A programming model for BSP with partitioned synchronisation

Alan Stewart

▶ **To cite this version:**

Alan Stewart. A programming model for BSP with partitioned synchronisation. Formal Aspects of Computing, 2010, 23 (4), pp.421-432. ⟨10.1007/s00165-010-0163-2⟩. ⟨hal-00624431⟩

HAL Id: hal-00624431

https://hal.science/hal-00624431v1

Submitted on 17 Sep 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Under consideration for publication in Formal Aspects of Computing

A programming model for BSP with partitioned synchronisation

Alan Stewart

School of Computer Science, The Queen's University of Belfast,
Belfast BT7 1NN, Northern Ireland.
Email: a.stewart@qub.ac.uk

Abstract. A BSP superstep is a *distributed* computation comprising a number of simultaneously executing processes which may generate asynchronous messages. A superstep terminates with a barrier which enforces a global synchronisation and delivers all ongoing communications. Multilevel supersteps can utilise barriers in which subsets of processes, interacting through *shared* memories, are locally synchronised (partitioned synchronisation). In this paper a state-based semantics, closely related to the classical sequential programming model, is derived for distributed BSP with partitioned synchronisation.

Keywords. BSP; state-based reasoning; partitioned synchronisation; UTP; parallel by merge; refinement; weakest preconditions.

1. Introduction

One inherent difficulty in the development of scientific software is the reconciliation of the requirements that code be both *correct* and *efficient*. Placing too much emphasis on *correctness* may result in an abstract, but inefficient, programming model. Many members of the scientific programming community have unjustly rejected functional programming languages because of perceived performance inadequacies. Alternatively, striving for optimal efficiency can run the risk of comprising software correctness and can result in the employment of architecture-specific programming models. In this paper it is proposed that the BSP [Bis04, McC95, Val90, Val08] model of distributed computation combines the benefits of a well-established cost calculus (for evaluating program efficiency) with the advantages of a simple programming framework closely related to sequential program refinement.

A BSP computation comprises a sequence of *supersteps*. A superstep contains a set of *independent* processes, each with local memory, which perform computations and generate asynchronous communications. A superstep terminates with a global barrier which synchronises processes and delivers ongoing communications. The cost of a superstep S with p processors can be calculated using the following assumptions:

- a basic arithmetic operation or local memory access has unit cost;
- the maximum number of local operations performed by a process is w;
- the maximum number of I/O operations performed by a process is h;
- the ratio of communication cost to the unit cost is g and

Correspondence and offprint requests to: Alan Stewart

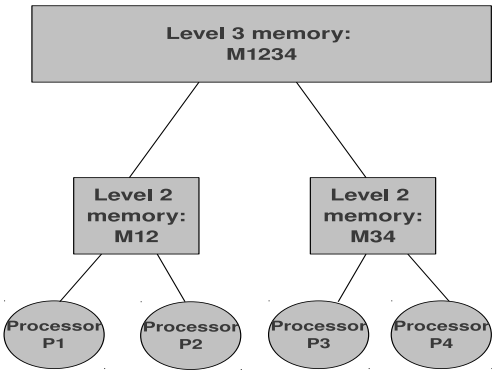

Fig. 1. A multilevel BSP architecture

- the cost of synchronising all of the processes within S is L.

The total cost of superstep S is defined to be $w + h \times g + L$. A wide range of parallel algorithms have been developed and analysed using the BSP cost calculus.

A variant shared memory version of BSP, BSPRAM, has been proposed by Tiskin [Tisk98]. Again superstep processes are independent with being used to synchronise processes and update shared memory. BSPRAM has been refined to a multilevel variant [Val08] by partitioning shared memory into a hierarchy. Memory at the top of the hierarchy is assumed to have the highest access cost while cache memories at the process level have the lowest access costs. Figure 1 shows a memory configuration for four processes, $P_1, \ldots, P_4$: processes P_1 and P_2 have access to memory M_{12} while processes P_3 and P_4 have access to memory M_{34}. All processes can (indirectly) access the memory M_{1234}. Processes can be synchronised in various ways: for example,

- processes P_1 and P_2 and processes P_3 and P_4 can be synchronised independently (partitioned synchronisation); P_1 and P_2 interact through M_{12} while P_3 and P_4 interact through M_{34}. The notion of having sub-machine synchronisations in BSP was first proposed by de la Torre and Kruskal [TK96].
- all processes can be synchronised with global interactions occurring via the shared memory M_{1234}.

Multilevel BSP can be used to analyse the computational cost of BSP algorithms on multicore architectures. For example, the architecture in Figure 1 has an associated 3-level cost model [Val08]. Each level is associated with four cost parameters: (p_i, g_i, L_i, m_i) where p_i is the number of level $i-1$ components inside a level i component, g_i is the communication bandwidth (relative to the basic computational cost) from level i to level $i+1$, L_i is the cost of synchronising all of a level i's sub-components and m_i is the size of memory available at level i. More realistically, consider a cost model for an architecture containing 4 identical multicores, each having 8 cores[1]. Each core has a cache memory and associated cost parameters:

$(p_1 = 1, g_1 = 1, L_1 = 3, m_1 = 8kB)$

Here L_1 refers to the cost of synchronising internal core threads. Each multicore has memory accessible by all its cores – L_2 is the cost of internal core synchronisation at the chip level while g_2 is the cost for a multicore to access level 3 memory:

$(p_2 = 8, g_2 = 3, L_2 = 23, m_2 = 3MB)$

[1] The example given here is based on the cost model for Sun Niagara UltraSparc T1 multicore chips given in [Val08].

