Alcom-FT Technical Report Series
ALCOMFT-TR-03-41

Barrier Synchronisation: axiomatisation and relaxation *

A. Stewart[1] and M. Clint[1] and J. Gabarró[2]
[1]School of Computer Science, The Queen's University of Belfast, Belfast, N. Ireland.
[2]Dep. LSI, Universitat Politècnica de Catalunya, Barcelona, Spain.

September 8, 2003

Abstract

The Bulk Synchronous Parallel (BSP) model is underpinned by a global synchronisation operation. This operation both synchronises processes and updates local stores using asynchronous communications. In this paper a generalised, non-deterministic, substitution axiom (over functions) is identified with BSP barrier synchronisation. Correctness proofs of BSP programs based on the axiom are structurally similar to proofs of correctness of sequential programs. Global synchronisation is often considered to be relatively expensive to implement on distributed architectures for applications having low levels of communications. In an extreme situation a process having no external interactions may have to wait unnecessarily at a barrier. In this paper a form of barrier operation is introduced which is straightforward to reason about and, with a view to increasing the potential for efficient implementation, permits barrier operations to be relaxed through partial uncoupling thereby allowing processes which do not need to synchronise to proceed.

Keywords. Hoare logic; high performance computing; BSP; global synchronisation; operation relaxation

1 Introduction

The raison d'être of high performance computing (HPC) is the creation of a combined software and hardware environment which facilitates the rapid parallel execution of scientific and engineering application programs. The major concern of applications programmers is execution speed. Thus, imperative programming languages are almost exclusively used in the development of HPC application software. These applications are generally non-reactive and can be specified as precondition/postcondition predicate pairs. Often parallel designs are derived from a sequential algorithm by (i) distributing data over processes, (ii) distributing, in a semantics preserving way, the computation over processes and (iii) adding communication operations to allow the acquisition of necessary non-local information.

It is much simpler to reason about a single execution sequence than a parallel composition of communicating processes (the latter involves consideration of all possible interleavings of process actions – see de Roever et al [8] for an overview of imperative trace-based proof rules). It is convenient to treat separate process actions as simultaneous events for the purpose of

*Work partially supported by: Spanish CICYT under grant TIC2002-04498-C05-03 (Tracer), IST program of the EU under contract IST-2001-33116 (Flags), Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

reasoning whilst allowing interleaved actions to occur during execution. This observation corresponds to the approach of Misra [20] where a *tight* execution strategy is assumed for reasoning about multiprograms while, with certain restrictions, a *loose* (interleaved) execution strategy is employed for implementation.

BSP [19, 23, 29] is an imperative model of parallel computation which facilitates, through enforced global synchronisations, correctness proofs in the style of those used for sequential program verification [24, 25, 26]. In particular, correctness arguments for BSP programs can be conducted by associating each *superstep* – that is a portion of (parallel) code embedded between adjacent global synchronisation operations (*barriers*) – with a predicate transformer [9] (in the same way that a sequential program may be identified with a predicate transformer). Thus, the execution of a BSP program corresponds to the generation of a sequence of global states, each of which may be verified by conventional (sequential) correctness arguments. Within a superstep interference is restricted to the operation of global synchronisation (this operation updates local stores on the basis of communications that have been generated during the superstep). Thus, global synchronisation may be identified with a form of multiple assignment.

The tightly synchronised BSP model is associated with a conventional reasoning framework. However, when executed on distributed architectures, global synchronisation may impose unnecessarily stringent requirements. Thus, for example, a process having no external interactions will be synchronised at a barrier when it could safely proceed. Barrier synchronisation may be relaxed in two respects:

- by removal of non-participating processes from barrier synchronisations; and
- a barrier synchronisation with an *explicit* inter-process connection interface offers the potential for efficient implementation.

In this paper an axiomatic reasoning framework for BSP programs is developed which is amenable to the realisation of synchronisation barriers as a collection of MPI-like [22] point-to-point communications. The first aim is to provide a framework for conducting BSP correctness arguments *directly* at the level of the programming language – it is hoped that implementers have a means of transferring efficient abstract BSP algorithms [4, 19] into correct implementations. Secondly it is hoped that the simplicity of the barrier semantics is evidence of the appropriateness of the BSP model for the expression of scientific computations.

The paper is organised as follows. A brief outline of the BSP programming model is given in §2. A substitution mechanism, which is used to give a semantics to BSP global synchronisation, is developed in §3. This mechanism is more abstract and, consequently, clearer than earlier versions of barrier substitution [24, 25, 26]. A concrete axiom for *barrier*, based on the substitution rule, is given in §4. The similarities between correctness proofs of sequential and BSP-like programs are illustrated in §5 using a Gaussian elimination example. Efficiency of execution of BSP supersteps is considered in §6 and §7. In particular, the problem of maintaining consistency of data partitions over supersteps is considered in §6. Relaxations of the coupling requirements of global synchronisation are considered in §7. Finally, some comments are made on the feasibility of relaxing barriers automatically using static analysis.

2 BSP

In this section the basic properties of a variant subset of the BSP programming model [19, 23, 29] are summarised. Let S, S_1, S_2 etc. denote superstep processes and let $\alpha(S)$, $\alpha_R(S)$, $\alpha_W(S)$ denote, respectively, the set of free variables, the set of free read-only variables and the set of free read-write variables in S; x, y etc. denote simple variables; and $\|^{ss}$ denotes the superstep composition operator. The superstep $S_1 \|^{ss} S_2$ is valid provided that its constituent processes are disjoint:

$$disjoint(S_1, S_2) \quad =_{df} \quad (\alpha_W(S_1) \cap \alpha(S_2) = \{\}) \wedge (\alpha_W(S_2) \cap \alpha(S_1) = \{\})$$

A superstep process is constructed using sequential programming operators and two asynchronous communication instructions *put* and *get*: $put(x, e)$ denotes a future assignment of the value of e to a non-local destination, x; $get(x, \text{“}e\text{”})$ denotes the assignment of the value resulting from the (non-local) future evaluation of e to the local variable, x. A crucial distinction is made between the nature of e in the two communication statements: e within $put(x, e)$ is the *value* of e whereas e within $get(x, \text{“}e\text{”})$ is a *syntatic expression*. To avoid confusion unevaluated expressions are enclosed within quotation marks.

