Synchronising Asynchronous Communications

A. Stewart and M. Clint

Department of Computer Science,
The Queen's University of Belfast,
Belfast BT7 1NN,
N. Ireland.

Abstract. BSP is a model of parallel computation which employs global synchronisation as a means of ensuring that a set of communications has reached completion. The efficiency properties of the model have been widely investigated. In this paper the model's associated semantic framework is studied. An axiomatic treatment of global synchronisation is presented. The proof rule proposed for synchronisation is evaluated in the context of semantic frameworks for a general parallel process model and for data-parallel computation.

1 Introduction

A general theory of parallel computation, such as CSP [13], is interesting semantically because of the rich variety of ways in which processes may interact. It is this same richness (non-determinism) that makes the task of reasoning about parallel programs challenging. Parallelism in scientific computation is used as a means of increasing execution speed; to this end there is no need to consider a process to be an agent which interacts with an unpredictable environment. Thus, one concern of the scientific programming community has been the development of a model of parallel computation which allows the construction of efficient multi-processor programs and which has an associated set of simple reasoning laws which facilitates proofs of correctness of such programs. A suitable framework for parallel scientific computation may be developed by restricting a general theory [18] - for example, the SIMD [4] [19] and data-parallel models [2] [7] [9] [20].

BSP [21] [14] is another model of computation which restricts the ways in which component processes can interact. It has two features which are of particular interest semantically:

1. global synchronisation (also a feature of the SIMD and data-parallel models); and
2. the separation of communication and computation.

Aspects of the efficiency of the BSP model have been widely investigated [10]. In this paper the semantic framework which underlies the global synchronisation of asynchronously communicating processes is explored and BSP is related to

other computational models (asynchronous communicating processes and data-parallelism). An axiomatic definition [5] [12] of non-blocking (asynchronous) communication (*put* and *get*) and global synchronisation is developed.

Global synchronisation is a useful operation to employ when constructing distributed scientific programs; its postcondition is a global assertion which conjoins the properties of participating processes. Global synchronisation is pleasing semantically because it is more transparent to make statements about an entire data space than to manage a set of disjoint statements (separate process proofs over a partitioned data space) which must be interrelated in "time".

One technique which can be used to achieve efficient distributed computation is to overlap communication and computation within a single process; employment of such a technique may prevent communication blocking and permit the separation of communication and computation. For example, a CSP [13] process may specify the parallel composition of communication and computation as follows:

$$comm - P \parallel comp - P$$

where comm-P is a (synchronised) communication process and comp-P is an independent computation process. Alternatively, in an asynchronous environment the statement $c!!e;\ S$ specifies asynchronous sending of the value of the expression e along the channel c (communication) followed by execution of the statement(computation) S. Operationally, communication and computation can be overlapped after evaluation of e.

Global synchronisation and communication/computation overlapping may be combined to provide a framework for the development of transparent and efficient distributed programs. This may be achieved within either a synchronous or an asynchronous message passing framework. In particular, the provision of synchronisation points for asynchronously communicating processes corresponds closely to conventional implementations of the BSP model.

In Section 2 a general theory of asynchronous communication is presented in order to highlight the benefits of global synchronisation which is then discussed in detail in Section 3. The treatment of asynchronous communication is taken from [8] and the treatment of interference from [1] [3] [17].

2 General Asynchronous Communication

An axiomatic definition of asynchronously communicating processes is presented here in order to illustrate the difficulties of reasoning in such a setting. Consider a parallel distributed computation $\|_{i \in I}\ P_i$ where $\|$ is the parallel composition operator and $\{P_i | i \in I\}$ is a set of processes. Each process has local data and information is exchanged through channels. A channel links two processes and is unidirectional (information may flow in one direction only).

Conventionally, an asynchronous communication along a uni-directional channel c is specified by a statement, $c!!e$, in a sender process P_s where e is a local

expression of P_s and a"matching" statement, $c??x$, in a receiver process P_r, $P_r \neq P_s$, where x is a local variable of P_r. It is assumed that P_s can continue execution after obeying its send instruction whereas P_r is blocked at the receive instruction until the communication is delivered (asymmetry). Messages arrive in the same order as they are sent.