The overall architecture contains a large globally accessible memory:

$(p_3 = 4, g_3 = \infty, L_3 = 108, m_3 \leq 128GB)$

This example illustrates how partitioned synchronisation has the potential to be much less computationally expensive than global synchronisation (see the L_2 and L_3 costs above). Architectures comprising a number of multicores may be assembled either by building an interconnect network linking the multicores or by adding an overarching memory hierarchy (as described above). The intent of this paper is to model different forms of synchronisation patterns (rather than the details of memory hierarchies). In particular, localised barriers are distinguished from global synchronisations.

A state- based semantics [Bk09, BvW98, Bh03, Dij68, Dij76] for distributed BSP with partitioned synchronisation is derived. The intent is twofold:

1. to devise a BSP program development framework akin to the classical sequential refinement approach of Dijkstra. To this end a superstep is treated as a state transformer (in much the same way that assignments are treated in Dijkstra's guarded command language [Dij76]); and
2. to provide an abstraction (partitioned synchronisation) which distinguishes different kinds of barrier.

Alternative BSP semantics have been developed with different goals in mind:

1. Jifeng, Miller and Chen [HMC96] give a *reactive* trace-based semantics for BSP;
2. Yifeng Chen [CS04, CS03] has devised a design logic, LOGS (logic of global synchrony), for use in the formal development of BSP programs. LOGS uses a process-based approach to program design; and
3. Loulergue, Hains, and Fosy have developed a functional model of BSP [LHF00] with the aim of combining the speed gains of parallel computation with the benefits of algebraic reasoning. The approach has been developed to provide a BSP skeleton framework [JL09].

In §2 a brief overview of some of the pertinent features of the UTP [HH98] framework for program construction is presented. A distributed stated-based superstep semantics is devised in §3. Superstep laws and termination properties are listed in §4 while a weakest precondition semantics for barrier synchronisation is defined in §5. Under certain conditions a superstep can be decoupled into less computationally expensive, localised, operations [TK96, Val08]. In §6 a variant form of barrier, *partitioned synchronisation*, is defined. In §7 a superstep program environment is described and in §8 a specification of *PrefixSum* is refined into a BSP system which utilises a number of partitioned synchronisations. Finally, in §9 it is shown how to refine superstep systems into compositions of interfering processes.

2. Preliminaries: Unified Theories of Programming (UTP)

UTP [HH98] is a framework for refining predicates into programs. Here a brief overview of those features of UTP that are relevant to the development of a multilevel BSP programming model is given. A predicate P is defined over a finite set of (free) variables αP where

$\alpha P = in\alpha P \cup out\alpha P$

and where
$in\alpha P$ is a set of undashed variables (initial values) and
$out\alpha P$ is a set of dashed variables (final values).
The predicate $x' = a + x \wedge a' = a$ with alphabet $\{x, a, x', a'\}$ corresponds to the program $x := a + x$. The operation SKIP over an alphabet $A = \{x, x'\}$ is defined as

$I\!I_A =_{df} x' = x$

Predicates may be composed using the operators $\lhd\ \rhd$ (selection), $\parallel$ (independent parallel composition), $\sqcap$ (non-deterministic choice) and ; (sequential composition):

$$\begin{array}{rll} P \lhd b \rhd Q =_{df} (P \wedge b) \vee (Q \wedge \neg b) & \text{where } \alpha b \subseteq \alpha P = \alpha Q \\ P \parallel Q =_{df} P \wedge Q & \text{where } out\alpha P \cap out\alpha Q = \{\} \\ P \sqcap Q =_{df} P \vee Q & \text{where } \alpha P = \alpha Q \\ P(m, m'); Q(m, m') =_{df} \exists v.P(m, v) \wedge Q(v, m') & \text{where } out\alpha P = \{m'\} \wedge in\alpha Q = \{m\} \end{array}$$

An additional operation (separating simulation) is used to rename an output variable of a predicate (and adjust the alphabet of the predicate accordingly). For example, the simulation U_0 renames m' as $0.m'$:

$$U_0(m) =_{df} \exists 0.m.\exists m'.0.m' = m$$

$$\alpha U_0(m) =_{df} \{m, 0.m'\}$$

A simulation $U_1(m)$ which renames m' as $1.m'$ can be defined in a similar way. Finally, the alphabet of a predicate R can be extended to include the variables x and x' as follows: $R_{+x} =_{df} R \wedge x' = x$. For example suppose that a predicate P has variable m in its alphabet. The predicate $P; U_0(m)_{+out\alpha P\setminus\{m\}}$ has alphabet $(\alpha P\setminus\{m'\})\cup\{0.m'\}$ and is such that the output m' of P is equivalent to output $0.m'$ of $P; U_0(m)_{+out\alpha P\setminus\{m\}}$.

3. Barrier Synchronisation

A superstep contains a set of *independent* processes which perform computations and generate asynchronous communications. A superstep terminates with a global barrier which synchronises processes and delivers ongoing communications. Let variable m denote a set of ongoing asynchronous communications that have been generated but have not, as yet, been delivered. A set of communications is modelled as a *relationship* between destinations (i.e. the variables to be updated) and values - see [HMC96, SC01, SCG04]:

$$m : VAR \mapsto VAL$$

Here VAR is a set of variable names and VAL is a set of values.

Superstep $SS(P, Q)$ comprises process bodies P and Q; $SS(P, Q)$ is well-defined if the variables occurring in P and the variables occurring in Q have, at most, the name m in common:

$$\alpha P \cap \alpha Q \subseteq \{m, m'\}$$

For example, if $\alpha P = \{x, m, x', m'\}$ and $\alpha Q = \{y, m, y', m'\}$ then $SS(P, Q)$ is a valid superstep. The definedness condition above permits superstep processes to be executed independently (see below).