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

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

where VAR denotes the space of variables, EXP denotes the space of expressions over variables and VAL denotes the space of values. The relation m is used to record all communications evoked by a superstep. Within the composition $S_1 \|^{ss} S_2$, the variable spaces of S_1 and S_2 are augmented by local message relations, $1.m$ and $2.m$ respectively. A message relation m is modified by the execution of *get* and *put* instructions as follows:

$$\begin{aligned} put(x, e) \quad &=_{df} \quad m := m \cup \{\text{“}x\text{”} \mapsto e\} \text{ with } \text{“}x\text{”} \neq \text{“}m\text{”} \\ get(x, \text{“}e\text{”}) \quad &=_{df} \quad m := m \cup \{\text{“}x\text{”} \mapsto \text{“}e\text{”}\} \text{ with } \text{“}x\text{”} \neq \text{“}m\text{”} \end{aligned}$$

Here the condition “x” $\neq$ “m” ensures a clean separation of auxiliary and programming variables. The effect of a communication is to update an auxiliary message buffer m in a way analogous to that used in the semantics of CSP [15] which associates communication with (auxiliary) traces. *put* and *get* instructions can be used to realise conditional communications. For example:

$$\begin{aligned} put(x, e) \ \lhd b \rhd \Pi \quad &=_{df} \quad m := ((m \cup \{\text{“}x\text{”} \mapsto e\}) \ \lhd b \rhd \ m) \\ get(x, \text{“}e\text{”}) \ \lhd b \rhd \Pi \quad &=_{df} \quad m := ((m \cup \{\text{“}x\text{”} \mapsto \text{“}e\text{”}\}) \ \lhd b \rhd m) \end{aligned}$$

where Π is the empty statement (SKIP) and the conditional statement $S_1 \lhd b \rhd S_2$ satisfies $S_1 \lhd true \rhd S_2 = S_1$ and $S_1 \lhd false \rhd S_2 = S_2$ – see [15] for further details. The communication laws allow several *d*ifferent messages to be sent to the same destination – for example, the relation

$$m = \{\text{“}x\text{”} \mapsto 1, \text{“}x\text{”} \mapsto 2\}$$

models a situation in which x is sent two inconsistent messages. In this case one of the messages is selected non-deterministically. A BSP message relation may contain unevaluated terms – for example, the relation

$$m = \{\text{“}x\text{”} \mapsto 1, \text{“}x\text{”} \mapsto \text{“}y\text{”}\}$$

models a situation in which x may be sent an unevaluated term “y”.

The messages contained in m are delivered synchronously through execution of a barrier operation. In practice it is convenient to specify explicitly the set of variables that may be updated by a barrier operation (see distributed barriers in §6). $barrier(m : \Gamma)$ denotes a synchronisation operation which “delivers”, under alphabet restriction Γ, the set of messages contained in m. Here Γ limits the variable update interface. More precisely,

$$m : \Gamma =_{def} \{d \mapsto r | d \in \Gamma \wedge (d \mapsto r) \in m\}$$

denotes the largest subset of m whose domain is contained within Γ.

The characteristic feature of a superstep is the merging of the results of individual process computations. Assume that processes S_1 and S_2 are such that all asynchronous communication statements (*puts* and *gets*) have been replaced by their definitions (updates of m). Then, letting $\alpha(S_1 \parallel^{ss} S_2) \subseteq \Gamma$

$$\begin{array}{lll} S_1 \parallel^{ss} S_2 & =_{\mathrm{d}f} & \mathrm{var} \quad 1.m, 2.m := \{\mapsto\}, \{\mapsto\} \ ; \\ & & \qquad ((S_1)^m_{1.m} \parallel^I (S_2)^m_{2.m}) \ ; \ barrier((\overline{1.m} \cup \overline{2.m}) : \Gamma) \\ & & \mathrm{end} \quad 1.m, 2.m \end{array}$$

Here, $S^m_{1.m}$ denotes S with all free occurrences of m replaced by $1.m$ (i.e. a local copy of m, $1.m$, is made within S_1), $\parallel^I$ denotes the independent parallel composition operator and $\overline{m}$ denotes an evaluation of m:

$$\overline{} : (VAR \mapsto (VAL \cup VAR)) \mapsto (VAR \mapsto VAL)$$

Map evaluation assigns a value to all named variables in the range of its input map argument:

$$dom(\overline{m}) = dom(m) \wedge \forall \text{“}z\text{”} \in dom(m).\overline{m}(\text{“}z\text{”}) = EVAL(m(\text{“}z\text{”}))$$

where *dom* returns the *proper* domain of its relation argument. For a variable y EVAL satisfies:

$$EVAL(y) = y \wedge EVAL(\text{“}y\text{”}) = y$$

Thus, message evaluation consists in selecting the variable names that occur in message contents and evaluating these names in the state immediately prior to the *barrier* operation. For a terminating superstep, m must be a *finite* relation; *barrier* may be defined over an explicit finite alphabet $\Gamma = \{x, \ldots, z\}$ using a multiple conditional assignment:

$$\begin{array}{lllll} barrier(m : \Gamma) & =_{\mathrm{d}f} & x, \ldots, z & := & \sqcap\overline{m}(\text{“}x\text{”}) \ \lhd \text{“}x\text{”} \in dom(m) \ \rhd x, \\ & & & & \vdots \\ & & & & \sqcap\overline{m}(\text{“}z\text{”}) \ \lhd \text{“}z\text{”} \in dom(m) \ \rhd z \end{array}$$

where $\{x, \ldots, z\} = \Gamma$ and $\sqcap$ denotes non-deterministic selection. The non-deterministic operator $\sqcap$ is not a function – however, in §3, problems associated with $\sqcap$ are resolved by initially extracting functions from the relation $\overline{m}$. As in [15] multiple assignments can be treated as explicit predicates [13]:

$$barrier(m:\Gamma) \quad = \quad \forall z \in \Gamma.\ \big((\text{“}z\text{”} \in dom(m) \Rightarrow z' = \sqcap\overline{m}(\text{“}z\text{”})) \ \wedge\ (\text{“}z\text{”} \notin dom(m) \Rightarrow z' = z) \big)$$

Here a variable z in the post-barrier state is denoted by z'.

3 Non-deterministic predicate substitution

An axiomatic definition of a superstep must incorporate a substitution mechanism which captures the effect of the delivery of a collection of asynchronous communications (denoted by m). One approach to determining whether or not the store associated with a variable name is to be updated using a communicated value is to test the variable name against the denotation of m (whose domain is the set of variable names to be updated). Such a test may be identified with the predicate $\lambda z.\text{“}z\text{”} \in dom(m)$.

More generally, let the *predicate substitution* ${}^{\lambda z.p(z)}Q_f$ – where z is a name which does not occur free in the expression Q, p is a predicate and f is a *finite* substitution function – denote the set of replacements

$$\{x \to f(x) | x \in \alpha(Q) \wedge p(x)\}$$

applied to Q. Here f is assumed to have type $VAR \mapsto VAL \cup EXP$ while p can be only satisfied for arguments of type VAR (i.e. only names can be updated). In order that the substitution be well defined it is necessary that $\{x | x \in \alpha(Q) \wedge p(x)\} \subseteq dom(f)$. For example, the predicate $\lambda z.false$ stipulates that no substitutions be made, the predicate $\lambda z.true$ specifies that every (unbound) name in an expression be replaced. In order to make precise this informal description, predicate substitution is defined over the structure of assertions. To reduce notational clutter ${}^{\lambda z.p(z)}P_f$ is abbreviated by ${}^{p}P_f$ in situations where there is no ambiguity with regard to the bound variable z.

$$\begin{array}{rcll}
{}^{p}(\bullet P)_f & = & \bullet({}^{p}P_f) & \\
{}^{p}(P \circ Q)_f & = & ({}^{p}P_f) \circ ({}^{p}Q_f) & \\
{}^{p}(Ux.P)_f & = & Uk.({}^{p\wedge z\neq k}(P^x_k)_f) & \text{where } k \text{ is a fresh name} \\
{}^{p}(x\ R\ y)_f & = & ({}^{p}x_f)R({}^{p}y_f) & \\
{}^{p}y_f & = & \left\{ \begin{array}{ll} f(y) & \text{if } p(y) \\ y & \text{otherwise} \end{array} \right. & \\
{}^{p}c_f & = & c &
\end{array} \qquad (1)$$

where x and y are arbitrary names, c is a constant, P and Q are predicates, $\bullet$ is a monadic operator (such as the logical operator, $\neg$), $\circ$ is a dyadic operator (such as the logical operator, $\wedge$), R is a relation (such as the arithmetic relation, $<$) and U is a universal quantifier (either $\forall$ or $\exists$).