The situation can be modelled by introducing two auxiliary "trace" variables (sequences), OUT_c and IN_c, which record the history of messages sent along, and received from, channel c. It is necessary that the sequence IN_c be a prefix of the sequence OUT_c. A send, $c!!e$, is modelled (locally in a process) by an assignment to OUT_c:

$$\{p^{OUT_c}_{OUT_c::e}\}c!!e\{p\}$$

where $OUT_c :: e$ is the sequence formed by appending the value of the expression e to OUT_c and p^x_y denotes the substitution of free occurrences of x in p by y.

The relationship between the before and after states of a receive depends on the external environment. One approach to capturing the influence of the environment on a process [1] [3] [15] [17] consists in:

1. conducting a local proof using assumptions about the environment; and
2. discharging the outstanding assumptions using process composition arguments.

Thus, locally, a hypothesis may be made about the effect of external influences on a receive, say:

$$\{p\}c??x\{q\}$$

This relationship must now be verified with respect to the sender process. The effect of a "matched" communication is to deliver a value, e, to a variable x thereby updating x and IN_c:

$$\{p^{x,IN_c}_{e,IN_c::e}\}c!!e \parallel c??x\{p\}.$$

However, it is necessary to ensure that only semantically matching communication pairs are engaged in - i.e. the ith send along c must match the ith receive along c. Thus, the concept of co-operation is defined as follows:

a part of a *local* proof $\{p_1\}c!!e\{q_1\}$
co-operates
with a part of a *local* proof $\{p_2\}c??x\{q_2\}$
iff
$\{p_1 \wedge p_2 \wedge IN_c = OUT_c\}c!!e \parallel c??x\{q_2\}$

Note, again, the asymmetry of asynchronous communication. In practice, in order to facilitate proofs, it may be useful to extend the definition of co-operation in order that auxiliary variables can be updated simultaneously with message delivery.

Proofs conducted in a general asynchronous communication framework are made cumbersome by:

1. the need to "match" the two endpoints of a communication;
2. the possible occurrence, anywhere within a process definition, of a receive instruction ; and
3. the fact that communicating processes do not synchronise.

Things can be simplified by placing restrictions on communications - for example, by the imposition of requirements that:

1. processes globally synchronise; and
2. asynchronous communications are delivered to pre-specified global locations before computation can proceed past the synchronisation point, simplify matters in two ways:
   - the need to specify two partners (and, hence, the need for matching) in a communication can be avoided; and
   - the need to discharge intermediate receive assumptions can be avoided (a communication is only "delivered" at a synchronisation point).

The details of this restricted framework are now described.

3 Synchronising Asynchronous Communications

An asynchronous distributed program is a parallel composition of processes. An entity, the superstep [21], is used to specify computation/communication followed by global synchronisation. A superstep denotes a state transformation; supersteps may be combined using conventional sequential program operators (sequential composition, selection and repetition). Thus, supersteps can be considered to be building blocks for programs. For example, the program

$$\|_{k\in\{1,\cdots,100\}}\ T_k; \|_{k\in\{1,\cdots,100\}}\ U_k$$

comprises two supersteps each of which has the same internal process structure. There are implicit global synchronisations at the start and end of the program and also between the two supersteps (specified by the composition operator ";").

The conventional model of distributed computation is based on a set of processes, each of which has local memory. In BSP, however, memory is modelled as a global entity in order to bridge the gap between shared and distributed memory architectures. In a similar way a superstep is denoted by a global state transformation; in particular, a component process in a superstep is considered to have external access to a slice of global memory. The external data of a process P_k may be specified using a mechanism resembling the VDM external facility [16]. For example, a process P_k which has access only to the (global) array segment $A[10..20]$ can be defined as:

P_k::
ext $A[10..20]$
 S

where S is the body of the computation. Thus, data may persist after completion of the execution of P_k. In order to prevent possible interference on the global memory it is necessary to ensure that no two components of a superstep have access to a common external data element. Let

$$disjoint(\{P_i | i \in I\})$$

mean that each P_i's, $i \in I$, access to external data does not interfere with any other process's access to external memory (disjoint data spaces). The details of such a partition are straightforward and are not explicitly discussed.

Global synchronisation points can be interpreted as implicit"receive communicatio" commands. In this way the need for two separate communication instructions can be avoided. This semantic simplification is acceptable because the information from incoming communications cannot be used during a superstep. Thus, information sent during a *particular* superstep can only be used in *subsequent* supersteps.