A superstep process is a sequence of conventional programming instructions and asynchronous communication operations. Execution of the asynchronous communication $put(x, e)$ evaluates the local expression e and sends the resulting value, asynchronously, to the "foreign" variable x; semantically $put(x, e)$ is defined to update the communication space m as follows:

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

An asynchronous send $put(x, e)$ in process P is well-defined if $\alpha e \subseteq \alpha P$ where αe denotes the set of variables that occur in expression e. In a superstep different processes may generate and send inconsistent messages to the same destination variable. For example, P may contain the statement $put(x, 1)$ while Q contains $put(x, 2)$. In this case the combined set of ongoing communications has the form

$$m = \{\ldots, x \mapsto 1, x \mapsto 2, \ldots\}$$

$SS(P, Q)$ is defined in such a way that component processes utilise disjoint output communication spaces. The separate communication spaces are merged (§7.2 [HH98]) and the resulting messages are delivered in a terminating barrier, $SYNC$. Let

$$P0 = \exists m.m' = \{\}; P; U_0(m)_{+\alpha P\setminus m}$$
$$Q1 = \exists m.m' = \{\}; Q; U_1(m)_{+\alpha Q\setminus m}$$

be *independent* processes generated by renaming the output variable m' in both P and Q. Note that $P0$ and $Q1$ are initialised using empty communication relations. Superstep $SS(P, Q)$ first executes $P0$ and $Q1$, thereby performing local computations and generating asynchronous communications. Subsequently the asynchronous messages are delivered in parallel. Let $A = \alpha P\setminus\{m, m'\}$ and $B = \alpha Q\setminus\{m, m'\}$: Then:

$$SS(P, Q) =_{df} \exists 0.m', 1.m'.(P0||Q1);\ SYNC((A \cup B) \lhd (0.m \cup 1.m))_{A\cup B}$$

Here $\gamma \lhd m$ denotes the restriction of the domain of map m to the set γ:

$$\gamma \lhd m =_{df} \{x \mapsto y | (x \mapsto y) \in m \wedge x \in \gamma\}$$

The operation $SYNC(m)_\gamma$ is a variant form of parallel assignment which models the behaviour of a barrier synchronisation for a communication map m and an alphabet γ where $dom(m) \subseteq \gamma$ and where dom is a function which returns the domain of its argument.

$$\alpha SYNC(m)_\gamma = \gamma$$

If f is a function then $SYNC(f)_\gamma$ is a deterministic set of parallel assignments:

$$\begin{aligned} SYNC(f)_\gamma &= \forall x \in \gamma . x' = f(x) \lhd x \in dom(f) \rhd x' = x \\ &= ||_{x \in dom(f)}\ x := f(x) \end{aligned} \tag{1}$$

Here x is a meta-variable (rather than a conventional programming variable). It follows that

$$SYNC(\{\})_\gamma = II_\gamma$$

If the map m is not a function then $SYNC(m)_\gamma$ is defined non-deterministically:

$$SYNC(m)_\gamma =_{df} \sqcap_{f \preceq m} SYNC(f)_\gamma \tag{2}$$

Here $f \preceq m$ denotes the set of *functions* f which cover m and are consistent with it:

$$f \preceq m =_{df} f \subseteq m \wedge function(f) \wedge dom(f) = dom(m)$$

For example, if $m = \{x \mapsto 1, x \mapsto 2, y \mapsto 1, y \mapsto 2\}$ four functions can be extracted from m using $\preceq$: $f_1 = \{x \mapsto 1, y \mapsto 1\}$, $f_2 = \{x \mapsto 2, y \mapsto 1\}$, $f_3 = \{x \mapsto 1, y \mapsto 2\}$ and $f_4 = \{x \mapsto 2, y \mapsto 2\}$. The set $\{f | f \preceq m\}$ is guaranteed to be non-empty. The degree of non-determinism of $SYNC(m)_\gamma$ is bound for *finite* m (if the cardinalities of the domain and range of m are a and b, respectively, then at most b^a functions can be extracted from m).

Synchronisation is destructive in that destination variables are overwritten; an alternative form of barrier $SYNC^+(m)_\gamma$ updates destination variables by adding message content to destination content. For functional f:

$$SYNC^+(f)_\gamma = \forall x \in \gamma . x' = x + f(x) \lhd x \in dom(f) \rhd x' = x$$

For non-functional m,

$$SYNC^+(m)_\gamma = \sqcap_{f \preceq m} SYNC^+(f)_\gamma$$

A superstep with barrier $SYNC^+$ is denoted $SS^+(P, Q)$ (see §8).

4. Superstep laws and termination

Supersteps inherit many of the properties of parallel composition:

$$\begin{aligned} SS(P, Q) &= SS(Q, P) \\ SS(II_A, II_B) &= II_{A \cup B} \\ SS(P \lhd b \rhd Q, R) &= SS(P, R) \lhd b \rhd SS(Q, R) \\ SS(P \sqcap Q, R) &= SS(P, R) \sqcap SS(Q, R) \end{aligned}$$

If $\alpha P \subseteq \alpha P' \backslash \{m, m'\}$ and $\alpha Q \subseteq \alpha Q' \backslash \{m, m'\}$ then $(P \parallel Q); SS(P', Q') = SS(P; P', Q; Q')$.

