Alcom-FT Technical Report Series
ALCOMFT-TR-01-156

Algebraic rules for reasoning about BSP programs ⋆

A. Stewart[1], M. Clint[1], J. Gabarró[2]

[1] Department of Computer Science, The Queen's University of Belfast
Belfast BT7 1NN, N. Ireland
[2] Dep. LSI, Universitat Politècnica de Catalunya, Barcelona
Pau Gargallo 5, 08028-Barcelona, Spain
gabarro@lsi.upc.es

Abstract. A collection of algebraic laws describing the behaviour of BSP programming structures is developed. Asynchronous communications are treated as assignments to a message relation and, as such, obey assignment laws. Supersteps are identified as compositions of independent processes followed by a *barrier* operation (global synchronisation) which ensures that ongoing asynchronous communications are resolved; *barrier* is treated as a generalised form of multiple assignment. The model has similarities to that of He et al. [5] - for example, the construction of communication bundles. However, He's system is based on a trace model [7] while the approach taken here is grounded on multiple assignment and independent parallel composition. The derivation, from its sequential representation, of a parallel BSP-style version of Floyd's shortest path algorithm is used to illustrate the use of the proposed algebraic laws.

Keywords Algebraic laws; synchronization barriers; BSP; Constructive methods; Parallel programming

1 Introduction

It is widely accepted that the construction of efficient and reliable parallel programs is an order of magnitude more complex than the development of corresponding sequential software. A key cause of complexity in reasoning about parallel programs is non-determinism [2]. In a shared memory environment non-determinism may arise from two processes updating the same variable; the outcome of the computation depends on the order in which the variable is updated. This cannot be pre-determined and, consequently, non-deterministic reasoning (interleaved semantics) must be adopted. Each non-deterministic branch of the computation is open to further, similar, non-deterministic sub-division. Thus, the degree of non-determinism in a computation may grow rapidly.

In contrast, the scope for non-determinism is severely limited in the BSP (bulk synchronous parallel) model of computation [17, 4, 11]. The basic BSP computational structure is the superstep. All processes are synchronised at the start and end of a superstep. A process has exclusive access to a data subset and may communicate information with other processes through an asynchronous message transfer mechanism.

⋆ Partially supported by ACI HB1999-0093, and CICYT under grant TIC1999-0754-C03-02 and by the EC project ALCOM-FT, IST-1999-14186.

Crucially, messages sent during one superstep cannot be utilised by the recipient in the same superstep. Thus, conceptually, message delivery can be considered to occur at superstep termination. Non-determinism may only arise through communication conflict (two or more processes sending different values to the same storage address). Thus, an asynchronous communication can, at worst, increase the degree of non-determinism within a superstep by one. Note that the conflict does not affect the internal state until superstep termination; it is this factor which limits the degree of non-determinism.

Further, the fundamental role of global synchronisation in BSP aids the construction of correct programs. A proof theory for a general reactive system may involve a natural two level semantics: an *a priori* semantics given to isolated processes and a composite semantics given to the entire system. In contrast, synchrony permits global program properties to be established and this has the effect of simplifying the associated reasoning framework.

The limited nature of non-determinism and the use of synchrony in BSP suggests that the model may provide a suitable basis for the development of scientific parallel programs. The goal of this paper is to make explicit laws [13] for reasoning about BSP programs thereby subjecting the model, as a basis for software development, to critical evaluation.

2 Algebraic Laws

In this section a collection of laws which govern the behaviour of BSP programs is developed. The specification/programming notation used is based on the framework developed in Unifying Theories of Programming [8]. Communications are treated as buffer updates and, as such, obey assignment laws. Superstep composition can be expressed in terms of parallel independent composition and a *barrier* operation which delivers buffered messages to their destinations. Laws for independent composition are well known while *barrier* may be treated as a special form of multiple assignment. The proposed model is consistent with Valiant's original definition of BSP [17].

Let:
S, S_1 etc. denote superstep processes;
x, y etc. denote process variables;
v etc. denote a list of all process variables;
e, f etc. denote expressions; and
$\alpha(S)$ denote the alphabet of S [1].

Let $\|^{ss}$ denote the superstep composition operator. Consider the superstep $S_1 \|^{ss} S_2$ where S_1 and S_2 have exclusive access to *persistent* data components $\alpha(S_1)$ and $\alpha(S_2)$ respectively. A necessary precondition for the composition $S_1 \|^{ss} S_2$ to be made is that $\alpha(S_1) \cap \alpha(S_2) = \{\}$.

A superstep process is constructed using sequential programming operators and two asynchronous communication instructions *put* and *get*: these denote, respectively, a future assignment of the value of a local expression to a non-local destination and

[1] The alphabet of a process S is normally defined to be the set of free variables in S. This convention is relaxed in the context of communication statements - see later.

a request for non-local information. For example, $put(x,3)$ sends the value 3 to non-local variable x and $get(y, a*b)$ requests the future evaluation the non-local expression $a * b$ and the subsequent assignment of the resulting value to (local variable) y. In both cases communication is constrained so that current superstep computations are unaffected: rather, communication delivery is resolved at superstep termination.

