ for the function $\lambda x \bullet (f_1(x) \uplus f_2(x))$.

Let T be a set of state components. Id_T stands for the identity relation on T. For instance, we will write $Id_{\{ls,gs\}}$ for the predicate $ls' = ls \wedge gs' = gs$.

3.3 Local Operations

Execution of II does nothing, but terminates successfully.

$$\mathrm{II} \mathrel{\widehat{=}} \mathbf{true} \vdash Id_{\{ls,gs,out,tr\}}$$

Process CHAOS, sometimes denoted as $\bot$, is the worst process. It may have any observation in *OBS*; thus its behaviour is totally unpredictable.

$$\mathsf{CHAOS} \mathrel{\widehat{=}} \mathbf{false} \vdash \mathbf{true}$$

Let P and Q be processes and b be a boolean expression which contains no global variables $P \lhd b \rhd Q$ executes P if b holds; Q otherwise.

$$P \lhd b \rhd Q \mathrel{\widehat{=}} (b \wedge P) \vee (\neg b \wedge Q)$$

Laws. $P \lhd \mathbf{true} \rhd Q = P$
$P \lhd b \rhd P = P$
$P \lhd b \rhd Q = Q \lhd \neg b \rhd P$
$(P \lhd b \rhd Q) \lhd c \rhd R = P \lhd b \wedge c \rhd (Q \lhd c \rhd R)$
$P \lhd b \rhd (Q \lhd c \rhd R) = (P \lhd b \rhd Q) \lhd c \rhd (P \lhd b \rhd R)$

Let P, Q be processes. Execution of their sequential composition begins with P. If P does not terminate, Q will not start execution. Otherwise, Q will start execution from the final state of P.

$$P;Q \mathrel{\widehat{=}} \exists\, \widetilde{ok}, \widetilde{s} \bullet P[\widetilde{ok}, \widetilde{s}/ok', s'] \wedge Q[\widetilde{ok}, \widetilde{s}/ok, s]$$

Laws. $\mathrm{II}; P = P; \mathrm{II} = P$
$(P;Q);R = P;(Q;R)$
$(P \lhd b \rhd Q);R = (P;R) \lhd b \rhd (Q;R)$

Declaration var z declares a variable z for use in the program that follows. For simplicity, we assume that z is not already in scope. We say that variable x is *free* in process P if it is not in the scope of any declaration of x in P, and *bound* otherwise.

A declaration assigns the special value *void* $\in$ *Val* to the new variable.

$$\mathsf{var}\ z \mathrel{\widehat{=}} \mathbf{true} \vdash ls' = (ls \oplus (z \mapsto void)) \wedge Id_{\{gs,out,tr\}} \quad \text{if } z \notin \mathrm{dom}(ls)$$

Laws. $P;\ \mathsf{var}\ z = \mathsf{var}\ z;\ P$ if z not free in P
$(\mathsf{var}\ z;\ P) \| Q = \mathsf{var}\ z;\ (P \| Q)$ if $z \notin \mathrm{var}(Q)$

Execution of an assignment $x := e$ assigns the value of an expression e to a variable x. For simplicity, we assume that evaluation of e always delivers a result, so the assignment will always terminate. In a BSP program assignment is a local computation, i.e. operands of assignment contain no global variables.

$$x := e \mathrel{\widehat{=}} \mathbf{true} \vdash ls' = ls \oplus \{x \mapsto e\} \wedge Id_{\{gs,out,tr\}} \ .$$

Laws. $(x := x) = \mathrm{II}$
$(x := d; x := e) = (x := e[d/x])$
$(x := e); (P \lhd b \rhd Q)) = (x := e; P) \lhd b[e/x] \rhd (x := e; R)$

3.4 Synchronisation

A synchronisation operation, denoted by synch, suspends the execution of a process until all of its parallel partners are ready to engage in the synchronisation, and all communications amongst the participating processes take effect. From the view of the process, synch completes its output operations performed in the superstep, and updates the values of global variables. However the new state of global variables is not determined by the process alone; it depends upon the output operations of all the processes which synchronise and the original values of the global variables. At the end of its execution, the output state is cleared and a new synchronisation point is appended to the trace *tr*.

$$\mathsf{synch} \mathrel{\widehat{=}} \mathbf{true} \vdash Id_{\{ls\}} \wedge out' = \lambda px \bullet \emptyset \wedge tr' = tr^\frown\!<\!\langle out, gs'\rangle\!>$$

where $\lambda px \bullet \emptyset$ denotes a function mapping each global variable px to an empty bag, and $^\frown$ denotes the concatenation operator of sequences.