Example
Consider the predicate substitution:

$${}^{\lambda z.z=x}(x > y)_{\{x\mapsto a, y\mapsto b\}} = (a > y)$$

Informally the substitution $f = \{x \mapsto a, y \mapsto b\}$ has to be applied to all variables in the expression $x > y$ which satisfy predicate $p(z) = \lambda z.z = x$. Variable x is rewritten since $p(x)$ holds but variable y remains unchanged since $\neg p(y)$. The overall effect of the substitution is to replace x with a within the expression $(x > a)$. Formally,

$$^{\lambda z.z=x}(x > y)_{\{x\mapsto a, y\mapsto b\}} \quad = \quad \left(\left(^{\lambda z.z=x}(x)_{\{x\mapsto a, y\mapsto b\}}\right) > \left(^{\lambda z.z=x}(y)_{\{x\mapsto a, y\mapsto b\}}\right)\right) \quad = \quad (a > y)$$

□

A predicate substitution degenerates to traditional *simultaneous* substitution if the set of variables associated with the replacement predicate p can be determined explicitly. Thus

$$\frac{\{z | z \in \alpha(Q) \wedge p(z)\} \quad = \quad \{n_1, \ldots, n_k\}}{^pQ_f \quad = \quad Q^{\,n_1 \quad ,\ldots, \quad n_k}_{f(n_1),\ldots,f(n_k)}} \tag{2}$$

The range of substitutions associated with a predicate may be extended as follows:

$$\frac{p \Rightarrow q}{^pQ_f \quad = \quad ^qQ_{f \lhd p \rhd id}}$$

where id denotes the identity function. Then, since $p \Rightarrow true$ for all predicates p,

$$^pQ_f \quad = \quad ^{true}Q_{f \lhd p \rhd id}$$

Thus, predicate substitution can be regarded as replacement of every free variable in the alphabet of Q - mirroring the predicate definition of *barrier* given in §2. However, to be consistent with conventional axiomatic semantics, a predicate substitution will normally be associated *only* with the the set of variables whose values are being *explicitly* updated.

The properties of predicate substitution developed so far relate to parallel term replacement. However, the definition of predicate substitution can be extended to encompass nondeterministic behaviour by extending the replacement mechanism from finite *functions* to finite *relations*.

$$^pQ_{f_1 \sqcap f_2} \quad =_{\mathrm{d}f} {}^pQ_{f_1} \wedge {}^pQ_{f_2} \tag{3}$$

Here, conjunction is used in a way akin to the weakest precondition semantics of nondeterministic selection [10]. Consider the problem of using a relation to define a substitution. For example, let $r = \{x \mapsto a, x \mapsto b\}$. Then $^{z=x}(x = x)_r \;=\; (a = a) \wedge (b = b)$. To exclude the possibility of applying an *inconsistent* set of substitutions for x, relations are decomposed into collections of their constituent functions. Substitution, using *any* of these functions, ensures that each occurrence of the *same* variable in a predicate is replaced by the *same* term. Predicate substitution over a replacement *relation* r involves making substitutions over *all* replacement functions that can be extracted from r. Let $f \preceq r$ mean that f is a *deterministic* set of messages *extracted* from r:

$$f \preceq r =_{\mathrm{d}f} f \subseteq r \wedge function(f) \wedge (dom(f) = dom(r)) \tag{4}$$

The first clause ensures that f and r are consistent, the second that f is deterministic and the third that both f and r have identical coverage. The relation $\preceq$ is reflexive only for functions.

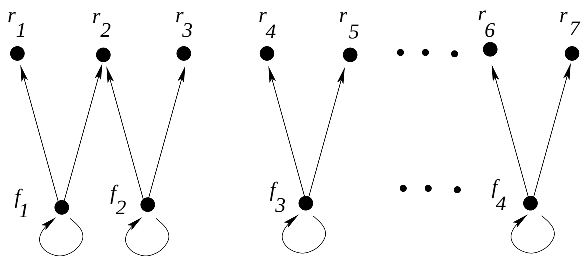

Figure 1: Sketch of the ordering of functions (f etc.) and non-functional relations (r etc.) under $\preceq$

In particular, $\{\mapsto\} \preceq \{\mapsto\}$. The ordering imposed on relations by $\preceq$ generates two layers of elements; the lower layer comprises functions while the upper layer comprises non-functional relations. For any relation r there is always a function f such that $f \preceq r$. $\preceq$ is anti-symmetric

$$f_1 \preceq f_2 \wedge f_2 \preceq f_1 \Rightarrow f_1 = f_2$$

and transitive (in the sense allowed by the two layer ordering)

$$f_1 \preceq f_2 \wedge f_2 \preceq r \Rightarrow f_1 \preceq r(\wedge f_1 = f_2)$$

A picture of the extraction ordering is given in Figure 1. Given $\preceq$, relational predicate substitution may be defined as follows:

$$^{p}Q_r \quad =_{\mathrm{d}f} \bigwedge_{f \preceq r} {}^{p}Q_f \tag{5}$$

Note that the properties of $\preceq$ guarantee that the set $\{f | f \preceq r\}$ is non-empty. Further, the conjunction $\bigwedge_{f \preceq r} {}^{p}Q_f$ is $finite$ if r is $finite$ (for example, if the cardinalities of the domain and range of r are n and m, respectively, then at most m^n functions can be extracted from r). Non-deterministic selection takes place over functions rather than over individual names. If the cardinality of $dom(r)$ is 1 then (5) degenerates to the weakest precondition semantics of non-deterministic selection [9, 10] since the set of functions satisfying $f \preceq r$ is the set of one-point functions each of which associates its single domain element with an element of $rng(r)$. Note that non-determinism is restricted to a computable (and, hence, monotonic) setting. Thus, the possible source of inconsistency in theories of non-deterministic functions described in [21] is excluded from the model above.

A predicate substitution is refined by reducing its degree of non-determinism:

$$^{p}Q_{r1} \sqsubseteq {}^{p}Q_{r2} \quad =_{\mathrm{d}f} \quad r2 \subseteq r1 \wedge dom(r1) = dom(r2)$$

This refinement is valid because $\{f | f \preceq r_2\} \subseteq \{f | f \preceq r_1\}$ and hence,

$$(\bigwedge_{f \preceq r1} {}^{p}Q_f) \quad \Rightarrow \quad (\bigwedge_{f \preceq r2} {}^{p}Q_f)$$

i.e. $^{p}Q_{r1} \Rightarrow {}^{p}Q_{r2}$ ensures that the refined transformation has a weaker precondition (larger domain) than its specification.