The semantics of supersteps can be extended to reason about termination. In UTP a design $P \vdash Q$ is such that if assumption P holds then it is guaranteed that Q terminates in a state specified by Q [HH98].

$$P \vdash Q =_{df} (ok \wedge P) \Rightarrow (ok' \wedge Q)$$

Here the variables ok and ok' are used to record whether a program has been started and whether it has terminated, respectively. Predicate $true$ denotes arbitrary behaviour (including possible non-termination) and can be represented by either $false \vdash false$ or $false \vdash true$. The termination properties of supersteps

are closely related to those of independent parallel composition. If a superstep process diverges then so too does the superstep:

$$SS(true, Q) = true$$

If the component processes of a superstep terminate then so too does the superstep:

$$\frac{P_0 \vdash P \quad , \quad Q_0 \vdash Q}{P_0 \wedge Q_0 \vdash SS(P, Q)}$$

Terminating superstep processes can only generate *finite* communication spaces.

5. A Weakest Precondition semantics for $SYNC$

A weakest precondition semantics of $SYNC(m)_\gamma$ is derived below. Let P be a predicate and f be a *finite* function with $dom(f) \in \alpha P$. P^f denotes P modified by the application of the *simultaneous* substitutions $\{x \to f(x) | x \in dom(f)\}$. Let x and y be variables, c a constant, P and Q predicates, R a relation and k a *fresh* variable name: P^f is defined over the structure of P as follows:

$$\begin{array}{ll} (\neg P)^f = \neg(P^f) & (x\ R\ y)^f = x^f\ R\ y^f \\ (P \wedge Q)^f = P^f \wedge Q^f & (P \vee Q)^f = P^f \vee Q^f \\ (\forall x.P)^f = \forall k.(P^x_k)^f & (\exists x.P)^f = \exists k.(P^x_k)^f \\ y^f = f(y) \lhd y \in dom(f) \rhd y & c^f = c \end{array} \tag{3}$$

Here $e_1 \lhd b \rhd e_2$ is e_1 if b is true and e_2 otherwise. It follows that $P^{\{\}} = P$ and $P^{\{x \mapsto v\}} = P^x_v$ (conventional substitution). The weakest precondition of a deterministic barrier $SYNC(f)_\gamma$ is:

$$SYNC(f)_\gamma\ \mathbf{WP}\ P \ = \ ||_{x \in dom(f) \cap \gamma}\ x := f(x)\ \mathbf{WP}\ P \ = \ P^{\gamma \lhd f}$$

A substitution over a finite mapping m is defined as:

$$P^m =_{df} \bigwedge_{f \preceq m} P^f \tag{4}$$

If $dom(m) = \{x\}$ then $\bigwedge_{f \preceq m} P^f$ degenerates to Dijkstra's semantics of non-determinism [Dij76] $\bigwedge_{v \in rng(m)} P^x_v$. Thus, P^m generalises the $\mathbf{WP}$ semantics of non-deterministic choice ($\sqcap$):

$$(P \sqcap Q)\ \mathbf{WP}\ R = (P\ \mathbf{WP}\ R) \wedge (Q\ \mathbf{WP}\ R)$$

P^m is used to define the $\mathbf{WP}$ semantics of non-deterministic $SYNC$:

$$\begin{aligned} SYNC(m)_\gamma\ \mathbf{WP}\ P &= \bigsqcap_{f \preceq (\gamma \lhd m)} SYNC(f)_\gamma\ \mathbf{WP}\ P \\ &= \bigwedge_{f \preceq (\gamma \lhd m)} (SYNC(f)_\gamma\ \mathbf{WP}\ P) \ = \ \bigwedge_{f \preceq (\gamma \lhd m)} P^f \ = \ P^{\gamma \lhd m} \end{aligned}$$

A weakest precondition of $SYNC^+$ can be devised by modifying the substitution rule P^f to P^{+f}:

$$y^{+f} = y + f(y) \lhd y \in dom(f) \rhd y$$

6. Partitioned synchronisation

Global synchronisation is often considered to be prohibitively expensive to implement. In certain situations superstep barriers can be implemented using localised forms of synchronisation. Let $SS(\{P_i | i \in I\})$ be a superstep comprising a set of processes indexed by a set I and let PS be a partition of I:

$$partition(I, PS) =_{df} (\bigcup_{J \in PS} J) = I \wedge \forall J, K \in PS.\ J \cap K \neq \{\} \Rightarrow J = K$$

Superstep processes can always be decoupled since they are independent:

$$||_{i\in I} P_i = ||_{J\in PS}\ (||_{j\in J} P_j)$$

A termination barrier can be decoupled using partition PS if the set of messages generated within each partitioned set of processes is destined for delivery within the same process set:

$$\frac{\mathit{partition}(I, PS) \wedge \forall J \in PS.\ \bigcup_{j\in J}\ \mathit{dom}(j.m) \subseteq \bigcup_{j\in J}\ \alpha P_j}{SYNC(\bigcup_{i\in I} i.m)\ \ =\ \ ||_{J\in PS} SYNC(\bigcup_{j\in J} i.m)} \tag{5}$$

In such circumstances an implementation need only synchronise sets of processes within the same partition. The smallest partition $\mathit{partition}(I, \{I\})$ contains one index set and corresponds to complete inter-dependence. The largest partition, $\mathit{partition}(I, \{\{i\}|i \in I\})$ corresponds to a completely decoupled system.