Laws.
$$\begin{aligned}(x := e); \mathsf{synch} &= \mathsf{synch}; (x := e)\\ \mathsf{synch}; (P \lhd b \rhd Q) &= (\mathsf{synch}; P) \lhd b \rhd (\mathsf{synch}; Q)\end{aligned}$$

3.5 Output

Our output operation is very general; it sends a set of bags of values to a set of global variables. Let $g : PVar \rightarrow \mathrm{Bag}(Val)$ be a total function mapping each global variable to a bag (probably an empty bag) of values. The operation put(g) sends each global variable the bag of values which is associated with the variable by the function g.

$$\mathsf{put}(g) \mathrel{\widehat{=}} \mathbf{true} \vdash Id_{\{ls, gs, tr\}} \wedge out' = out \uplus g$$

BSP processes typically proceed in cycles of three phases: input, local computation, output. To make the most efficient use of the communications network it is usually desirable to postpone the execution of input and output operations until the end of the superstep; thus we would like laws which allow put (and get) statements to be merged and postponed:

Laws.
$$\begin{aligned}\mathsf{put}(g_1); \mathsf{put}(g_2) &= \mathsf{put}(g_1 \uplus g_2)\\ \mathsf{put}(g); (x := e) &= (x := e); \mathsf{put}(g) \quad \text{if } \forall\, px \in PVar \bullet x \notin \mathrm{var}(g(px))\\ \mathsf{put}(g); (x := e) &= \mathsf{var}\ z;\ (x, z := e, x);\ \mathsf{put}(\lambda px \bullet (g(px)[z/x]))\end{aligned}$$

3.6 Input

Let $f : LVar \rightarrowtail\!\!\!\!\!+ PVar$ be a (partial) function which maps a local variable to a global variable from which it is going to input. The input operation get(f) is defined by

$$\mathsf{get}(f) \mathrel{\widehat{=}} \begin{cases} \mathrm{II} & \text{if } \mathrm{dom}(f) = \emptyset\\ (\mathrm{II} \lhd out = \lambda px \bullet \emptyset \rhd \mathsf{synch});\\ (x_1, \ldots, x_n := f(x_1), \ldots, f(x_n)) & \text{if } \mathrm{dom}(f) = \{x_1, \ldots, x_n\}\end{cases}$$

From the definition of input operation, it follows that a synchronisation must take place between an output and an input, provided that the output and input operations are not trivial.

Law. $\mathsf{put}(g); \mathsf{get}(f) \;=\; \mathsf{put}(g); \mathsf{synch}; \mathsf{get}(f)$
$\qquad\qquad\qquad\qquad$ if $(g \neq \lambda px \bullet \emptyset) \wedge \mathrm{dom}(f) \neq \emptyset)$

For the same reason that we merge and postpone output operations, we wish to merge input operations and bring them before local computations.

Laws.
$$\begin{aligned}
\mathsf{get}(f_1); \mathsf{get}(f_2) &= \mathsf{get}(f_1 \oplus f_2)\\
(x := e); \mathsf{get}(f) &= \mathsf{get}(f) \quad \text{if } x \in \mathrm{dom}(f)\\
(x := e); \mathsf{get}(f) &= \mathsf{get}(f); (x := e) \quad \text{if } (\{x\} \cup \mathrm{var}(e)) \cap \mathrm{dom}(f) = \emptyset\\
(x := e); \mathsf{get}(f) &= \mathsf{var}\ z;\ z := y;\ \mathsf{get}(f); x := e[z/y]
\end{aligned}$$
$\qquad$ where $x \notin \mathrm{dom}(f) \wedge y \in \mathrm{dom}(f)$

3.7 Parallel Composition

Let P and Q be processes. The notation $P \| Q$ describes a process which executes P and Q in parallel where their interactions are via shared global variables at the synchronisation points. Processes P and Q start their execution at the same environment (consisting of gs, out, tr and ok). During the execution, P and Q communicate with each other by sending values to global variables. At each synchronisation point the new value of a global variable is chosen internally from the bag of messages it has received from P, Q and their parallel partners at that moment. If $\langle ok_1, s_1, s_1', ok_1' \rangle$ and $\langle ok_2, s_2, s_2', ok_2' \rangle$ are the observations of P and Q, respectively, we introduce a notation $(s_1', ok_1')\&(s_2', ok_2')$ to denote the set of possible values of final state s' and boolean variable ok' of $P \| Q$.

$$\begin{aligned}
P \| Q \;\widehat{=}\; & \exists\, \langle ok_1, s_1, s_1', ok_1' \rangle, \langle ok_2, s_2, s_2', ok_2' \rangle \bullet\\
& \qquad P[ok_1, s_1, s_1', ok_1' / ok, s, s', ok'] \wedge Q[ok_2, s_2, s_2', ok_2' / ok, s, s', ok']\\
& \wedge (ok = ok_1 = ok_2)\\
& \wedge (tr = tr_1 = tr_2 = <>) \wedge (out = out_1 = out_2 = \emptyset)\\
& \wedge (s', ok') \in (s_1', ok_1')\&(s_2', ok_2')
\end{aligned}$$

We leave the definition of & to Appendix A.