The current state of ongoing communications is recorded by an auxiliary variable m, a relation from variable names (message destinations) either to values or syntactic expressions (message content).

$$m : VAR \mapsto (VAL \cup EXP)$$

where VAR denotes the space of variable addresses, VAL denotes the space of values and EXP denotes the space of expressions. The relation m is used to record all communications that are invoked in a superstep. Inside the composition $S_1 \parallel^{ss} S_2$, the data spaces of S_1 and S_2 are augmented by local message relations, $1.m$ and $2.m$ respectively. A message relation may be constructed from a sequence of *get* and *put* instructions. For example,
$1.m = \{\mapsto\}$ is the initial message state of S_1;
$1.m = \{\text{“}x\text{”} \mapsto 3\}$ is the message relation of S_1 resulting from the instruction $put(x,3)$; and
$2.m = \{\text{“}y\text{”} \mapsto \text{“}a*b\text{”}\}$ is the message relation of S_2 generated by the instruction $get(y, a*b)$.
More precisely [2], [3] :

$$put(x,e) \quad =_{df} \quad m := m \cup \{\text{“}x\text{”} \mapsto e\}$$

$$get(x,e) \quad =_{df} \quad m := m \cup \{\text{“}x\text{”} \mapsto \text{“}e\text{”}\}$$

where e is an expression not involving array elements and “e” denotes the syntactic string e rather than the value denoted by e. Thus, a *put* instruction comprises a (local) expression evaluation and the placement of the result in the message buffer m whereas a *get* instruction requests a future evaluation of the non-local string “e” [4].

[2] The meaning of a *put* statement comprises an assignment of an expression value e; as such, it includes a definedness check $\mathcal{D}(e)$ in the underlying model [8]. A *get* statement involves a *delayed* expression evaluation at the next *barrier*; as such, a *get* statement need not involve a definedness check while *barrier* should. In practice [6], communication expressions are often restricted to simple variables.

[3] The alphabet of $put(x,e)$ is defined to be $\alpha(e)$ and the alphabet of $get(x,e)$ is defined to be $\{x\}$.

[4] Note: syntactic strings “e” should be treated as constants in algebraic laws. Using multiple assignment:

$$(y := 4; m := m \cup \{\text{“}x\text{”} \mapsto y\}) \quad = \quad (y, m := 4, m \cup \{\text{“}x\text{”} \mapsto 4\})$$

whereas

$$(y = 4; m := m \cup \{\text{“}x\text{”} \mapsto \text{“}y\text{”}\}) \quad = \quad (y, m := 4, m \cup \{\text{“}x\text{”} \mapsto \text{“}y\text{”}\})$$

These instructions relax the convention that assignments have variable names on the LHS and values on the RHS (see the treatment of both x and e on the RHS of the message assignment in the $get(x, e)$ above). In essence m acts as a holder for both actual values (e) and non-local evaluation requests ("e").

The relation union operator ($\cup$) is both associative and commutative. Thus, adjacent communication statements commute under sequential composition:

$$put(x, e) \ ; \ put(y, f) \quad = \quad put(y, f) \ ; \ put(x, e) \tag{L1}$$

Further, the RHS of the *get* instruction involves an update which is independent of the current computational state (i.e. "e" and "x" need not be evaluated). Consequently,

$$S \ ; \ get(x, e) \quad = \quad get(x, e) \ ; \ S \tag{L2}$$

The need to distinguish between an expression as a string and its associated value is further apparent in the case of array based communications. For example, consider the instruction $get(x, a[i])$. Here, $a[i]$ is assumed to be a non-local array reference (to be evaluated in a later state) whereas i is assumed to be a local index expression to be evaluated in the current state. The notation $e(i)$ is used to separate non-local name components (e) from components (i) which stand for their underlying values. The notation $P \lhd b \rhd Q$ is used to denote a process that behaves like P if b holds and like Q otherwise [8]. Then, [5]

$$put(x[i], e) \quad =_{df} \quad m := m \cup \{\text{``}x\text{''}[i] \mapsto e\}$$

$$get(x[i], e(i)) \quad =_{df} \quad m := m \cup \{\text{``}x\text{''}[i] \mapsto \text{``}e\text{''}(i)\}$$

The algebraic rules above - together with termination or divergence proofs [1] [8] (see §3, for example) - permit reasoning about *individual* superstep processes. Further rules are needed to reason about supersteps. The characteristic feature of a superstep

[5] Conventionally, an assignment to an array element is expressed as an assignment to the entire array in order to ensure that names appear on the LHS and expressions appear on the RHS:

$$x[i] := e \quad = \quad x := (x : i; e)$$

where

$$(x : i; e)[j] \quad = \quad e \lhd i = j \rhd x[j]$$

It is possible to re-express the definitions of array based communications in similar ways:

$$put(x[i], e) \quad = \quad m := m \cup \{\text{``}x\text{''} \mapsto (x : i; e)\}$$

$$get(x[i], e(i)) \quad = \quad m := m \cup \{\text{``}x\text{''} \mapsto (x : i; \text{``}e\text{''})\}$$

However, this rewriting does not circumvent the problem of separating names and values. Consequently, we retain the earlier notations.

is the merging of individual process computations. Assume that processes S_1 and S_2 are such that all asynchronous communication statements have been replaced by their definitions (updates of m). Then,

$$\begin{array}{rl} S_1 \parallel^{ss} S_2 & =_{df} \text{ var } \ 1.m, 2.m := \{\mapsto\}, \{\mapsto\} \\ & \qquad ((S_1)^m_{1.m} \parallel^I (S_2)^m_{2.m}) \ ; \ barrier \\ & \quad \text{end } \ 1.m, 2.m \end{array}$$

Here, $S^m_{1.m}$ denotes S with all free occurences of m replaced by $1.m$ (i.e. a local copy of m, $1.m$, is made within S), $\parallel^I$ denotes the independent parallel composition operator and *barrier* denotes a merge operation. The definition of $S_1 \parallel^{ss} S_2$ allows a separation of concerns to be made between (i) computation and message construction and (ii) message delivery. The laws for reasoning about $\parallel^I$ are well known:

$$S_1 \parallel^I S_2 \quad = \quad S_1; S_2 = S_2; S_1 \tag{L3}$$

The merge operation *barrier* is defined as a multiple conditional assignment:

$$barrier \quad =_{df} \quad \forall \text{“}z\text{”} \in dom(1.m \cup 2.m).\ z := \sqcap(1.m \cup 2.m)(z)$$

Here, $\sqcap$ denotes non-deterministic selection from a set - thus, $\sqcap T \in T$. Note that a message $\sqcap m(z)$ may either be a value or an expression. In both cases $\sqcap m(z)$ appears on the RHS of an assignment and so is evaluated in the current *barrier* state. Alternatively, *barrier* can be expressed as a total assignment:

$$barrier \quad =_{df} \quad v := \sqcap(1.m \cup 2.m)(\text{“}v\text{”}) \triangleleft \text{“}v\text{”} \in dom(1.m \cup 2.m) \triangleright v$$

where v denotes a list of all program variables from $\alpha(barrier) - \{1.m, 2.m\}$. The message delivery component of *barrier* is a form of data parallel assignment in which a subset of variables receiving messages are simultaneously updated: a variable x which is sent one or more messages through m is overwritten at the *barrier* by non-deterministically selecting and evaluating one of its message contents. However, in cases where m is a function, *barrier* is deterministic.