Consider a superstep $SS(\{P_i|i \in I_1 \cup I_2\})$ where $I_1 = \{2, 4, \ldots, 2^n\}$ and $I_2 = \{1, 3, \ldots, 2^n - 1\}$. If even(odd) indexed processes communicate only with other even(odd) indexed processes then

$$SS(\{P_i|i \in I_1 \cup I_2\})\ =\ SS(\{P_i|i \in I_1\})||SS(\{P_i|i \in I_2\})$$

If the process sets $\{P_i|i \in I_1\}$ and $\{P_i|i \in I_2\}$ are assigned for execution to sets of cores which either have a shared memory or a fast localised interconnect network then $SS(\{P_i|i \in I_1 \cup I_2\})$ can be implemented in a way that exploits the computing potential of the hardware. Such a partitioned synchronisation is likely to execute faster than a conventional BSP barrier which enforces a global synchronisation. Partitioned synchronisation can be characterised by the following algebraic law:

$$\frac{A \cap B = \{\} \wedge \mathit{dom}(m) \subseteq A \wedge \mathit{dom}(n) \subseteq B}{SYNC(m \cup n)_{A\cup B}\ =\ SYNC(m)_A \parallel SYNC(n)_B} \tag{6}$$

7. Superstep systems

Supersteps may be embedded within a sequential programming framework; in this way BSP systems can be developed using conventional "sequential refinements" while the inherent parallelism of superstep programs can be used to exploit the computational potential of multiprocessor architectures (see §9).

The embedding of supersteps within a sequential programming environment requires a generalisation of the definition of superstep. Consider the repetitive $(*)$ construct below:

$$\textbf{var}\ i = 0;\ \ (i \neq n) * (SS(P, Q);\ i := i + 1)\ \ \textbf{end}\ i$$

Here superstep processes P and Q share a "global" variable, i; in order to maintain superstep process independence i must be read-only for both P and Q (although i can be updated at a barrier). Let $\alpha_W P$ denote those variables which can be updated by P. For example, $\alpha_W\ \mathit{put}(i, e) = \{m\}$ and $\alpha\ \mathit{put}(i, e) = \{i, m\} \cup \alpha e$. Processes P and Q are independent if

$$\alpha_W P \cap \alpha Q \subseteq \{m\} \wedge \alpha_W Q \cap \alpha P \subseteq \{m\}$$

This modified definition of independence allows P and Q to share *read-only* variables.

All supersteps in a system must have a fixed degree of parallelism, say k. The ith superstep process P_i, $1 \leq i \leq k$, has write access to a set of local variables, D_i, and read-only access to a set of "global" variables, D_0 where

$$\forall i, j.\ 1 \leq i < j \leq k \Rightarrow D_i \cap D_j = \{\} \wedge D_i \cap D_0 = \{\}$$

(i.e. $\alpha_W P_i \subseteq D_i$ and $\alpha P \subseteq D_0 \cup D_i$, $1 \leq i \leq k$). The structure of a superstep computation is outlined in Figure 2. The definition of a *partitioned* superstep is modified to ensure that "global" variables are not updated at barriers:

$$\frac{\mathit{partition}(I, PS) \wedge \forall J \in PS.\ \bigcup_{j\in J}\ \mathit{dom}(j.m) \subseteq \bigcup_{j\in J}\ D_j}{SYNC(\bigcup_{i\in I} i.m)\ \ =\ \ ||_{J\in PS} SYNC(\bigcup_{j\in J} i.m)} \tag{7}$$

$$C^k \in \mathbf{Comp} \quad ::= \quad SS(P_1, \ldots P_k) \;\mid\; \|_{J \in PS}\, SS(\{P_i | i \in J\}) \;\mid\; x := e \;\mid\; C_1^k ; C_2^k \;\mid\; C_1^k \lhd b \rhd C_2^k \;\mid\; b * C^k$$
$$\text{where} \quad P_1, \ldots P_k \in \mathbf{Process}, \quad \alpha P_1 \subseteq D_1 \cup D_0, \ldots \alpha P_k \subseteq D_k \cup D_0$$
$$b \in \mathbf{BooleanExp}, \alpha b \subseteq D_0, x \in D_0, \alpha e \subseteq D_0$$

Fig. 2. Structure of superstep systems.

8. The derivation of parallel prefix sum

The superstep framework above is used to derive a parallel program from a specification. The resulting program contains barriers with varying degrees of synchronisation. The derivation starts from a predicate specification and a number of refinement steps are carried out [Bk09, Dij68] until a superstep program is derived. The program development is heavily influenced by R. Back's approach to constructing programs from invariants. Given an integer l, $1 \leq l$, and an array A with integer elements $\{a_i | 1 \leq i \leq 2^l\}$ parallel prefix sum [Akl89] is specified as

$$\mathit{PrefixSum}(l, A) =_{df} \forall i.1 \leq i \leq 2^l . a_i' = \sum_{1 \leq k \leq i} a_k$$

Here a_i and a_i' denote the initial and final states of array element a_i, respectively. For example:

$$\mathit{PrefixSum}(2, A) \;=\; a_1' = a_1 \;\wedge\; a_2' = a_1 + a_2 \;\wedge\; a_3' = a_1 + a_2 + a_3 \;\wedge\; a_4' = a_1 + a_2 + a_3 + a_4$$

The first step in the development of a program satisfying the specification is the creation of a predicate $I(j, l, A)$ which generalises $\mathit{PrefixSum}(l, A)$:

$$I(j, l, A) \;=_{df}\; \forall i.1 \leq i \leq 2^j . a_i' = \sum_{1 \leq k \leq i} a_k \quad \wedge \quad \forall i.2^j < i \leq 2^l . a_i' = \sum_{i - 2^j < k \leq i} a_k$$

It follows that $I(0, l, A) = II$ and that $I(l, l, A) = \mathit{PrefixSum}(l, A)$. A program which satisfies *PrefixSum* can be constructed by finding a mechanism for transforming $I(0, l, A)$ into $I(l, l, A)$. Using composition [HH98] it can be shown (see refinement 1, Appendix) that

$$0 \leq j < l \quad \Rightarrow \quad I(j, l, A); (||_{2^j < i \leq 2^l} a_i' = a_i + a_{i-2^j}) = I(j + 1, l, A)$$