Laws.
$$\begin{aligned}
(P \| Q) \| R &= P \| (Q \| R)\\
P \| Q &= Q \| P\\
\mathrm{II} \| P &= P\\
\mathsf{CHAOS} \| P &= \mathsf{CHAOS}\\
(P \lhd b \rhd Q) \| R &= (P \| R) \lhd b \rhd (Q \| R)
\end{aligned}$$

3.8 Parallel Block and Hiding

In describing the behaviour of an individual process, we need to record its local state and its synchronisations, which are the interfaces between the process and its sequential partners and parallel partners, respectively. However, for a group of parallel processes which are delimited in a parallel block we want to conceal the local states of the parallel processes and the internal interactions between them.

Let P be a BSP process composed of a group of parallel partners, and *obs* be an observation of P. A hiding operation Hide(*obs*) abstracts away the internal behaviours of parallel components of P from *obs*.

The global state of P is updated at each synchronisation point and finally at the termination point of P. We shall use a function choice(gs, out) to produce a set of all possible new global states at each of these points. The definition of choice(gs, out) is left to be decided at the implementation level. An observation is said to be ***consistent*** if the new global state after each synchronisation lies in the set choice(gs, out). The observations of a parallel block par P endpar must be consistent.

A parallel block par P endpar hides the observations of P.

$$\begin{aligned}
\mathsf{par}\ P\ \mathsf{endpar}\ \widehat{=}\ & \exists\ \widehat{ok}, \widehat{s}, \widehat{s}', \widehat{ok}' \bullet P[\widehat{ok}, \widehat{s}, \widehat{s}', \widehat{ok}' / ok, s, s', ok'] \\
& \wedge\ \langle ok, s, s', ok' \rangle \in \mathrm{Hide}(\langle \widehat{ok}, \widehat{s}, \widehat{s}', \widehat{ok}' \rangle) \\
& \wedge\ \mathrm{consistent}(\langle \widehat{s}, \widehat{s}' \rangle)
\end{aligned}$$

$$\begin{aligned}
\widehat{obs} \in \mathrm{Hide}(obs) \iff\ & \widehat{ok} = ok\ \wedge\ \widehat{ok}' = ok' \\
& \wedge\ \widehat{ls} = gs \\
& \wedge\ \widehat{ls}' \in \mathrm{choice}(gs', out') \\
& \wedge\ \widehat{gs}' = \widehat{gs}\ \wedge\ \widehat{out}' = \widehat{out}\ \wedge\ \widehat{tr}' = \widehat{tr}
\end{aligned}$$

$$\begin{aligned}
\mathrm{consistent}(\langle \widehat{s}, \widehat{s}' \rangle)\ \widehat{=}\ & tr'[1].gs \in \mathrm{choice}(gs, tr'[1].out)\ \wedge \\
& \forall\, 1 < k \leq \#tr' \bullet tr'[k].gs \in \mathrm{choice}(tr'[k-1].gs, tr'[k].out)
\end{aligned}$$

Laws. par CHAOS endpar $=$ CHAOS
par II endpar $=$ II

4 Normal Form

To illustrate the power of our mathematical model, we can derive laws which can be used to reduce every finite program text of our BSP language to a normal form. This is a sequential program consisting of a sequence of input/compute/outpu phases, which is probably followed by an atomic process or CHAOS if the program diverges.

Definitions. *Let* lv *be a list of local variables and* e *be a list of expressions with the same length of* lv. *An* atomic process A *of a BSP program is a program in the form*

$$\begin{array}{lll} \multicolumn{3}{l}{\mathsf{get}(f);\ (\mathbf{lv} := \mathbf{e});\ \mathsf{put}(g)} \\ \text{where} & f : LVar \twoheadrightarrow PVar & \text{the input function} \\ \text{and} & LVar = \mathrm{var}(\mathbf{lv}) & \text{the set of local variables} \\ \text{and} & g : PVar \to \mathrm{Bag}(Val) & \text{the output function.} \end{array}$$

A Phase is an atomic process followed by a synchronisation.