A consequence of the definition of superstep merging is (since m is invariant through *barrier* assignment):

$$barrier \ ; \ barrier \quad = \quad barrier \tag{L4}$$

An empty message relation, m, has the effect of reducing *barrier* to the empty statement, π:

$$m = \{\mapsto\} \Rightarrow \quad barrier \quad = \quad \pi \tag{L5}$$

Furthermore, it follows (from the commutativity of $\parallel^I$) that superstep composition is commutative:

$$S_1 \parallel^{ss} S_2 \quad = \quad S_2 \parallel^{ss} S_1 \tag{L6}$$

and that *get* operations can be re-expressed in terms of *put* operations:

$$((S_1 \ ; \ put(y, e)) \parallel^{ss} S_2) \quad = \quad (S_1 \parallel^{ss} (S_2 \ ; \ get(y, e))) \tag{L7}$$

The notation $\|^{ss}\ (S_1, \ldots, S_n)$ is used to denote supersteps constructed from n, $n \geq 2$, processes.

$$\begin{aligned} \|^{ss}\ (S_1, \ldots, S_n) \quad =_{df}\ & \mathrm{var}\ \ 1.m, \ldots, n.m := \{\mapsto\}, \ldots, \{\mapsto\} \\ & \qquad (\|^{I}\ ((S_1)^{m}_{1.m}, \ldots, (S_n)^{m}_{n.m}))\ ;\ \mathit{barrier} \\ & \mathrm{end}\ \ 1.m, \ldots, n.m \end{aligned}$$

Note that $\|^{ss}$ is not associative. Thus, in general,

$$(S_1 \|^{ss} (S_2 \|^{ss} S_3)) \quad \neq \quad ((S_1 \|^{ss} S_2) \|^{ss} S_3)$$

The loss of associativity is due to the deletion of information about the message relations at superstep termination. In $((S_1 \|^{ss} S_2) \|^{ss} S_3)$ the message relation $(1.m \cup 2.m)$ defines the context of the first *barrier* while $3.m$ defines the context of the second. However, in $(S_1 \|^{ss} (S_2 \|^{ss} S_3))$, $(2.m \cup 3.m)$ defines the context of the first *barrier* while $1.m$ defines the context of the second. Strictly speaking, expressions like $((S_1 \|^{ss} S_2) \|^{ss} S_3)$ lie outside the scope of the BSP model which requires a global synchronisation across *all* processes (rather than a local synchronisation which excludes S_3 as in the example above). Note that it is *global* synchronisation which offers a mechanism for relating the progress of different processes within a superstep thereby facilitating the derivation of a complexity framework for predicting execution times of BSP programs. Under the reasonable assumption that S_1 and S_2 do not communicate with variables of S_3 then $((S_1 \|^{ss} S_2) \|^{ss} S_3)$ can be re-expressed as a flat superstep:

$$((S_1 \|^{ss} S_2) \|^{ss} S_3) \quad = \quad (\|^{ss}\ (S_1, S_2, \pi_{\alpha(S_3)})) \quad ; \quad (\|^{ss}\ (\pi_{\alpha(S_1)}, \pi_{\alpha(S_2)}, S_3))$$

where π_z is the empty statement with alphabet $\{z\}$. The scope of a superstep may be extended as follows:

$$S \quad ; \quad (S_1 \|^{ss} S_2) \quad = \quad ((S \quad ; \quad S_1) \|^{ss} S_2) \tag{L8}$$

where $\alpha(S) \cap \alpha(S_2) = \{\}$ and S does not contain *barrier* operations (supersteps). However, in general,

$$(S_1 \|^{ss} S_2) \quad ; \quad S \quad \neq \quad ((S_1 \quad ; \quad S) \|^{ss} S_2)$$

where $\alpha(S) \cap \alpha(S_2) = \{\}$ (on the LHS, S is the last statement to be executed whereas *barrier* is the last on the RHS). Supersteps obey the following distribution law:

$$(S_1 \lhd b \rhd S_2) \|^{ss} S_3 \quad = \quad (S_1 \|^{ss} S_3) \lhd b \rhd (S_2 \|^{ss} S_3) \tag{L9}$$

Later, the derivation from its sequential form of a BSP program is developed. In order to support this transformation a number of auxiliary laws are now developed. Firstly, the correspondence between a multiple assignment and a superstep containing only communication statements is addressed:

$$1.x, \ldots, N.x := 1.e, \ldots, N.e \quad = \quad \|^{ss}_{1 \leq r \leq N}\ \mathit{get}(r.x, r.e) \tag{L10}$$

The law may be used to transform a set of global assignments into communication statements.
Proof:

$\|^{ss}_{1 \leq r \leq N}\ get(r.x, r.e)$

$=$ < by the definition of get >

$\|^{ss}_{1 \leq r \leq N}\ m := m \cup \{``r.x" \mapsto ``r.e"\}$

$=$ < by the definition of superstep >

var $1.m, \ldots, N.m := \{\mapsto\}, \ldots, \{\mapsto\}$
  $\|^{I}_{1 \leq r \leq N}\ r.m := r.m \cup \{``r.x" \mapsto ``r.e"\}$;
  barrier ;
end $1.m \ldots N.m$