4 Axiomatic definition of *barrier*

In this section an axiomatic definition [14] of *barrier* is given in terms of a predicate substitution ([11, 24, 25, 26] include rules for reasoning about *put* and *get* operations). Consider a superstep S, with alphabet $\Gamma = \{x, \ldots, z\}$ and message relation m; operationally, $barrier(m : \Gamma)$ corresponds to a set of simultaneous assignments:

$$\begin{array}{llll} barrier(m:\Gamma) & =_{df} \quad x, \ldots, z & := & \sqcap \overline{m}(\text{“}x\text{”}) \ \lhd \text{“}x\text{”} \in dom(m) \ \rhd x, \\ & & & \vdots \\ & & & \sqcap \overline{m}(\text{“}z\text{”}) \ \lhd \text{“}z\text{”} \in dom(m) \ \rhd z \end{array}$$

Note that the set of names whose values are changed is not explicitly specified by the statement $barrier(m : \Gamma)$ – rather, it is extracted from the relation m. Informally, each variable in the domain of the communication m is assigned a corresponding, non-deterministically selected message. On the right hand side of the assignment the expression $\overline{m}(\text{“}z\text{”})$ is evaluated for each variable z in the domain of m. The meaning of *barrier* is then captured by the axiom:

$$\{^{\text{“}z\text{”} \in dom(m):\Gamma} Q_{\overline{m}}\} barrier(\overline{m} : \Gamma)\{Q\} \tag{6}$$

The precondition can be expanded, by (5), to:

$$\bigwedge_{f \preceq \overline{m}} {}^{\text{“}z\text{”} \in dom(m):\Gamma} Q_f$$

Thus, the axiom of barrier synchronisation permits an implementation to choose any consistent message function f, $f \preceq \overline{m}$, as its communication set. The effect of using $\overline{m}$ as the replacement relation is to ensure that any unevaluated references in m are evaluated in the state immediately preceding the *barrier* operation. Thus, a $get(x, \text{“}e\text{”})$ instruction which occurs in a superstep process is a request for a future evaluation of e; in the definition above e should be evaluated by the relevant superstep process immediately prior to termination (i.e. in the state immediately preceding *barrier*).

Example
An annotated superstep involving non-deterministic communication is given below:

$$\{true\} \ \big(\Pi \ \|^I \ \Pi \ \|^I \ get(z, \text{“}x\text{”}); get(z, \text{“}y\text{”})\ \big); barrier(\overline{m}) \ \ \{z = x \vee z = y\}$$

Here two passive superstep processes receive communication requests from the third process. That the program satisfies its pre-post-condition pair follows from:

$$\{true\} \ \big(\Pi \ \|^I \ \Pi \ \|^I \ get(z, \text{“}x\text{”}); get(z, \text{“}y\text{”})\ \big) \ \ \{\overline{m} = \{\text{“}z\text{”} \to x, \text{“}z\text{”} \to y\}\}$$

and

$$\Big\{ \bigwedge_{f \preceq \overline{m}} {}^{\text{“}z\text{”} \in dom(\overline{m})} (z = x \vee z = y)_f \Big\} \ barrier(\overline{m}) \ \ \{z = x \vee z = y\}$$

In particular,

$$\overline{m} = \{\text{“}z\text{”} \to x, \text{“}z\text{”} \to y\} \Rightarrow \bigwedge_{f \preceq \overline{m}} {}^{\text{“}z\text{”} \in dom(\overline{m})} (z = x \vee z = y)_f$$

The consequent expands to

$$\bigwedge_{f \preceq \overline{m}}^{\text{“}z\text{”} \in dom(\overline{m})} (z = x \vee z = y)_f \quad = \quad \left((z = x \vee z = y)_x^z \wedge (z = x \vee z = y)_y^z\right) \quad = \quad true$$

□

5 Program development: an example

The semantics of BSP-type programs may be based on predicate transformers. Supersteps comprise independent compositions and global *barrier*s: the Hoare-logic of the former is well known; the predicate transformer semantics of the latter has been given in §3 and §4. In this section it is demonstrated that correct parallel programs can be constructed in a way similar to that used in the development of provably correct sequential software. The example chosen for illustration is the construction of a program for Gaussian elimination [3, 30].

Consider the set of n linear equations $Ax = b$ where $A \in \Re^{n \times n}$ is a given non-singular matrix, $b \in \Re^n$ is a given vector and $x \in \Re^n$ is a vector of unknowns whose values are to be computed:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix}$$

The solution for x is given by:

$$x = A^{-1} b$$

However, the explicit calculation of A^{-1} is expensive so an alternative solution method is almost always used. Let $[Ab]$ be A augmented with b as its $(n+1)$st column. Gaussian elimination is used to transform A into an upper triangular form from which calculation of x may be achieved using the technique of backward substitution. Only the elimination phase, which has $n-1$ steps, is considered here. After the kth step, $1 \leq k < n$, columns $1 \ldots k$ of $[Ab]$ are in upper triangular form. Let $[Ab]^{(k)}$ denote $[Ab]$ after the kth step of the elimination process with $[Ab]^{(0)} = [Ab]$:

$$[Ab]^{(k)} = \begin{pmatrix} a_{1,1}^{(k)} & a_{1,2}^{(k)} & \cdots & a_{1,k+1}^{(k)} & \cdots & a_{1,n}^{(k)} & b_1^{(k)} \\ & a_{2,2}^{(k)} & \cdots & a_{2,k+1}^{(k)} & \cdots & a_{2,n}^{(k)} & b_2^{(k)} \\ & & \ddots & \vdots & & \cdots & \vdots \\ & 0 & & a_{k,k+1}^{(k)} & \cdots & a_{k,n}^{(k)} & b_k^{(k)} \\ & & & \vdots & & \cdots & \vdots \\ & & & a_{n,k+1}^{(k)} & \cdots & a_{n,n}^{(k)} & b_n^{(k)} \end{pmatrix}$$

It is usual to employ a pivoting strategy in the elimination process in order to ensure numerical accuracy. However, in order to highlight correctness issues, a Gaussian elimination algorithm *without* pivoting is considered here.

The steps of Gaussian elimination transform the system $[Ab]$ in such a way that (i) the solution vector x of $[Ab]$ is preserved and (ii) the (regular) sparsity pattern of $[Ab]$ is increased. Let

$$safety(k) =_{\mathrm{d}f} nonsingular(A^{(k)}) \wedge (A^{(k)})^{-1}b^{(k)} = (A^{(0)})^{-1}b^{(0)}$$

be a predicate which specifies that the solution of the transformed system of equations represented by $[Ab]^{(k)}$ is that of the solution of the original system $[Ab]^{(0)}$ (if $A^{(k)}$ is non-singular then $(A^{(k)})^{-1}$ is well-defined and unique) and

$$triangular(k) \;=_{\mathrm{d}f}\; \forall c : 1 \leq c < k : \forall r : c < r \leq n : a^{(k)}_{r,c} = 0$$

be a liveness predicate which specifies that the first $k-1$ columns of $[Ab]^{(k)}$ have the required triangular form. The kth phase of Gaussian elimination involves the subtraction of multiples of row k of $[Ab]^{(k-1)}$ from rows $k+1 \ldots n$:

$$\{1 \leq k < n\} \;\; row\_subtract \;\; \left\{\forall r : k < r \leq n : ab^{(k)}_{r,1\ldots n} = ab^{(k-1)}_{r,1\ldots n} - \left(ab^{(k-1)}_{k,1\ldots n} \times (ab^{(k-1)}_{r,k} / ab^{(k-1)}_{k,k})\right)\right\}$$