$$Phase \mathrel{\widehat{=}} A;\ \mathsf{synch}$$

A BSP program is said to be in normal form *if it is in the form*

$$NF \ = \ \mathsf{CHAOS} \mid A \mid Phase \mid Phase; NF \mid \mathsf{if}\ b_1 \to NF \ \square \ \ldots \ \square \ b_n \to NF \ \mathsf{fi} \ .$$

In the above definition the multi-conditional if $b_1 \to P_1 \ \square \ \ldots \ \square \ b_n \to P_n$ fi is defined as follows.

$$\mathsf{if}\ b_1 \to P_1 \ \square \ \ldots \ \square \ b_n \to P_n \ \mathsf{fi} \mathrel{\widehat{=}} (\bigvee_{i=1}^{n} b_i \wedge P_i) \vee (\bigwedge_{i=1}^{n} \neg b_i \wedge \bot)$$

To explain how to reduce a program text to normal form, it is sufficient to show how each primitive operation can be written in normal form and how each operator, when applied to operands in normal form, yields a result expressible in normal form. Limited by space, here we will only give a treatment to parallel constructs. For convenience, we shall write A_i for atomic processes in the form

$$\mathsf{get}(f_i);\ (\mathbf{lv}_i := \mathbf{e}_i);\ \mathsf{put}(g_i) \ .$$

The following four laws convert any parallel composition into normal form.

Laws. $(A_1; \mathsf{synch}) \parallel (A_2; \mathsf{synch}) \ = \ A; \mathsf{synch}$
$A_1 \parallel A_2 \ = \ A$
$(A_1; \mathsf{synch}; P) \parallel (A_2; \mathsf{synch}; Q) \ = \ ((A_1; \mathsf{synch}) \parallel (A_2; \mathsf{synch}));\ (P \parallel Q)$
$(A_1; \mathsf{synch}; P) \parallel (A_2) \ = \ ((A_1; \mathsf{synch}) \parallel (A_2; \mathsf{synch}));\ (P \parallel \mathrm{II})$

$$\text{where } A \mathrel{\widehat{=}} \mathsf{get}(f_1 \oplus f_2);\ (\mathbf{lv}_1, \mathbf{lv}_2 := \mathbf{e}_1, \mathbf{e}_2);\ \mathsf{put}(g_1 \uplus g_2)$$

The next three laws show how to move a superstep out of a parallel block. Let **pv** be the list of global variables of P, f be an input function and **v** be the list of variables in dom(f), where $\mathbf{v} = \{x_1, \ldots, x_n\}$. We write $f(\mathbf{v})$ for $f(x_1), \ldots, f(x_n)$.

Laws. $\mathsf{par}\ \mathsf{get}(f); P\ \mathsf{endpar} \ = \ \mathsf{var}\ \mathbf{z};\ \mathbf{z} := f(\mathbf{v});\ \mathsf{par}\ P[\mathbf{z}/\mathbf{v}]\ \mathsf{endpar}$
$\mathsf{par}\ \mathbf{lv} := \mathbf{e}; P\ \mathsf{endpar} \ = \ \mathsf{par}\ P[\mathbf{e}/\mathbf{lv}]\ \mathsf{endpar}$
$\mathsf{par}\ \mathsf{put}(g); \mathsf{synch}; P\ \mathsf{endpar} \ = \ \mathbf{pv} := \mathrm{choice}(\mathbf{pv}, g(\mathbf{pv}));\ \mathsf{par}\ P\ \mathsf{endpar}$

Two final laws complete the transformation scheme:

Laws. $\mathsf{par}\ A\ \mathsf{endpar} \ = \ \mathsf{par}\ A; \mathsf{synch}\ \mathsf{endpar}$
$\mathsf{par}\ P \lhd b(\mathbf{pv}) \rhd Q\ \mathsf{endpar} \ = \ \mathsf{par}\ P\ \mathsf{endpar} \ \lhd b(\mathbf{pv}) \rhd \ \mathsf{par}\ Q\ \mathsf{endpar}$

5 Future Work

We have presented a mathematical model and a set of algebraic laws for BSP programming. The main objective is to provide a semantics as general as possible, in order to ensure that the implementor has the greatest possible scope to provide an efficient implementation on various forms of parallel architectures. One obvious application of our laws is to transform sequential programs into BSP programs using the normal form transformations.

Our goal is to develop a comprehensive theory for specification, refinement and implementation of BSP programs.

6 Acknowledgements

The authors wish to acknowledge W.F. McColl and C.A.R. Hoare for their discussions and encouragements. The comments given by the members of Oxford Parallel were also helpful for an earlier version of this paper.