$=$ < by the definition of *barrier* with v denoting the list of all program variables in the set $\alpha(barrier) - \{1.m, \ldots, N.m\})$ >

var $1.m, \ldots, N.m$
  $1.m, \ldots, N.m := \{1.x \mapsto 1.e\}, \ldots, \{N.x \mapsto N.e\}$;
  $v := \sqcap(1.m \cup \ldots \cup N.m)(``v") \lhd ``v" \in dom(``1.x" \cup \ldots \cup ``N.x") \rhd v$;
end $1.m \ldots N.m$

$=$ < by the definitions of multiple conditional assignment and local declaration >

$1.x, \ldots, N.x := 1.e, \ldots, N.e$

$\square$

Secondly, it is useful, in the development of efficient supersteps, to extend the scope of a message declaration m to include a prefix process S which is independent of m [8]:

$$S \text{ ; var } m \text{ ; } T \text{ ; end } m \quad = \quad \text{var } m \text{ ; } S \text{ ; } T \text{ ; end } m$$

where $m \notin \alpha(S)$.

Finally, a composition of an independent parallel composition and a superstep may be simplified provided that the independent processes do not refer to m:

$$(\|^{I}_{1 \leq r \leq N}\ S_r); (\|^{ss}_{1 \leq r \leq N}\ T_r) \quad = \quad \|^{ss}_{1 \leq r \leq N}\ (S_r; T_r) \qquad \text{(L11)}$$

where $\{1.m, \ldots N.m\} \cap \alpha(S_i) = \{\} \wedge \alpha(S_i) \subseteq \alpha(T_i)$, $1 \leq i \leq N$

Proof:

$(\|^{I}_{1\leq r\leq N} S_r); (\|^{ss}_{1\leq r\leq N} T_r)$
$= <$ by the definition of superstep $>$
$\|^{I}_{1\leq r\leq N} S_r;$
var $1.m, \ldots, N.m := \{\mapsto\}, \ldots, \{\mapsto\}$
$\quad (\|^{I}_{1\leq r\leq N} T_r);\ barrier\ ;$
end $1.m \ldots N.m$
$= <$ by the extension of message declarations $>$

var $1.m, \ldots, N.m := \{\mapsto\}, \ldots, \{\mapsto\}$
$\quad \|^{I}_{1\leq r\leq N} S_r;$
$\quad (\|^{I}_{1\leq r\leq N} T_r);\ barrier\ ;$
end $1.m \ldots N.m$

$= <$ by the definition of superstep $>$

$\quad \|^{ss}_{1\leq r\leq N} (S_r; T_r)$

$\square$

A superstep may be transformed into a (sequential) normal form [8] thereby allowing the equality of computations to be tested. Hoare and Jifeng [8] define a series of increasingly complex normal forms depending on the nature of the underlying computations. In the context of this paper the simplest normal form is total assignment:

$$x, y, \ldots, z, m := e, f, \ldots, g, r$$

where $x, y, \ldots, z$ is a list of all program variables, m is a message relation and $e, f, \ldots, g, r$ is a list of expressions with r denoting a relation. It is assumed that m can be directly assigned a relation r and that *get* and *put* instructions can be re-expressed, using their definitions, as assignments to m. In this way individual processes can be reduced to an appropriate normal form - see Hoare and Jifeng [8] for further details.

Supersteps have the effect of "delivering" the message relation m at a *barrier*, with communication conflicts being resolved non-deterministically. A non-deterministic total assignment has the form:

$$x, y, \ldots, z, m := e, f, \ldots, g, r_1 \quad \sqcap \quad x, y, \ldots, z, m := h, i, \ldots, j, r_2$$

A superstep composition $S_1 \|^{ss} S_2$ may be transformed into sequential form using its definition. Then the *barrier* operation can be converted to non-deterministic normal form by rewriting it using its definition. The remainder of the components used in the rewrite (local declaration, sequential computation) obey well known laws and can be reduced to normal form [8]. Thus, a normal form for superstep composition can be generated.

3 Operational illustrations

In this section some semantic consequences of the superstep model are highlighted using some simple examples. In particular, the limited non-deterministic behaviour of supersteps is illustrated. Example 1 shows a deterministic superstep:

Example 1. Consider $\mathcal{NO\_RACE} = S_1 \parallel^{ss} S_2$ where two variables x and y are updated and read.

$$
\begin{array}{llll}
S_1 =_{df} & x := 1; & S_2 =_{df} & y := 4; \\
 & get(x, y); & & get(y, x); \\
 & x := x + 1 & & y := y + 2
\end{array}
$$

with $\alpha(S_1) = \{x\}$ and $\alpha(S_2) = \{y\}$.
Let σ and m be, respectively, the final state and message relation after execution of $S_1 \parallel^I S_2$:

$$
\begin{aligned}
\sigma &=_{df} 1.\sigma \cup 2.\sigma = \{x \mapsto 2, y \mapsto 6\} \\
m &=_{df} 1.m \cup 2.m = \{\text{“}x\text{”} \mapsto \text{“}y\text{”}, \text{“}y\text{”} \mapsto \text{“}x\text{”}\}
\end{aligned}
$$

Since m is a function, execution of *barrier* gives the *unique* state:

$$
\begin{aligned}
& \lambda z.\sigma(m(z)) \lhd z = (x \vee y) \rhd \sigma(z) \\
= & \{x \mapsto \sigma(y), y \mapsto \sigma(x)\} \\
= & \{x \mapsto 6, y \mapsto 2\}
\end{aligned}
$$

Note that superstep and interleaving semantics provide different outcomes for this example. Consider an interleaved semantics of $\mathcal{RACE} = S_1' \parallel^I S_2'$ where two shared variables x and y are updated and read.

$$
\begin{array}{llll}
S_1' =_{df} & x := 1; & S_2' =_{df} & y := 4; \\
 & x := y; & & y := x; \\
 & x := x + 1 & & y := y + 2
\end{array}
$$

An interleaved semantics is based on all possible computational sequences which may arise: in some cases, the progress of S_1' may outstrip that of S_2' leading to particular outcomes; alternatively, the progress of S_2' may outstrip that of S_1' leading to alternative outcomes. As a result, for this example, non-deterministic behaviour is exhibited under interleaving semantics.