Here $ab^{(k)}_{i,j}$ is the (i,j)th element of $[Ab]^{(k)}$ It is known from linear algebra that the solution vector x and the non-singularity of A are preserved through (scaled) row subtraction. Thus, $safety(k)$ is an invariant of $row\_subtract$. The implementation of $row\_subtract$ can be facilitated by strengthening this precondition by asserting:

$$\{1 \leq k < n \wedge triangular(k)\} \;\; row\_subtract \;\; \{triangular(k+1)\}$$

After application of $row\_subtract$ the first k columns of $[Ab]^{(k)}$ have triangular form since:

$$triangular(k) \quad \wedge \quad a^{(k)}_{(k+1)\ldots n,k} = 0 \Rightarrow triangular(k+1)$$

Gaussian elimination can be implemented by the sequential computation:

$\{nonsingular(A)\}$
var k
    for $k := 1$ **to** $n-1$ **do** $row\_subtract$
end k
$\{safety(n) \wedge triangular(n)\}$

$I[1\ldots k) \;=\; safety(k) \wedge triangular(k)$ is a sufficiently strong invariant for the loop in the sequential program above. The adequacy of $I[1\ldots k)$ follows straightforwardly from the specification of $row\_subtract$.

A BSP version of Gaussian elimination can be devised in a way similar to that used in the derivation of the sequential version above. The essential difference is that the BSP program processes *distributed* information. For simplicity it is assumed that the computation is organised using $n+1$ processes with the ith process, $1 \leq i \leq n+1$, having write access to the ith column of $[Ab]$. In practice p processors (with $p << n$) and alternative data distributions are employed in implementations [1]. To perform row subtraction in the kth step, $1 \leq k < n$,

[1] Alternative data distributions (such as the 2D cyclic distribution) have a lower computational complexity in the BSP model than the column distribution used in the example above. See van de Vorst [30] and Bisseling [4] for further details.

process k computes and communicates the required row scalings:

$$\{1 \leq k < n\} \;\; find\_multipliers \;\; \left\{k.mult_{(k+1,\ldots,n)} = \frac{1}{a_{k,k}^{(k-1)}}\left(a_{k+1,k}^{(k-1)} \ldots a_{n,k}^{(k-1)}\right)\right\}$$

Here $i.v$ denotes the version of variable v in process i. Given that

$$\alpha_W(find\_multipliers) \;=\; \{k.mult\}$$

it follows that $safety(k)$ and $triangular(k)$ are invariants of $find\_multipliers$. The remaining processes of the parallel Gaussian elimination system are delayed until the multipliers are available. Thus, the multipliers are generated by the system:

$$\|_{1\leq i\leq n+1}^{I} \; (find\_multipliers \lhd (i = k) \rhd \Pi)$$

Local copies of the multipliers can be generated by superstep *communicate*:

$$\begin{gathered}\{\forall r : k < r \leq n : k.mult_r = a_{r,k}^{(k-1)}/a_{k,k}^{(k-1)}\} \\ communicate \\ \{\forall j : k < j \leq n+1 : j.mult_{(k+1,\ldots,n)} = k.mult_{(k+1,\ldots,n)}\}\end{gathered}$$

Superstep *communicate* may be implemented by letting processor k put copies of $k.mult_{(k+1,\ldots,n)}$ into the local memories of processors $k+1, \ldots n+1$ according to:

$$\|_{1\leq i\leq n+1}^{ss} \left(\textbf{for} \;\; j := k+1 \;\textbf{to}\;\; n+1 \;\textbf{do}\; put(j.mult_{(k+1,\ldots,n)}, k.mult_{(k+1,\ldots,n)}) \right) \;\; \lhd (i = k) \rhd \;\; \Pi$$

In the kth step only the kth buffer is non-empty:

$$k.\overline{m} = \{\text{“}(k+1).mult_{(k+1,\ldots,n)}\text{”} \to k.mult_{(k+1,\ldots,n)}, \ldots, \text{“}(n+1).mult_{(k+1,\ldots,n)}\text{”} \to k.mult_{(k+1,\ldots,n)}\}$$

The correctness of the implementation follows from rules (2) and (6). Again, $safety(k)$ and $triangular(k)$ are invariants of *communicate*. The computation $row\_subtract$ can be implemented in parallel as

$$\|_{k<i\leq n+1}^{I} \; local\_subtract_i$$

where each local calculation uses the set of multipliers $(k+1).mult, \ldots, (n+1).mult$:

$$\begin{gathered}\{(1 \leq k < n) \wedge \forall r : k < r \leq n : i.mult_r = a_{r,k}^{(k-1)}/a_{k,k}^{(k-1)}\} \\ local\_subtract_i \\ \{\forall r : k < r \leq n : a_{r,i}^{(k)} = ab_{r,i}^{(k-1)} - (ab_{k,i}^{(k-1)} \times i.mult_r)\}\end{gathered}$$

Finally, the following BSP-like program results.

$\{nonsingular(A)\}$
var k
for $k := 1$ **to** $n-1$ **do**
  $\|_{1\leq i\leq n+1}^{I} \; (find\_multipliers \lhd (i = k) \rhd \Pi)$;
  $\|_{1\leq i\leq n+1}^{I} \left(\textbf{for} \;\; j := k+1 \;\textbf{to}\;\; n+1 \;\textbf{do}\;\; put(j.mult_{(k+1,\ldots,n)}, k.mult_{(k+1,\ldots,n)}) \right) \;\; \lhd (i = k) \rhd \;\; \Pi$;
  $barrier(1.\overline{m} \cup \ldots (n+1).\overline{m})$;
  $\|_{1\leq i\leq n+1}^{I} \; local\_subtract_i \lhd (i > k) \rhd \Pi$
end k
$\{safety(n) \wedge triangular(n)\}$

The invariant for the loop is the same as for the sequential computation. The actions $find\_multipliers$ and $local\_subtract$ can be implemented as multiple assignments in a straightforward way. Finally, an explicit alphabet can be associated with the barrier operation – the communication variables are $\Gamma = \{1.mult, \ldots, (n+1).mult\}$. Then the synchronisation can be recast as:

$$barrier(1.\overline{m} \cup \ldots \cup (n+1).\overline{m} : \Gamma)$$

6 Efficiency: Partition Consistency

The (non-standard) BSP model is organised around supersteps (rather than processes which have to be composed). This means that (local) barriers must always match up thereby simplifying the underlying semantics. However, this simplification comes at the expense of efficiency. In this section the property *partition consistency* is introduced before readdressing the efficiency issue.

The superstep computational model may be regarded as a sequence of *fork and join* operations - one pair for each superstep. One way to implement a superstep is to distribute data over processes immediately prior to a computation (fork) and to gather the results from the processes at superstep completion (join). However, such data distribution is costly. Here we address the question of whether a sequence of supersteps can be efficiently implemented by *reusing* previous data partitions.