Thus, in a conventional way
$\mathit{PrefixSum}(l, A) \;=$
$\quad$ **var** $j := 0$; $(j \neq l) * ((||_{2^j < i \leq 2^l} a_i := a_i + a_{i-2^j}) \;;\; j := j + 1)$ **end** j
The parallel assignment $||_{2^j < i \leq 2^l} a_i' := a_i + a_{i-2^j}$ can be refined (see refinement 2, Appendix) to:

$$SS^+(P_1, \ldots, P_{2^l}) \text{ where } P_i \;=_{df}\; (\mathit{put}(a_{i+2^j}, a_i)) \lhd 1 \leq i \leq 2^l - 2^j \rhd II), \;\; 1 \leq i \leq 2^l$$

In this superstep program the set of global variables D_0 is $\{j\}$ and each superstep process $P_i, 1 \leq i \leq 2^l$ has an alphabet $\alpha_W P_i = \{a_i\}, 1 \leq i \leq 2^l$. Consequently:

$\mathit{PrefixSum}(l, A) \;=$
$\quad$ **var** $j := 0$; $(j \neq l) * (SS^+(P_1, \ldots, P_{2^l})$; $j := j + 1)$ **end** j

Finally, the superstep within the loop body can be partitioned (see refinement 3, Appendix):

$\mathit{PrefixSum}(l, A) \;=$
$\quad$ **var** $j := 0$; $(j \neq l) * (||_{J \in PI} SS^+(\{P_j | j \in J\})$; $j := j + 1)$ **end** j

where the partition PI is: $PI = \{\{k, k + 2^j, \ldots, k + 2^l - 2^j\} | 1 \leq k \leq 2^j\}$.

Initially, when $j = 0$, $PI = \{\{1, 2, \ldots, 2^l\}\}$ – hence the initial superstep cannot be decoupled. When $j = 1$ $PI = \{\{1, 3, \ldots, 2^l - 1\}, \{2, 4, \ldots 2^l\}\}$ and the corresponding superstep can be partitioned into sets of even and odd indexed processes. When $j = 2$ $PI = \{\{1, 5, \ldots, 2^l - 3\}, \{2, 6, \ldots, 2^l - 2\}, \{3, 7, \ldots 2^l - 1\}, \{4, 8, \ldots 2^l\}\}$. On the last iteration of the loop $j = l - 1$ and $PI = \{\{1, 1 + 2^{l-1}\}, \{2, 2 + 2^{l-1}\}, \ldots, \{2^{l-1}, 2^l\}\}$: the superstep

is decoupled into 2^{l-1} independent parts. Each successive superstep contains fewer barrier dependencies and, consequently, has greater potential for efficient implementation.

The synchronisation cost for executing *PrefixSum*, for $l \geq 5$, on the multicore architecture described in §1 is given under the assumption that processes are distributed over multicore chips as follows:

- Multicore 1 (MC_1): processes $\{P_i | i \in \{1, 5, \ldots, 2^l - 3\}\}$
- Multicore 2 (MC_2): processes $\{P_i | i \in \{2, 6, \ldots, 2^l - 2\}\}$
- Multicore 3 (MC_3): processes $\{P_i | i \in \{3, 7, \ldots, 2^l - 1\}\}$
- Multicore 4 (MC_4): processes $\{P_i | i \in \{4, 8, \ldots, 2^l\}\}$

The first two iterations of an execution of *PrefixSum* (see $j = 0, 1$ above) require two global synchronisations (cost $2L_3$).

Further assume that processes assigned to MC_1 etc. are internally distributed over cores as follows:

- Core 1 (C_1): processes $\{P_i | i \in \{1, 33, \ldots, 2^l - 31\}\}$
- Core 2 (C_2): processes $\{P_i | i \in \{5, 37, \ldots, 2^l - 27\}\}$

  $\vdots$
- Core 8 (C_8): processes $\{P_i | i \in \{29, 61, \ldots, 2^l - 3\}\}$

Iterations 3,4 and 5 can be synchronised using partitioned synchronisations at the multicore level (cost $3L_2$). Finally the remaining iterations involve synchronisations at core level (cost $(l-5) \times L_1$, provided that the cores have sufficient memory). Thus, a non-partitioned version of the prefix sum program has associated synchronisation cost $l \times L_3$ whereas the partitioned version has associated synchronisation cost $2 \times L_3 + 3 \times L_2 + (l-5) \times L_1$ where it is assumed that $L_3 >> L_2 >> L_1$.

9. Implementation Projections

The proposed framework for developing BSP programs has a simple semantics because it is given at a system level. Unfortunately, a direct implementation of such a system would involve unnecessary synchronisations over statements containing shared variables (e.g. $j := j+1$ in *PrefixSum*). To resolve this problem a renaming process projection is defined. Projection $\mathbf{P}_i^{\{x \mapsto i.x | x \in D_0 \cup \{m\}\}}$ maps a superstep system onto its ith component process where each shared variable, say x, is renamed $i.x$ (see the definition of substitution in §5). $\mathbf{P}_i^f$ is defined by structural induction over superstep programs:

$$
\begin{aligned}
\mathbf{P}_i^f(SS(P_1, \ldots P_k)) &=_{df} \textbf{ var } i.m := \{\}; \mathbf{P}_i^f(P_i); LSYNC_i(\{1, \ldots, k\}); \textbf{ end } i.m \\
\mathbf{P}_i^f(SS(\{P_j | j \in I_1\}) \parallel SS(\{P_j | j \in I_2\})) &=_{df} \textbf{ var } i.m := \{\}; \\
&\qquad \mathbf{P}_i^f(P_i); (LSYNC_i(I_1) \lhd i \in I_1 \rhd LSYNC_i(I_2)); \\
&\quad \textbf{end } i.m \\
\mathbf{P}_i^f(\ x := e\ ) &=_{df} x^f := e^f \\
\mathbf{P}_i^f(\ put(x, e)\ ) &=_{df} i.m := i.m \cup (\{j.x \mapsto e | 1 \leq j \leq k\} \lhd x \in D_0 \rhd \{x \mapsto e\}) \\
\mathbf{P}_i^f(\ P; Q\ ) &=_{df} \mathbf{P}_i^f(P); \mathbf{P}_i^f(Q) \\
\mathbf{P}_i^f(\ P \lhd b \rhd Q\ ) &=_{df} \mathbf{P}_i^f(P) \lhd b^f \rhd \mathbf{P}_i^f(Q) \\
\mathbf{P}_i^f(\ b * P\ ) &=_{df} b^f * \mathbf{P}_i^f(P)
\end{aligned}
$$

Here local synchronisation $LSYNC$ is defined as a communicating CSP [HH98] process:

$$
LSYNC_i(I) =_{df} ((\|^{CSP}_{j \in I - \{i\}} c_{ij}!\alpha P_j \lhd i.m) \|^{CSP} (\|^{CSP}_{j \in I - \{i\}} c_{ji}?j.m)) \rightarrow SYNC(\bigcup_{j \in I} j.m)
$$

where $||^{CSP}$ is the interfering parallel composition operator of CSP and $\{c_{ij} | i, j \in I\}$ is a set of communication channels connecting processes. $LSYNC$ synchronises participating processes in a partition, acquires non-local communication data and updates its variables accordingly. Note that an asynchronous communication

to a "global variable" is projected onto a set of communications to "local variables" – thus, supersteps with communications to "global variables" cannot be partitioned. A system S, with k processes per superstep, is implemented as a message passing system as follows:

$$\|^{CSP}_{i\in\{1,\ldots,k\}} \mathbf{P}_i^{\{x\mapsto i.x|x\in D_0\cup\{m\}\}}(S)$$

The resulting composition can be directly implemented using an MPI-like [MPI95] programming environment.

10. Conclusion

Ideally a computing environment should be designed in a top-down fashion, starting with a high-level programming model and ending with an architecture which supports the required high-level operations. One example of such an environment was the CSP \ OCCAM \ transputer system. Unfortunately, programming languages and computer architectures are usually designed independently. Currently there are acute difficulties (particularly in high performance computing) in finding a programming model which has a simple semantics appropriate for program development and which, at the same time, can be efficiently mapped onto multicore architectures.

Co-Array Fortran (CAF) is a Single Program Multiple Data parallel language designed to exploit the computing potential of multicore architectures: processes can asynchronously read and write non-local data. Unless great care is taken when using synchronisation statements, process compositions may be associated with multiple execution paths - some of these may be unwanted (e.g. data a is read by P_1 from P_2 before P_2 has initialised a). It is highly undesirable that high-level languages offer the potential for programs to be subject to race conditions. It is proposed here that the BSP computational model should be re-evaluated for the following reasons:

1. Multicore chips are now being designed with 16 and 32 cores – on-chip synchronisation in such environments has the potential to be relatively inexpensive compared with synchronisation on distributed processor architectures.
2. For large scale architectures comprising numerous multicore chips the multilevel BSP model offers the potential for efficient execution of many applications (e.g. divide and conquer algorithms) through the use of partitioned synchronisation.
3. It has been demonstrated in this paper that multilevel BSP has an associated programming model with simple reasoning and refinement laws

It is proposed that BSP programs be constructed in two phases: an initial sequential-like system development followed by a projection onto localised process computations. The first phase provides the benefits of reasoning about systems in a sequential-like way while there is potential for developing tools to carry out the second phase automatically.

Acknowledgement
The author is grateful to M. Clint J. Gabarro, P. Kilpatrick and the referees for their comments on earlier drafts of this paper.

References

[Akl89] Akl SG (1989) The Design and Analysis of Parallel Algorithms, Prentice-Hall, Englewood Cliffs, NJ.

[BvW98] Back R-J and von Wright J (1988) Refinement Calculus: A Systematic Introduction, Springer, New York.

[Bk09] Back R-J (2009) Invariant based programming: basic approach and teaching experiences, Formal Aspects Comput 21(3): 227-244.

[Bh03] Backhouse R (2003) Program Construction: Calculating Implementations from Specifications, Wiley.

[Bis04] Bisseling RH (2004) Parallel Scientific Computation: A Structured Approach using BSP and MPI, Oxford University Press.

[CS03] Chen Y and Sanders JW (2003) Top-Down Design of Bulk-Synchronous Parallel Programs, Parallel Processing Letters **13** 389–400.

[CS04] Chen Y and Sanders JW (2004) Logic of global synchrony, ACM Trans. Program. Lang. Syst. **26** 221–262.

[Dij68] Dijkstra E. W.: A constructive approach to the problem of program correctness, BIT 8, 3 (1968), 174–186.

[Dij76] Dijkstra EW (1976) A Discipline of Programming, Prentice-Hall, Englewood Cliffs, NJ.

[HMC96] He J, Miller Q and Chen L (1996) Algebraic Laws for BSP Programming in Euro-Par'96, LNCS **1124** Volume II, Springer, eds: L. Bouge, P. Fraigniaud, A. Mignotte and Y. Robert, 359–368.

[Hoa69] Hoare CAR (1969) An axiomatic basis for computer programming, Communications ACM, **12** (10) 576–580.