Example 2 illustrates the case of concurrent writes to a single location; here the degree of non-determinism in the system is given by the number of conflicting write statements (and bounded by the total number of superstep communications):

Example 2. Let $\mathcal{CW} = S_1 \parallel^{ss} S_2$ where:

$$
\begin{array}{llll}
S_1 =_{df} & x := 0; & S_2 =_{df} & put(x, 1); \\
 & get(x, y) & & y, z := 10, 20 \\
 & get(x, z) & &
\end{array}
$$

with $\alpha(S_1) = \{x\}$ and $\alpha(S_2) = \{y, z\}$. After execution of the independent composition of $S_1 \parallel^I S_2$ the following state and message relation are produced:

$$
\begin{aligned}
\sigma &=_{df} 1.\sigma \cup 2.\sigma = \{x \mapsto 0, y \mapsto 10, z \mapsto 20\} \\
m &=_{df} 1.m \cup 2.m = \{\text{“}x\text{”} \mapsto \text{“}y\text{”}, \text{“}x\text{”} \mapsto \text{“}z\text{”}, \text{“}x\text{”} \mapsto 1\}
\end{aligned}
$$

Process merging yields the state:

$$
\begin{aligned}
& \lambda z.1 \sqcap \sigma(y * z) \lhd z = x \rhd \sigma(z) \\
= & \{y \mapsto 10, z \mapsto 20, x \mapsto 1 \sqcap 10 \sqcap 20\}
\end{aligned}
$$

with the assignment to x exhibiting three-way non-determinism.

Example 3 illustrates the behaviour of a divergent (non-terminating) superstep.

Example 3. Consider the program $\mathcal{NO\_DELIVERY} = S_1 \|^{ss} S_2$ where:

$$S_1 =_{df} \quad \begin{array}{l} x := 0; \\ \text{while } true \text{ do} \\ \quad put(y, x); x := x + 1 \\ \text{od} \end{array} \qquad\qquad S_2 =_{df} \quad \pi_y$$

On execution the following message relation is constructed:

$$1.m = \{\text{“}y\text{”} \mapsto 0 \sqcap 1 \sqcap 2 \sqcap 3, \cdots\}$$

Here, m is an example of unbounded (non-terminating) non-determinism. Letting $\perp$ denote a divergent process within the framework of [8] it follows that:

$$\perp \; ; \; S \quad = \quad \perp$$

Furthermore, from the definition of $\|^I$:

$$(\perp \quad \|^I \quad S) \quad = \quad \perp$$

Thus,

$$(\perp \quad \|^{ss} \quad S) \quad = \quad \perp$$

$\mathcal{NO\_DELIVERY}$ is an example of a computation which attempts to construct an unbounded message but never delivers it.

Example 4 illustrates the non-associativity of superstep composition; here the order of composition determines the order of communications:

Example 4.

$$\begin{array}{lclcl} (put(y,3) \|^{ss} (\pi_z \|^{ss} put(y,4))) & = & (put(y,3) \|^{ss} y := 4))) & = & y := 3 \\ ((put(y,3) \|^{ss} \pi_z) \|^{ss} put(y,4)) & = & (y := 3 \|^{ss} put(y,4)) & = & y := 4 \end{array}$$

4 Derivation of a superstep based implementation of Floyd's shortest path algorithm

In this section the derivation, from its sequential representation, of a parallel BSP version of Floyd's algorithm for computing the shortest paths between all pairs of N vertices in a given weighted graph G is sketched. This algorithm has been widely studied; parallel versions [9] and distributed versions [16] are well known.

It is assumed that the weights of the edges of G are:

$$\forall i, j : 1 \leq i, j \leq N \text{: } w(i,j) = \begin{cases} 0 & i = j \\ 1 & (i,j) \text{ is an edge of G} \\ \infty & \text{otherwise} \end{cases}$$

A (sequential) realisation of Floyd's algorithm is given below:

$\{\forall i,j : 1 \le i,j \le N : d(i,j) = w(i,j)\}$
var k, r, s ;
for $k := 1$ to N do
(for $r := 1$ to N do
(for $s := 1$ to N do
$d(r,s) := \min\{d(r,s), d(r,k) + d(k,s)\}$) ;) ;
end k, r, s
$\{\forall i,j : 1 \le i,j \le N : d(i,j) =$ minimum distance between i and $j\}$

To derive a BSP implementation of Floyd's algorithm it is instructive to consider how the computation may be split over concurrent processes. Consider the following row partitioning of d: let process P_r, $1 \le r \le N$, update the rth row of d with $\{d(r, 1 \ldots N)\} \in \alpha(P_r)$. Observe that row k of d is utilised in the update of row r:
$d(r,s) := \min\{d(r,s), d(r,k) + d(k,s)\}$.
First, the properties of row k and the main diagonal of d (in the kth iteration of the outer loop) are investigated. Observe that:
$\forall r,s : 1 \le r,s \le N : d(r,s) \ge 0 \wedge d(r,r) = 0 \Rightarrow$
$(d(r,r) \;\; := \;\; \min\{d(r,r), d(r,k) + d(k,r)\}) = \pi$

< by the definition of min and π >

and
$\forall k : 1 \le k \le N : d(k,k) = 0 \Rightarrow (d(k,s) := \min\{d(k,s), d(k,k) + d(k,s)\} = \pi)$