Consider the problem of implementing a sequential composition of supersteps:

$$(S_1 \parallel^{ss} S_2) \quad ; \quad (S_3 \parallel^{ss} S_4).$$

Ideally, this computation would be implemented by mapping S_1 and S_3 to one process (thread) with combined alphabet $\alpha(S_1) \cup \alpha(S_3)$ and S_2 and S_4 to another process (thread) with combined alphabet $\alpha(S_2) \cup \alpha(S_4)$. In other words an implementation can be found by transforming a sequential composition of parallel programs into a parallel composition of sequential programs [2]. Such a transformation is possible provided that composite threads have disjoint alphabets. Disjointness is guaranteed if the parallel programs in the sequential composition are *partition consistent*. A pair of compositions $(S_1 \parallel^{ss} S_2)$, $(S_3 \parallel^{ss} S_4)$ are partition consistent iff $disjoint(S_1, S_4)$ and $disjoint(S_2, S_3)$. By the definition of partition consistency and from the disjointness of superstep processes we have:

$$disjoint((S_1; S_3)\ ,\ (S_2; S_4))$$

This property guarantees that partition consistent supersteps can be converted into threads with disjoint alphabets. Partition consistency can be extended in obvious ways to include compositions having more than two components and sets (rather than pairs) of compositions.

Example
Consider the BSP Gaussian elimination program. Process i, $1 \leq i \leq (n+1)$ in each parallel composition has alphabet:

$$\alpha_W(S_i) = \{i.mult, [Ab](1 \ldots n, i)\} \ \wedge \ \alpha_R(S_i) = \{k\}$$

This partition is disjoint and so the implementation can be mapped into a composition of threads without the need for data repartitioning. □

7 Efficiency: Relaxing *barrier*

It is helpful, from a semantic viewpoint, to assume that all of the updates arising from a barrier synchronisation occur simultaneously. However, in a distributed environment the updates may be effected in different processes at different times. The realisation of synchronous algorithms on asynchronous architectures has been widely studied – it has been shown that a synchronous algorithm can be *correctly* realised on a distributed system employing *synchronisers* [1, 18]. In this section the distributed implementation of BSP supersteps is considered. A BSP-like global barrier may be implemented as a parallel independent composition of local store updates:

$$barrier(m : \Gamma_1 \cup \Gamma_2) = \quad barrier(\overline{m} : \Gamma_1) \parallel^I barrier(\overline{m} : \Gamma_2)$$

where $\Gamma_1 \cap \Gamma_2 = \{\}$.

Proof

$$\begin{array}{l}
barrier(m : \Gamma_1 \cup \Gamma_2) \\
= < \text{ treating barrier as an explicit predicate } > \\
\forall z \in (\Gamma_1 \cup \Gamma_2).((``z" \in dom(m) \Rightarrow z' = \sqcap\overline{m}(``z")) \ \wedge \ (``z" \notin dom(m) \Rightarrow z' = z)) \\
= < \text{ algebra, with } \Gamma_1 \cap \Gamma_2 = \{\} \ > \\
\forall z \in \Gamma_1.((``z" \in dom(m) \Rightarrow z' = \sqcap\overline{m}(``z")) \ \wedge \ (``z" \notin dom(m) \Rightarrow z' = z)) \wedge \\
\forall z \in \Gamma_2.((``z" \in dom(m) \Rightarrow z' = \sqcap\overline{m}(``z")) \ \wedge \ (``z" \notin dom(m) \Rightarrow z' = z)) \\
= < \text{ treating barrier as a predicate and using the definition of } \parallel^I \text{ [15] } > \\
barrier(\overline{m} : \Gamma_1) \parallel^I barrier(\overline{m} : \Gamma_2)
\end{array}$$

□

On the left hand side *barrier* is treated as a multiple update, in accordance with the definitions given in §2 and §4. On the right hand side it is necessary first to evaluate m in order to prevent erroneous interference between processes – the use of $\overline{m}$ ensures that the alphabets of the processes within the independent composition are disjoint. In order to facilitate efficient implementations (by avoiding message evaluation) it is henceforth assumed that supersteps employ only *put* communications with the result that $m = \overline{m}$.

Example
Consider the global synchronisation operation in the Gaussian elimination program:

$$barrier(1.m \cup \ldots \cup (n+1).m : \Gamma)$$

This barrier may be distributed as:

$$\parallel^I_{1 \le i \le n+1} barrier(1.m \cup \ldots \cup (n+1).m : \{i.mult\})$$

Each process can only update its local copy of *mult* through communication. □

BSPlib [12] implements global synchronisation on distributed message passing architectures in three phases [23]:

1. all of the processes exchange communication information (total exchange); this permits the number, sizes and destination addresses of all messages to be determined;

2. *gets* are translated into *puts*; and

3. after the total exchange is complete each process stores the number of its incoming messages. All superstep messages from S_i to $S_j, 1 \leq i, j \leq p$ are packaged into a bundle before transmission. A process is released from its local barrier after its required number of messages have been received.

Alternative implementations of BSP type barriers [5, 17] avoid the total exchange phase of BSPlib synchronisation – thereby, possibly, increasing efficiency [2]. For example, in [17] experimental results show that relaxing a BSP program for Fourier Transform in a way that avoids the total exchange operation results in a saving of 61.5% of the communication execution time on a 32 processor IBM SP2 machine. The PUB *oblivious* barrier operation $bsp\_oblsyn(\ldots, n)$ terminates after n incoming messages have been received. Here the number of incoming messages is given explicitly and need not be determined at run-time.

This idea can be extended so that a local barrier is associated with the set of superstep processes that communicate with it (i.e. its interface). Consider a system comprising processes $S_1, \ldots, S_p$. The local barrier operation within S_1 may be rewritten (in the absence of self-addressed messages) as:

$$barrier(\ (2.m \cup \ldots \cup p.m) : \alpha(S_1)\ )$$

In practice some of the message relations $2.m : \alpha(S_1), \ldots, p.m : \alpha(S_1)$ may be empty. The explicit removal of empty relations may promote the efficient implementation of the barrier operation in two respects:

- a local barrier operation over an empty relation corresponds to SKIP(Π) since $barrier(\{\}) = \Pi$ – thus, it is possible that a superstep process can bypass a synchronisation barrier;

- suppose that the local barrier in process S_1 above satisfies

$$3.m : \alpha(S_1) = \{\} \wedge \ldots \wedge p.m : \alpha(S_1) = \{\}$$

  In the absence of self-addressed messages the local barrier can be expressed as

$$barrier(2.m : \alpha(S_1))$$

  This operation can be implemented by synchronising S_1 *only* with S_2. Thus, in general, there is the possibility that interference between superstep processes can be restricted to a narrow interface. (Here it is assumed that all messages from S_2 to S_1 are packaged into a single bundle so that only one communication between the two processes is generated).

[2]Note that in certain situations – particularly when communication traffic is dense – efficiency gains may result from sending the *size* of expected messages around the communication network in an initial synchronisation phase: such information may allow optimisation through message combination and route selection. Other approaches to deriving efficient implementations of *barrier* include:

1. arranging the processes into subsets each of which may execute localised supersteps [7] (note, however, that this approach would make the reasoning framework more complex); and

2. two phase randomised broadcasting [4].

Barriers may be relaxed by the programmer during the process of program construction. However, it is necessary to ensure the consistency of relaxed local barriers and superstep communications. This can be achieved by adding a *definedness* condition, $\mathcal{D}$, to the axiomatic specification of *barrier*. $\mathcal{D}$ is conjoined to the barrier's normal precondition which guarantees consistency between messages generated in the superstep body and the relaxed barrier.