References

1. R. H. Bisseling and W. F. McColl: Scientific Computing on Bulk Synchronous Parallel Architectures. Technical Report 836, Department of Mathematics, University of Utrecht, December 1993.
2. T. Cheatham, A. Fahmy, D. C. Stefanescu, and L. G. Valiant: Bulk Synchronous Parallel Computing – A Paradigm for Transportable Software. Technical Report TR-36-94, Harvard University, Computer Science Department, 1994.
3. C. A. R. Hoare, I. J. Hayes, J. He, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. A. Sufrin: Laws of Programming. *Communications of the ACM*, 30(8):672–687, August 1987. see Corrigenda in Communications of the ACM, 30(9): 770.
4. W. F. McColl: General Purpose Parallel Computing. In A. M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation. Proc. 1991 ALCOM Spring School on Parallel Computation*, pages 337–391. Cambridge University Press, 1993.
5. W. F. McColl: BSP Programming. In G Blelloch, M Chandy, and S Jagannathan, editors, *Proc. DIMACS Workshop on Specification of Parallel Algorithms*, Princeton, May 1994. American Mathematical Society.
6. Leslie G. Valiant: A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.
7. Leslie G. Valiant: BSP Computing. Technical report, Harvard University, Computer Science Department, January 1995.

A State Combination

Let P and Q be a pair of parallel partners. The state of their parallel composition $P||Q$ is a combination of the states of P and Q. $P||Q$ takes both of the local states of P and Q as its local state. Formally, it can be stated as

$$ls_{P||Q} \mathrel{\widehat{=}} ls_P \oplus ls_Q \ .$$

The global state of $P \| Q$ is the global state shared by P and Q, i.e.

$$gs_{P\|Q} = gs_P = gs_Q \ .$$

The combinations of two output states or two traces of synchronisation points is a bit more complex. For the initial state, since no output operations and synchronisations have been performed, the output state and trace of synchronisations are empty. During the execution of P and Q, the output state of $P\|Q$ is the union of output states of P and Q, which can be formulated as

$$out_{P\|Q} \ \widehat{=} \ out_P \uplus out_Q \ .$$

We overload the operator & to denote the combination of the output states.

$$\begin{array}{l} out_{P\|Q} \ \widehat{=} \ out_P \& out_Q \\ \quad \text{where } out_P \& out_Q \ \widehat{=} \ \begin{cases} out_P \uplus out_Q & \text{if } (\#tr_P = \#tr_Q) \\ out_P & \text{if } (\#tr_P > \#tr_Q) \\ out_Q & \text{if } (\#tr_P < \#tr_Q) \end{cases} \end{array}$$

The combination of two traces of synchronisation points combines the corresponding synchronisation points of the two traces. If one trace is shorter than the other — which is the circumstance when one of the process terminates earlier than the other — the excess synchronisation points of the longer trace are copied to the result of the combination. Again we overload the operator &.

$$\begin{array}{l} tr_{P\|Q} \ \widehat{=} \ tr_P \& tr_Q \\ \quad \text{where } \forall\, i \leq max(\#tr_P, \#tr_Q) \bullet \\ \qquad (tr_P \& tr_Q)[i] \ \widehat{=} \ \begin{cases} \langle (tr_P[i].out \uplus tr_Q[i].out), tr_P[i].gs \rangle \\ \qquad \text{if } (i \leq min(\#tr_P, \#tr_Q)) \\ tr_P[i] \quad \text{if } (\#tr_Q < i \leq \#tr_P) \\ tr_Q[i] \quad \text{if } (\#tr_P < i \leq \#tr_Q) \end{cases} \end{array}$$

If P or Q diverges, $P\|Q$ also diverges. The state after the divergence is arbitrary. Let tr be a sequence, $\#tr \geq n$. We define $tr{\uparrow}n$ as the prefix of tr with length n. The definition of the combination of two states is as follows.

Definition. Let (s_P, ok_P) and (s_Q, ok_Q) represent the states of a pair of parallel partners P and Q respectively at a given moment. The pair (s, ok) is a combination of (s_P, ok_P) and (s_Q, ok_Q) iff

$$\begin{array}{l} (ls = ls_P \oplus ls_Q \ \wedge \ gs = gs_P = gs_Q \ \wedge \ out = out_P \& out_Q \ \wedge \ tr \in tr_P \& tr_Q \\ \ \wedge\ ok = \mathbf{true}) \\ \lhd \ ok_P \wedge ok_Q \ \rhd \\ (((\neg ok_P \Rightarrow tr \geq (tr_P \& tr_Q){\uparrow}\#tr_P) \vee (\neg ok_Q \Rightarrow tr \geq (tr_P \& tr_Q){\uparrow}\#tr_Q)) \ . \end{array}$$