< since $\min(a,a) = a$ >

Thus, the computation in the doubly-nested inner loop may be re-expressed as:

$$\pi \lhd (r = k \vee s = r) \rhd d(r,s) \;\; := \;\; \min\{d(r,s), d(r,k) + d(k,s)\} \qquad (*)$$

< using the identities above for $d(k,s)$ and $d(r,r)$ >
Since row k of d is invariant in iteration k any reference to row k of d can be replaced by a reference to a new local variable which is initialised at the outset of the iteration block. This rewrite strategy can be extended to introduce N copies of row k of d, $1.t$, ..., $N.t$, as follows:

var $1.t$, ... , $N.t$
for $k := 1$ to N do
$1.t$, ... , $N.t := d(k, 1 \ldots N)$, ... , $d(k, 1 \ldots N)$;
$\vdots$
... $d(r,s) := \min\{d(r,s), d(r,k) + r.t(s)\}$;
end $1.t$, ... , $N.t$

< from assignment laws:
(var t ; $t := d$; $x := t$; end t) = (var t ; $t, x := d, d$; end t) = $x := d$ >

In a similar way the column index variable s can be spawned into N multiple copies, $1.s$, ..., $N.s$. In this way each row calculation can be made independent:

var $1.t, \ldots, N.t,\ 1.s, \ldots, N.s$
    (for $k := 1$ to N do
        $1.t\ ,\ \ldots\ ,\ N.t := d(k, 1 \ldots N)\ ,\ \ldots\ ,\ d(k, 1 \ldots N)$;
        $\|^{I}_{1\le r\le N}$ (for $r.s := 1$ to N do $d(r, r.s) := \min\{d(r, r.s), d(r, k) + r.t(r.s)\}$));
end $1.t \ldots k.t,\ 1.s, \ldots, N.s$

$<$ by unfolding, the definition of $\|^{I}$ and using the rewrite (*) again $>$
The next stage in the derivation is the conversion of the assignments to $1.t$, …, $N.t$ into communication statements. Applying Law L10 we get:

var $1.t, \ldots, N.t,\ 1.s, \ldots, N.s$
    for $k := 1$ to N do (
        var $1.m, \ldots, N.m := \{\mapsto\}, \ldots, \{\mapsto\}$
            $\|^{I}_{1\le r\le N}$ $get(r.t\ ,\ d(k, 1 \ldots N)\ )$; *barrier* ;
        end $1.m \ldots N.m$;
        $\|^{I}_{1\le r\le N}$ (for $r.s := 1$ to N do $d(r, r.s) := \min\{d(r, r.s), d(r, k) + r.t(r.s)\}$) ;
) ;
end $1.t \ldots N.t,\ 1.s, \ldots, N.s$

The outer loop may be unfolded to facilitate the recasting of parallel computations as efficient supersteps:

var $1.m, \ldots, N.m := \{\mapsto\}, \ldots, \{\mapsto\}$
    ($\|^{I}_{1\le r\le N}$ $get(r.t\ ,\ d(1, 1 \ldots N)\ )$); *barrier* ;
end $1.m, \ldots, N.m$;
for $k := 1$ to $N - 1$ do (
    $\|^{I}_{1\le r\le N}$ (for $r.s := 1$ to N do ($d(r, r.s) := \min\{d(r, r.s), d(r, k) + r.t(r.s)\}$));
    var $1.m, \ldots, N.m := \{\mapsto\}, \ldots, \{\mapsto\}$
        ($\|^{I}_{1\le r\le N}$ $get(r.t\ ,\ d(k+1, 1 \ldots N)\ )$); *barrier* ;
    end $1.m \ldots N.m$
) ;
$\|^{I}_{1\le r\le N}$ for $r.s := 1$ to N do $d(r, r.s) := \min\{d(r, r.s), d(r, N) + r.t(r.s)\}$
$<$ by unfolding and refolding the outer for loop $>$

The scope of the message declaration can be extended using Law L11. Thus, the inner loop of Floyd's algorithm is equivalent to:

var $1.m, \ldots, N.m := \{\mapsto\}, \ldots, \{\mapsto\}$
    $\|^{ss}_{1\le r\le N}$ (for $r.s := 1$ to N do ($d(r, r.s) := \min\{d(r, r.s), d(r, k) + r.t(r.s)\}$) ;
            $get(r.t\ ,\ d(k+1, 1 \ldots N))$);
end $1.m \ldots N.m$

Finally, the following (parallel) program is produced:

var $k,\ 1.t, \ldots, N.t,\ 1.s, \ldots, N.s$
($\|^{ss}_{1\le r\le N}$ $get(r.t\ ,\ d(1, 1 \ldots N)\ )$) ;
(for $k := 1$ to $N - 1$ do
    ($\|^{ss}_{1\le r\le N}$ (for $r.s := 1$ to N do ($d(r, r.s) := \min\{d(r, r.s), d(r, k) + r.t(r.s)\}$);
            $get(r.t\ ,\ d(k+1, 1 \ldots N))$) ;) ;
$\|^{I}_{1\le r\le N}$ for $r.s := 1$ to N do ($d(r, r.s) := \min\{d(r, r.s), d(r, N) + r.t(r.s)\}$);
end $k,\ 1.t, \ldots, N.t,\ 1.s, \ldots, N.s$

5 Efficiency Considerations

The superstep computational model comprises a sequence of *fork and join* operations - one for each superstep - interspersed with sequential computations. Such *fork and join* models are employed in widely used implementation languages - for example, FORTRAN 90 and High Performance FORTRAN. One way to implement a superstep is to distribute data over processes immediately prior to the computation (fork) and to gather the results from the processes at superstep completion. However, such data distribution is costly; here we address the question of whether a set of supersteps can be efficiently implemented by *reusing* previous data partitions.