Consider a superstep $S_1 \parallel^{ss} \ldots \parallel^{ss} S_p$ with corresponding relaxed barrier $barrier(r_1) \parallel^{I} \ldots \parallel^{I} barrier(r_p)$. This relaxed barrier is well defined if:

$$\mathcal{D}\big(barrier(r_1) \parallel^{I} \ldots \parallel^{I} barrier(r_p)\big) \;\; =_{\mathrm{d}f} \;\; \big(\forall i, j : 1 \le i, j \le p : i.m : \alpha(S_j) \neq \{\} \Rightarrow \text{“}i.m : \alpha(S_j)\text{”} \subseteq r_j\big)$$

Here “$i.m : \alpha(S_j)$” is an interface name – in order to avoid notational clutter the use of quotation marks around interface names is left as understood rather than explicitly given.

Example

Consider the system $(S_1 \parallel^{ss} S_2 \parallel^{ss} S_3) = (put(x, y) \parallel^{ss} \Pi \parallel^{ss} \Pi)$ where $y \in \alpha(S_1)$ and $x \in \alpha(S_2)$. The barrier may be relaxed to:

$$\big(\Pi \parallel^{I} barrier(1.m : \alpha(S_2))\big) \parallel^{I} \Pi$$

The relaxed barrier operation is correct since the consistency requirement

$$\mathcal{D}\big(barrier(\{\}) \parallel^{I} barrier(1.m : \alpha(S_2)) \parallel^{I} barrier(\{\})\big)$$

in the context of the message space $1.m = \{\text{“}x\text{”} \mapsto y\}$, $2.m = \{\}$ and $3.m = \{\}$ simplifies to $1.m : \alpha(S_2) \subseteq 1.m : \alpha(S_2)$. An implementation of this system may safely avoid terminal synchronisation constraints on S_1 and S_3; process S_2 must wait to receive an asynchronous communication from S_1 before leaving the superstep. □

Consider a superstep with a single conditional communication

$$put(v, e) \lhd b \rhd \Pi$$

in process S_j, $1 \le j \le p$ with $v \in \alpha(S_i)$, $1 \le i \le p$. The local synchronisation barrier for S_i may be safely relaxed to

$$barrier(j.m : \alpha(S_i))$$

Execution of this relaxed barrier forces S_i to wait to receive a message from S_j. If, at this point, $\neg b$ holds this message will be vacuous. There is the potential to improve the efficiency of the implementation if the communication guard b can be evaluated in both S_i and S_j (i.e. the variables in b are read only). In this case the barrier of S_i may be further relaxed to:

$$barrier(j.m : \alpha(S_i) \lhd b \rhd \{\})$$

Here a local barrier is associated with a *conditional* communication relation – if guard $\neg b$ holds this relation is empty (and S_i may safely proceed); otherwise it is $j.m : \alpha(S_i)$.

Example

Consider $(S_1 \parallel^{ss} S_2 \parallel^{ss} S_3) = \big(put(x, 3) \lhd b \rhd \Pi \parallel^{ss} \Pi \parallel^{ss} put(y, 4)\big)$ where $\{x, y\} \subseteq \alpha(S_2)$ and b is a read only variable which may be accessed by S_1, S_2 and S_3. Here

$$(b \Rightarrow 1.m = \{x \mapsto 3\} \wedge \neg b \Rightarrow 1.m = \{\}) \;\wedge\; 2.m = \{\} \;\wedge\; 3.m = \{y \mapsto 4\}$$

The superstep barrier may be relaxed to:

$$(\Pi \quad \parallel^{I} barrier((1.m \cup 3.m) : \alpha(S_2)) \lhd b \rhd (3.m : \alpha(S_2)) \quad \parallel^{I} \Pi \quad)$$

The barrier definedness condition yields two cases:

1. b: $\mathcal{D}$ reduces to
   $1.m : \alpha(S_2) \subseteq (1.m \cup 3.m) : \alpha(S_2)) \wedge 3.m : \alpha(S_2) \subseteq (1.m \cup 3.m) : \alpha(S_2)$

2. $\neg b$: $\mathcal{D}$ reduces to $3.m : \alpha(S_2) \subseteq 3.m : \alpha(S_2)$.

□

Example
Consider the Gaussian elimination program. In the kth stage the communication buffer is:

$$k.m = \{j.mult_{(k+1,\ldots,n)} \mapsto k.mult_{(k+1,\ldots,n)} | k < j \leq (n+1)\} \ \wedge \ \big(\forall i : 1 \leq i \leq (n+1) : i \neq k \Rightarrow i.m = \{\}\big)$$

Using the definedness condition the local synchronisation barrier in process i, $1 \leq i \leq n+1$, can be safely relaxed to:

$$barrier\big((k.m : \{i.mult\} \lhd (i > k) \rhd \{\}\big)$$

Here the communication guard, $i > k$, involves i – the generic process label – and k – a read-only variable. □

8 Discussion

The central aim of the work reported here is to develop a *specialised* theory of BSP which closely resembles that of the (sequential) guarded command language [9]. By providing such a theory it is hoped that the development of correct BSP programs will be only marginally more difficult than the development of corresponding correct sequential programs. The construction of such a theory involves three changes to the conventional theory of sequential computation:

1. a generalised form of multiple assignment which involves a context dependent component is introduced;

2. a generalised form of non-determinism which involves the extraction of functions from relations is given; and

3. a form of data partitioning is used.

The close relationship between the axiomatisation of BSP given here and that of the guarded command language [9] supports an argument in favour of the thesis that BSP programming is much more straightforward than parallel programming in a reactive setting. An example of a BSP program proof of correctness has been shown to have a form similar to the corresponding sequential proof. Thus, it is hoped that the work reported here may be used to support the correct realisation of available *abstract* BSP algorithms as *concrete* programs. Alternatively, the rules given here could be recast so as to devise a *sequential* BSP design calculus – however, such a task is outside the scope of this paper.

Other semantics for BSP have been developed with different goals in mind. For example,

1. Jifeng, Miller and Chen give a *reactive* semantics for BSP based on the *general* model used in Unifying Theories of Programming [15]. This approach allows the relationship between BSP and other parallel models to be formally explored;

2. Yifeng Chen [6] has devised a design logic, LOGS (logic of global synchrony), for use in the formal development of BSP programs. The logic extends the usual pre-condition/postcondition approach by including additional intermediate assertions (for synchronisation points). LOGS is elegant and *compositional* but its relationship to sequential computation is not as close as is the model described in this paper; and

3. Loulergue, Hains, and Fosy have developed a functional model of BSP [16] with the aim of combining the speed gains of parallel computation with the benefits of algebraic reasoning. There are similarities between the treatment of superstep processes as *vectors* [16] and the axiomatisation given here. However, the approach advocated in this paper is purposely designed to facilitate the development of correctness proofs for real-world imperative programs. Note that in [16] two models of BSP are considered. The second model allows for subsets of processes to synchronise independently in a *general* way (which is suitable for the representation of divide and conquer algorithms). The generality of this model means that it is non-confluent In contrast the distribution rule for *barrier* given in §7 preserves equality as the rule has been specialised through semantic restrictions.