[HH98] Hoare CAR and He J (1998) Unifying Theories of Programming, Prentice Hall.

[JL09] Javed N and Loulergue F (2009) OSL: Optimized Bulk Synchronous Parallel Skeletons on Distributed Arrays, in APPT 2009, LNCS Vol 5737, eds: Y Dou, R Gruber and J M Joller, 436–451.

[LHF00] Loulergue F, Hains G and Fosy C (2000) A calculus of BSP Programs, Science of Computer Programming, **37**(1-3) 253–277.

[MPI95] Snir M, Otto S, Huss-Lederman S, Walker D and Dongarra J (1995) MPI: The Complete Reference, MIT Press Cambridge, MA, USA.

[McC95] McColl WF (95) Scalable computing, in Computer Science Today: Recent Trends and Developments LNCS **1000**, Springer-Verlag, ed J. van Leeuwen 46–61.

[SC01] Stewart A and Clint M (2001) BSP-style computation: a semantic investigation, The Computer Journal, **44** (2001) 174–185.

[SCG04] Stewart A, Clint M and Gabarro J (2004) Barrier synchronisation: Axiomatisation and relaxation, Formal Asp. Comput, **16(1)**, 36–50.

[Tisk98] Tiskin A (1998) The Bulk-Synchronous Parallel Random Access Machine, Theoretical Computer Science, **196**, 109–130.

[TK96] de la Torre P and Kruskal CP (1996) Submachine Locality in the Bulk Synchronous Setting, in Euro-Par'96, LNCS **1124** Volume II, Springer, eds: L. Bouge, P. Fraigniaud, A. Mignotte and Y. Robert 352–358.

[Val90] Valiant LG (1990) A bridging model for parallel computation, Comm. ACM, **33** (8) 103–111.

[Val08] Valiant LG (2008) A Bridging Model for Multi-core Computing, in ESA 2008, LNCS **5193**, Springer, eds: D. Halperin and K. Mehlhorn.

Appendix: Program refinement proofs

Refinement 1:
Let $I(j,l,A) \;=\; \forall i.1 \leq i \leq 2^j.\; a'_i = \sum_{1 \leq k \leq i} a_k \quad \wedge \quad \forall i.2^j < i \leq 2^l.\; a'_i = \sum_{i-2^j < k \leq i} a_k.$
Then
$0 \leq j < l \Rightarrow \qquad I(j,l,A); (||_{2^j < i \leq 2^l} a'_i = a_i + a_{i-2^j}) \;\; = \;\; I(j+1,l,A)$

Proof:
Assume that $0 \leq j < l$
Then
$I(j,l,A); (||_{2^j \leq i \leq 2^l} a'_i = a_i + a_{i-2^j})$
= (by the rule of sequential composition and expansion of I)

$$\exists v_1, \ldots v_{2^l}.\; \forall i.1 \leq i \leq 2^j.\; \left(v_i = \sum_{1 \leq k \leq i} a_k\right) \quad \wedge \quad \forall i.2^j < i \leq 2^l.\; \left(v_i = \sum_{i-2^j < k \leq i} a_k\right) \wedge$$
$$\forall i.1 \leq i \leq 2^j.\; a'_i = v_i \quad \wedge \quad \forall i.2^j < i \leq 2^l.\; a'_i = v_i + v_{i-2^j}$$

= (by re-arrangement)

$$\forall i.1 \leq i \leq 2^{j+1}.\; \left(a'_i = \sum_{1 \leq k \leq i} a_k\right) \wedge \forall i.2^{j+1} < i \leq 2^l.\; \left(a'_i = \sum_{i-2^{j+1} < k \leq i} a_k\right)$$

$= I(j+1,l,A)$ □

Refinement 2:
Let $P_i \equiv put(a_{i+2^j}, a_i) \lhd 1 \leq i \leq 2^l - 2^j \rhd \Pi),\;\; 1 \leq i \leq 2^l$
Then:
$SS^+(P_1, \ldots, P_{2^l}) \;\; = \;\; (||_{2^j < i \leq 2^l}\; a'_i := a_i + a_{i-2^j})$

Proof:
$SS^+(P_1, \ldots, P_{2^l})$
= (by expansion)
$SYNC^+(\{a_{i+2^j} \mapsto a_i | 1 \leq i \leq 2^l - 2^j\})$
= (by index rearrangement)
$SYNC^+(\{a_i \mapsto a_{i-2^j} | 2^j < i \leq 2^l\})$
= (definition $SYNC$)
$(||_{2^j < i \leq 2^l}\; a'_i := a_i + a_{i-2^j})$ □

Refinement 3:
$\alpha\; dom(i.m) = \{a_{i+2_j}\} \lhd 1 \leq i \leq 2^l - 2^j \rhd \{\} \quad \Rightarrow \quad \forall J \in PS.\; \left(\bigcup_{j \in J}\; dom(j.m)\right) \;\subseteq\; \bigcup_{j \in J}\; \alpha D_j$
where $PS = \{\{k, k+2^j, \ldots, k+2^l - 2^j\} | 1 \leq k \leq 2^j\}$
and $D_i = \{a_i\},\; 1 \leq i \leq 2^l$

Proof:
Consider an arbitrary $J \in PS$.
J must have the form $\{k, k+2^j, \ldots, k+2^l - 2^j)\}$, for some k, $1 \leq k \leq 2^j$.
Then
$\bigcup_{j \in J} \alpha\; dom(j.m) = \{a_{k+2^j}, a_{k+2^{j+1}}, \ldots, a_{k+2^l - 2^j}\} \subseteq \bigcup_{j \in J}\; \alpha D_j$ □.