Consider the problem of implementing a sequential composition of supersteps:

$$(S_1 \parallel^{ss} S_1') \quad ; \quad (S_2 \parallel^{ss} S_2').$$

Ideally, this computation would be implemented by mapping S_1 and S_2 to one process (thread) (with combined alphabet $\alpha(S_1) \cup \alpha(S_2)$) and S_1' and S_2' to another thread (with combined alphabet $\alpha(S_1') \cup \alpha(S_2')$). However, this approach is valid only if the supersteps have the same number of components and the resulting (combined) process alphabets are disjoint. Alphabet disjointness can be guaranteed if the supersteps $(S_1 \parallel^{ss} S_1')$ and $(S_2 \parallel^{ss} S_2')$ are *partition consistent*:

Definition 1. *A pair of compositions* $(S_1 \parallel S_1')$, $(S_2 \parallel S_2')$ *are partition consistent iff* $\alpha(S_1) \cap \alpha(S_2') = \{\} \wedge \alpha(S_2) \cap \alpha(S_1') = \{\}$

By the definition of partition consistency and from the disjointness of superstep processes:

Lemma 1. *A pair of partition consistent supersteps* $(S_1 \parallel^{ss} S_1')$, $(S_2 \parallel^{ss} S_2')$ *satisfy* $(\alpha(S_1) \cup \alpha(S_2)) \cap (\alpha(S_1') \cup \alpha(S_2')) = \{\}$.

Lemma 1 guarantees that partition consistent supersteps can be converted into threads with disjoint alphabets; partition consistency can be extended in obvious ways to include compositions with more than 2 components, and sets (rather than pairs) of compositions.

Before considering how two partition consistent supersteps can be assigned to threads it is necessary to convert the global *barrier* operation into a localised form. Let *syn* denote a localised synchronisation instruction and $\parallel^{BSP}$ denote a thread composition operator. Then the relationship between the global superstep operations and their localised counterparts is given by:

$$(S_1 \parallel^{ss} S_2) \quad = \quad (S_1 \; ; \; syn) \parallel^{BSP} (S_2 \; ; \; syn) \tag{L13}$$

Note that *syn* cannot appear in either S_1 and S_2. Partition consistent threads $(T_1 \parallel^{BSP} T_2)$, $(T_3 \parallel^{BSP} T_4)$ can be merged as follows:

$$(T_1 \parallel^{BSP} T_2) \; ; \; (T_3 \parallel^{BSP} T_4) \quad = \quad (T_1 \; ; \; T_3) \parallel^{BSP} (T_2 \; ; \; T_4) \tag{L14}$$

Unfortunately, the supersteps in the BSP version of Floyd's algorithm are not partition consistent. Consider the superstep embedded in the *for* loop:

$\vdots$

(for $k := 1$ to $N-1$ do
    $(\|^{ss}_{1\le r\le N}$ (for $r.s := 1$ to N do $(d(r,r.s) := \min\{d(r,r.s), d(r,k)+r.t(r.s)\}$);
        $get(r.t\ ,\ d(k+1, 1\ldots N))$) ;) ;

Here the variable k occurs in each of the superstep process alphabets. In order to derive a partition consistent program it is necessary to distribute a localised copy of the for structure (with a uniquely named control variable) to each of the processes. In general, the computation:

for $k := a$ to b do
    $(T_1(k)\ \|^{BSP}\ T_2(k))$

can be re-expressed as:

(var $1.k$
    for $1.k := a$ to b do
        $T_1(1.k)$
end $1.k$)
	$\|^{BSP}$	(var $2.k$
		    for $2.k := a$ to b do
		        $T_2(2.k)$
		end $2.k$)

where $1.k$ and $2.k$ are new names. Thus, the for loop in the BSP version of Floyd's algorithm may be converted to:

$\|^{BSP}_{1\le r\le N}$
    for $r.k := 1$ to $N-1$ do
        (for $r.s := 1$ to N do $(d(r,r.s) := \min\{d(r,r.s), d(r,r.k)+r.t(r.s)\}$);
             $get(r.t\ ,\ d((r.k)+1, 1\ldots N))$; syn
)

Here the alphabet of the rth superstep process is: $\{d(r,1\ldots N), r.k, r.s, r.t\}$. Similarly, the alphabets of the rth superstep process in the first and last supersteps are, respectively: $\{r.t\}$ and $\{d(r,1\ldots N), r.s, r.t\}$. It follows that these three computations are partition consistent. Law L14 can now be applied to merge the computations into threads giving:

$\|^{BSP}_{1\le r\le N}$
(var $r.k$, $r.s$, $r.t$
    $get(r.t\ ,\ d(1, 1\ldots N)$) ; syn ;
    for $r.k := 1$ to $N-1$ do
        (for $r.s := 1$ to N do $(d(r,r.s) := \min\{d(r,r.s), d(r,r.k)+r.t(r.s)\}$);
        $get(r.t\ ,\ d((r.k)+1, 1\ldots N))$; syn);
    for $r.s := 1$ to N do $d(N,r.s) := \min\{d(N,r.s), d(N,r.k)+r.t(r.s)\}$);
end $r.k$, $r.s$, $r.t$)

Partition consistency is a desirable property of superstep-based computations. Where this property is not immediately enjoyed it may be possible to use certain (compilation) techniques to transform "*inconsistent*" computations into partition consistent form. For example, consider the sequential composition:

$$x := e \; ; \; (T_1 \parallel^{BSP} T_2)$$

where $x \in \alpha(T_1) \wedge \alpha(e) \subseteq \alpha(T_2)$. The computation may be rewritten in partition consistent form as:

$$(get(x, e) \; ; \; syn \; ; \; T_1) \parallel^{BSP} (syn; T_2)$$

In §2 a semantic basis for BSP style programming was proposed without addressing the problem of realising efficient implementations. The aim here has been to redress the balance between semantics and efficiency: in particular, the importance of maintaining consistent data partitions across processes as a means of enhancing efficiency has been emphasised. Note, however, that computations which are *not* partition consistent may be mapped to threads at the possible expense of extra communication.