A second aim of this paper is to investigate a means of reconciling BSP correctness proofs with efficiency. In order to increase the potential for efficient implementations, *barrier* may be relaxed by decoupling. The BSP model supports one-sided communication statements whereby a single process specifies all of the communication parameters. Relaxation consists in the distribution of these parameters to all relevant local barriers. In an earlier paper [27] barrier relaxation was realised by transforming barriers into corresponding sets of MPI-like asynchronous two-sided communications. In particular, the transformed system generated a communication for each *put* or *get* statement that was executed in the original BSP program. The approach taken in this paper improves upon this previous work in two ways:

1. All messages between two superstep processes can be collected and sent in a single package (rather than as a sequence of messages). Experiments on BSP implementations [23] have shown that it is more efficient to delay all communication until the end of a superstep: communication routing bottlenecks are reduced by doing so.

2. The transformation approach in [27] is not general; it can be applied only in a number of it specialised situations. In contrast the axiomatic approach used to relax barriers in this paper can be applied to *all* superstep computations.

There is potential *automatically* to relax (perhaps, in a pre-compilation phase) BSP barriers. It is straightforward to devise a transformation scheme for generating barrier dependencies in some static situations. For example, a barrier can be relaxed easily, provided that all superstep communications are (i) unconditional and (ii) allow message destinations to be determined statically. Consider a superstep $S_1 \parallel^{ss} \ldots \parallel^{ss} S_p$. A local barrier operation

$$barrier((1.m \cup \ldots \cup p.m) : \alpha(S_i))$$

in process $S_i, 1 \leq i \leq p$, can be *relaxed* to:

$$barrier(\bigcup\{j.m : \alpha(S_i) | 1 \leq j \leq p \wedge \exists v, e.(S_j \ contains \ put(v, e)) \wedge v \in \alpha(S_i)\})$$

Here $S \ \ contains \ \ s$ determines if the body of S includes the instruction s; the dependency for the barrier of S_i is constructed by a simple join (union) of the names of processes sending messages to S_i. It is easy to construct a transformation scheme for generating relaxed barrier interfaces for such restricted forms of communications. Indeed, more elaborate transformation schemes can be developed for sub-classes of conditional and generic communications [27].

The situation is more complex when communicating to array variables. In general, the destination of the communication $put(a(e_1), e_2)$ cannot be determined statically. However, elementary static analysis is sufficient to allow relaxed barriers to be generated automatically for many types of *regular* communication patterns. For example, in the Gaussian elimination example the destinations of the communication

$$put(j.mult_{(k+1,\ldots,n)}, k.mult_{(k+1,\ldots,n)}) \lhd (i = k) \rhd \Pi)$$

can be determined in the context of the for loop

$$\textbf{for } j := k + 1 \textbf{ to } n + 1 \textbf{ do}$$

Many algorithms in numerical linear algebra are amenable to automatic relaxation using static analysis. For example, a parallel realisation of the Gram-Schmidt orthogonalisation algorithm would have the same structure as Gaussian elimination (without pivoting). Communication patterns within matrix multiplication algorithms can be determined statically. Although such algorithms account for a significant proportion of the basic operations of numerical linear algebra there remain a number which have communication patterns that are difficult to determine in this way. For example, Gaussian elimination is problematic to analyse statically if pivoting is included: in this case the pivot column is data dependent.

In order to avoid the limitations of static analysis an architecture independent relaxation scheme has been proposed which can be applied by programmers.This scheme offers the potential for efficient implementation but is achieved at the cost of additional correctness requirements. BSP with explicit barrier interface relaxation offers a clean model of parallel computation that freed from the obscurity introduced by low level communication detail (as found, for example, in MPI), permits correctness arguments to be conducted in a straightforward fashion and, in its relaxed form, offers the potential for efficient execution.

Acknowledgement
The authors are grateful to two anonymous referees for their constructive comments on an earlier draft of this paper.

References

[1] Awerbuch B.: Complexity of network synchronization, JACM, **32** (4) (1985), 804–823.

[2] Bougé L., Cachera D., Le Guyadec Y., Utard G., Virot B.: Formal validation of data-parallel programs: a two-component assertional proof system for a simple language, Theoretical Computer Science **189** (1997) 71–107.

[3] Back R.J.R., Sere K.: Stepwise refinement of parallel algorithms, Science of Computer Programming **13** (1), (1989), 133–180

[4] Bisseling R.H.: Basic techniques for numerical linear algebra on Bulk Synchronous Parallel Computers, Proceedings first Workshop on Numerical Analysis and Applications, LNCS **1196** (1997), Springer-Verlag, 46–57.

[5] Bonorden O., Juurlink B., von Otto I., Rieping I.: The Paderborn University BSP (PUB) Library—Design, Implementation and Performance 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing, (1999).

[6] Chen Y., Sanders J.W.: Logic of Global Synchrony, CONCUR 2001, (2001), 487–501.

[7] de la Torre P., Kruskal C.P.: Submachine Locality in the Bulk Synchronous Setting, Euro-Par'96, LNCS **1124** Vol. 2, (1996), Springer 352–358.

[8] de Roever W.P. et al: Concurrency Verification, Cambridge University Press, 2001.

[9] Dijkstra E. W.: A Discipline of Programming, Prentice-Hall, 1976.

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

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

[12] 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.

[13] Hehner E. C. R.: Predicative programming, Part 1, Communications ACM, **27** (1984) 134–143.

[14] Hoare C. A. R.: An axiomatic basis for computer programming, Communications ACM, **12** (10) (1969) 576–580.

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

[16] Loulergue F., Hains G., Fosy C., A calculus of BSP Programs, Science of Computer Programming, **37**(1-3), (2000) 253–277 .

[17] Kim J-S., Ha S., Jhon C. S., Relaxed barrier synchronization for the BSP model of computation on message-passing architectures, Information Processing Letters, **66**(5), (1998) 247–253.

[18] Lynch N.: Distributed Algorithms, Morgan Kaufmann, 1996.

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

[20] Misra J.: A Discipline of Multiprogramming, Springer, 2001.

[21] Morris J. M., Bunkenburg A., A source of inconsistency in theories of nondeterministic functions, Science of Computer Programming, **43** (2002) 77–89.

[22] MPI: A Message-Passing Interface Standard. University of Tennessee, 1995.

[23] Skillicorn D.B., Hill J.M.D., McColl W.F.: Questions and answers about BSP, Scientific Programming, **6**, (1997) 249–274.

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

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

[26] Stewart A., Clint M.: BSP-style computation: a semantic investigation, The Computer Journal, **44** (2001) 174–185.

[27] Stewart A., Clint M., Gabarró J., Serna M.J.: Towards formally refining BSP barriers into explicit two-sided communications, in Euro-Par 2001: Parallel Processing, LNCS **2150**, Springer-Verlag, eds: R. Sakellariou, J. Keane, J. Gurd, L. Freeman(2001) 549–559.

[28] Stewart A., Clint M., Gabarró J.: Algebraic rules for reasoning about BSP programs, in: Constructive Methods for Parallel Programming, eds: S. Gorlatch and C Lengauer, Nova Science, (2002) 41–57.

[29] Valiant L. G.: A bridging model for parallel computation, Comm. ACM, **33** (8) (1990) 103–111.

[30] van de Vorst J.G. G.: The formal development of a parallel program performing LU-decomposition, Acta Informatica, **26** (1988) 1–17.