6 Related Work and Conclusions

The aim of this paper is to establish a *simple* reasoning framework for the development of BSP-style programs. To this end, a deliberate decision has been taken to remove low level implementation detail and, at the same time, to utilise *directly* the simplifying properties of global synchronisation. The work is now briefly evaluated with respect to:

- the standard BSP library, BSPlib [6];
- the reasoning framework of He et al [5] (see also [10]).

BSPlib is a library supporting BSP-style operations. For example, the library provides operations to:

- initialise a BSP computation;
- register an area of memory with a particular process;
- perform asynchronous communications and
- perform localised synchronisations.

Many of operations defined in this paper have counterparts in the libray; *barrier* is an exception. The superstep semantic model uses *barrier* in order to utilise the simplifying properties of global synchronisation (see below); on the other hand BSPlib provides a localised synchronisation statement, *bsp_sync* (similar to *syn*). The emphasis in the work reported is on the *simplicity* of the semantic framework. To this end an attempt has been made to avoid some of the implementation detail associated with BSPlib operations. For example, the BSPlib *get* operation has five parameters specifying, respectively, recipent process identifier, base address of communication data, an offset to augment the base address, a local address for storing result and the number of bytes of communication data. Here, a version of *get* is defined that avoids some of this implementation detail. It is hoped that this (simpler) model provides a basis for the design of BSP-style programs which can, if desired, be mapped on to BSPlib (or MPI - message passing interface library) implementations.

It is well known that synchrony can be employed to simplify the semantics of parallel computation [3, 12]. For example, a proof theory for a general reactive system may involve a natural two level semantics: an *a priori* semantics given to isolated processes and a *composite* semantics given to the entire system. In contrast, synchrony permits global program properties to be established *directly* and this has the effect of simplifying the associated reasoning framework. Consequently, the superstep has been selected as the basic computational unit in the proposed model. In contrast, the model of He et al [5] is based on isolating localised computations: each process is mapped on to an appropriate trace (observation) sequence which, subsequently, may be combined to determine composite behaviour. He's system has a number of advantages:

– computations are automatically partitioned consistently;
– a program consists of a single initial parallel composition of processes - the composition operator (similar to $||^{BSP}$) is associative. The trace sequences of each of the processes are available for combination by an associative trace join operation.

On the other hand He's model requires that:

– *syn* statements in separate processes must be correctly matched (using counters) in order to obtain a composite semantics (i.e. the ith *syn* statement executed in the first process must be matched with the ith *syn* statement in the second process etc.). For systems composed of processes with identical control structures matching may be straightforward. However, if the structures of the underlying processes are radically different, then matching may be a complex task.
– a two stage approach is used to determine correctness - first, process trace information is assembled and then composite behaviour is extracted.

The superstep model presented here is grounded on the concept of reasoning *directly* about BSP programs using a sequence of global state transformations. In effect, a global state transformation may be used to represent one abstract operation. Thus, for example, it is hoped that the algebraic framework proposed here provides a basis for recasting numerical linear algebraic operations directly into superstep implementations. Thus, the main aims of the work are:

– to clarify the semantics of the BSP model by providing a simple set of underlying algebraic laws; and
– provide a practical framework to aid the construction of parallel software.

7 Acknowledgment

The authors are very grateful to the three anonymous referees for their constructive comments on an earlier draft of this paper.

References

[1] Apt K.R, Olderog E-R.: Verification of Sequential and Concurrent Programs, Second Edition, Springer-Verlag, 1999.

[2] Dijkstra E. W.: Guarded commands, nondeterminacy and formal derivation of programs, Communications ACM, **18** (1975) 453–457.

[3] Elrad T., Francez N.: Decomposition of distributed programs into communication-closed layers, Science of Computer Programming, **2** (1982) 155–173.

[4] Gerbessiotis A. V., Valiant L. G.: Direct bulk-synchronous parallel algorithms, Journal of Parallel and Distributed Computing, **22** (1994) 251–267.

[5] He J., Miller Q., Chen L.: Algebraic laws for BSP programming, in Euro-Par '96, LNCS 1124, Springer-Verlag, edts: L. Bouge, P. Fraigniaud, A. Mignotte and Y. Robert (1996) 359–368.

[6] Hill 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 Computing **24**, (1998) 1947–1980.

[7] Hoare C. A. R.: Communicating Sequential Processes, Prentice Hall, 1985.

[8] Hoare C. A. R., Jefeng H.: Unifying Theories of Programming, Prentice Hall, 1998.

[9] Foster I.: Designing and building parallel programs, Addison Wesley, 1995.

[10] Lecomber D.: Methods of BSP programming, PhD dissertation, Oxford University, 1998.

[11] McColl W.F.: Scalable computing *in* Computer Science Today: Recent Trends and Developments LNCS Vol. 1000, Springer-Verlag, edt: J. van Leeuwen (1995) 46–61.

[12] Milner R.: Calculi for synchrony and asynchrony, Theoretical Computer Science, **25** (1983) 267–310.

[13] Roscoe A.W., Hoare C.A.R.: Laws of OCCAM programming, Theoretical Computer Science, **60** (1988) 177–229.

[14] Stewart A., Clint M.: Synchronising asynchronous communications *in* Euro-Par '97, LNCS 1300, Springer-Verlag, edts: C. Lengauer, M. Griebl and S. Gorlatch (1997) 511–520.

[15] Stewart A., Clint M., Gabarró J.: Axiomatic frameworks for developing BSP-style programs, Parallel Algorithms and Applications, **14** (2000) 271–292.

[16] Tel, G.:Distributed Algorithms, Cambridge, 1994.

[17] Valiant L. G.: A bridging model for parallel computation, Comm. ACM, **33** (1990) 103–111.