The elimination of the receive instruction allows a further simplification in the specification of a communication: a communication can be considered to be the evaluation of a local expression, e, the asynchronous communication of the resulting value and its assignment, at the synchronisation point, to a prespecified variable (address). Operationally, the value may be delivered to the receiver process at any point in the superstep; assignment to the"variable name" may take place before the global synchronisation point if it can be guaranteed that the variable name is not referenced in the remainder of the superstep. Only the case where a value assigned through communication to an array element (or to a subarray) is considered. For example, the statement

$$put(e,\ y[j])$$

specifies that the value of the expression e is to be communicated asynchronously to the appropriate process and assigned, on completion of the current superstep (or before, if legal), to $y[j]$. Note: the sending process should not have external access to $y[j]$.

The definition of send, $c!!e$, can be modified to model the semantics of *put*: rather than appending a value to a channel sequence, OUT_c, a value is marked as "in transit" to a target destination. An asynchronous communication may be viewed operationally as the transmission of a message over a network (or a bus in the case of shared memory machines). On completion of a superstep (or before, if the situation outlined above obtains) the message is copied into its designated location. The asynchronous transfer of information may be modelled by means of a function, *Transfer*, which maps variable names (addresses) and associated indices (if any) to values. An individual "in transit" function is defined for each process. Let $Transfer_i$ denote the "local" communication function of process i. The effect of an asynchronous *put* operation within process i is defined by modifying the asynchronous send rule (see [8]):

$$\frac{\{name(y)(j)\} \notin dom(Transfer_i)}{\{R^{Transfer_i}_{Transfer_i \dagger (name(y)(j) \rightarrow e)}\} put(e, y[j]) \{R\}} \qquad \textbf{(Rule put)}$$

The assumption $\{name(y)(j)\} \notin dom(Transfer_i\}$ ensures that multiple messages to the same "recipient" are never generated from within a single process. The rule has recourse to a function *name* in order to distinguish the name of a variable from its usual denotation (its value). $Transfer_i \dagger (name(y)(j) \rightarrow e)$ is a variant of the conventional notation for array updating [6] [11] and is defined by:

$$Transfer \dagger (name(y)(j) \rightarrow e)(z) = Transfer(z), \qquad name(z) \neq name(y)$$

$$Transfer \dagger (name(y)(j) \rightarrow e)(y) = \lambda k. \begin{cases} e & \text{if } j = k \\ Transfer(y)(k) & \text{otherwise} \end{cases}$$

In particular, subsequent assignments to the array y should not affect the destination of a value "in transit". Thus,

$$name(y) = name(y; i : f)$$

where $(y; i : f)$ denotes an array which is identical to y except at the point i where it has the value f [5] [6] [11]. Although subsequent assignments to y affect the conventional denotation(a function from index positions to values) they have no effect on the "name" interpretation - i.e. substitution is an identity operation through the function *name*; thus, $name(z)^x_e = name(z)$ where x is a meta-name and e is a meta-expression. Note that index positions in assertions are treated as values in the conventional way.

It is necessary to provide a non-interfering composition rule which ensures that the "in transit" data of each process are disjoint - i.e. it is vital that processes do not initiate conflicting data transfers through two or more communications to the same destination. The final state of a process records all local messages that have been sent. Thus, this requirement may be defined by:

$$non - interfering(\{Q_i\}, i \in I\}) \hat{=}$$
$$\wedge_{i \in I} Q_i \Rightarrow \forall i, j \neq i \in I. dom(Transfer_i) \cap dom(Transfer_j) = \emptyset$$

i.e. at the point immediately preceding the barrier (synchronisation) it is guaranteed that there are no conflicting communications. In non-interfering situations the composition of individual communication functions is non-problematic: if the domains of the component functions are mutually disjoint then the union of the individual functions will also be a function:

$$Transfer = \cup_{i \in I} Transfer_i$$

Global synchronisation transforms the constituent process postconditions (expressed as the conjunction of process post-conditions $\wedge_{i \in I} Q_i$) into the superstep post-condition (specified as an assertion R). Semantically, the barrier ensures that all communications are delivered to their destinations and then deletes all message routing information. Thus, the relationship between the pre- and post-conditions for synchronisation is given by:

$$\{R_{\emptyset}^{Transfer} \diamond Transfer\}$$
$$\forall x \in dom(Transfer).x := Transfer(x); Transfer := \emptyset$$
$$\{R\} \qquad (*)$$

where $\diamond$ is a predicate transformer which captures the concept of message delivery. The operator $\diamond$ is defined inductively over assertions as follows:

$$y(k) \diamond b = \begin{cases} b(y)(k) & \text{if } name(y)(k) \in dom(b) \\ y(k) & \text{otherwise} \end{cases}$$
$$(\circ P) \diamond b = \circ(P \diamond b)$$
$$(P \oplus Q) \diamond b = (P \diamond b) \oplus (Q \diamond b)$$

where P and Q are assertions and $\circ$ and $\oplus$ are unary and binary operators, respectively. In other words, transforming R to $R \diamond b$ ensures that all stores associated with message destinations, say d, belonging to $dom(b)$ are overwritten by the associated values being communicated, $b(d)$. The global synchronisation of a set of processes, given by the process bodies $\{S_i | i \in I\}$ is defined by:

$$\frac{\begin{array}{l}\{P_i \wedge Transfer_i = \emptyset\} S_i \{Q_i\}, i \in I \\ disjoint(S_i) \\ non - interfering(\{Q_i\}, i \in I\}) \\ \wedge_{i \in I} Q_i \Rightarrow (R_{\emptyset}^{Transfer}) \diamond \bigcup_{i \in I} Transfer_i\end{array}}{\{\wedge_{i \in I} P_i\} \parallel_{i \in I} S_i \{R\}} \qquad \textbf{(Rule syn)}$$

A simple proof which illustrates the use of this rule is given below.

4 Data-Parallel Assignment

Data-Parallel array assignment [9] [19] is a special kind of BSP operation. The data-parallel assignment $\forall i \in S.a(i) := E(i)$ denotes a set of array assignments determined by the index set S. In order to prevent interference, the expressions on the right hand side are evaluated before the assignments are made. The purpose of this section is to recast data-parallel assignment within the BSP framework and to prove that the predicate transformer for data-parallel assignment [9] [20] can be derived from the proof rules for BSP. In order to facilitate the expression of data-parallel assignment in BSP an additional operator is introduced: *get* is the counterpart of the operation *put*; a *get* operation within a process is a request that an expression (over the global store) be evaluated and assigned to a local memory location at the end of the superstep.

One important distinction between $put(e, y[j])$ and $get(e, y[j])$ relates to the evaluation of e. If e is purely "local" (i.e. the process has external access to the variables of e) then it is evaluated in the current state. If e refers to "non-local" data then an assumption about its point of evaluation needs to be made. The assumption is that index evaluation takes place in the current state (indices are

assumed to be "local") while outermost array references are "non-local" and are evaluated at process *termination* (i.e. the synchronisation point). Consider, for example, the operation $get(a[i], y[j])$. This specifies that the indices i and j are evaluated in the *current* state and that the reference to the expression $a[i]$ is evaluated when all processes have terminated. Operationally, $a[i]$ may be evaluated earlier in the computation if it can be guaranteed that the same value will result. Let e^f, where e is an expression, denote e with the outermost array references decorated with the symbol $\rightarrow$. Such decoration is used to denote the value of a variable in the final state produced by a process and is similar to the initial state mechanism of VDM [16] - for example,

$$(x[i] + y[z[k]])^f = \vec{x}\ [i] + \vec{y}\ [z[k]]$$

In the final state of a process $x = \vec{x}$ for all array variables x. Assignments subsequent to $get(e^f, y[j])$ have no effect on the "final state" array variables in e^f (a variable in the final state can be evaluated only in one state and is, consequently, a constant). The meaning of a *get* operation is defined by:

$$\frac{\{name(y)(j)\} \notin dom(Transfer_i)\}}{\{R^{Transfer_i}_{Transfer_i \dagger (name(y)(j) \rightarrow e^f)}\} get(e, y[j]) \{R\}}$$

(Rule get)

The data-parallel assignment $\forall i \in G.a(i) := E(i)$ can be represented as a BSP superstep where a process T_i is used to update $a[i]$, for each element i of G:

$T_i ::$

ext $a[i]$

$\{R_i \ ^a_{(a;i:E(i))} \wedge Transfer_i = \emptyset\}$

$get(E(i), a[i])$

$\{R_i \ ^a_{(a;i:E(i))} \wedge Transfer_i = \{name(a)(i) \rightarrow E(i)\}\}$ **(A0)**

where R_i is an assertion which does not involve the function $Transfer$. The correctness of T_i follows from **Rule get**. It is now shown that the superstep $\|_{i \in G}\ T_i$ effects the state transformation

$\{\wedge_{i \in G} R_i \ ^a_{(a;i:E(i))}\}\ \|_{i \in G}\ T_i \{\wedge_{i \in G} R_i\}$.

Proof:

1. proof **(A0)** above.
2. $disjoint(T_i, i \in G)$ follows since the external data of T_i is $a[i]$ and G is a set.
3. $non - interfering(\{R_i \ ^a_{(a;i:E(i))} \wedge Transfer_i = \{name(a)(i) \rightarrow E(i)\}\}, i \in G) =$

$$\wedge_{i \in G} R_i \ ^a_{(a;i:E(i))} \wedge Transfer_i = \{name(a)(i) \rightarrow E(i)\} \Rightarrow$$

$$\forall i, j \neq i \in G.dom(Transfer_i) \cap dom(Transfer_j) = \emptyset$$

<since $i \neq j \Rightarrow name(a)(i) \cap name(a)(j) = \emptyset$ >.

4. $\wedge_{i \in G} R_i \,{}^{a}_{(a;i:E(i))} \wedge Transfer_i = \{name(a)(i) \rightarrow E(i)\} \Rightarrow$
$((\wedge_{i \in G} R_i)^{Transfer}_{\emptyset}) \diamond \cup_{i \in G} Transfer_i$
Proof:
$((\wedge_{i \in G} R_i)^{Transfer}_{\emptyset}) \diamond \cup_{i \in G} Transfer_i = (\wedge_{i \in G} R_i) \diamond \cup_{i \in G} Transfer_i$
<since every R_i is free of $Transfer$ variables> =
$(\wedge_{i \in G} R_i) \diamond \cup_{i \in G} \{name(a)(i) \rightarrow E(i)\}$
<from antecedent> =
$(\wedge_{i \in G} R_i \diamond \{name(a)(i) \rightarrow E(i)\})$
<since the external data of T_i is $a[i]$ then $var(R_i) = a[i]$ > =
$\wedge_{i \in G} R_i \,{}^{a}_{(a;i:E(i))}$
<by definition of $\diamond$ and array substitution> =
$true$
<from antecedent>.
5. $\{\wedge_{i \in G} R_i \,{}^{a}_{(a;i:E(i))}\} \,\|_{i \in G}\, T_i \{\wedge_{i \in G} R_i\}$
< **Rule syn**, 1, 2, 3, 4 >
6. $\wedge_{i \in G} VAR(Q_i) \subseteq \{a(i)\} \Rightarrow (\wedge_{i \in G} Q_i \,{}^{a}_{(a;i:E(i))}) = (\wedge_{i \in G} Q_i)^{a}_{(a;j \in G:E(i))}$
<exercise, using the definition of simultaneous substitution below>.
7. $\{R^{a}_{(a;i:E(i))}\} \,\|_{i \in G}\, T_i \{(R)\}$
<5,6, the definition of simultaneous substitution

$$(a; j \in G : E(j))(k) = \begin{cases} E(k) & \text{if } k \in G \\ a(k) & \text{otherwise} \end{cases}$$

and letting $R = \wedge_{i \in G} R_i$>.

In this way **Rule Data-Parallel** (see [9] [20]) is indirectly derived from **Rule syn.**

5 Discussion

Proof rules for reasoning about synchronisation have been presented. **Rule get** and **Rule put** are variants of standard assignment (to auxiliary variables) rules and, thus, soundness and completeness results follow immediately. No explicit soundness result has been derived for **Rule syn.** However, it has been shown that synchronisation with communication delivery can be modelled using a variant of data-parallel assignment (see (*) §3). In this case the uniformity of data-parallelism is simulated through the function $Transfer$. In particular, individual assignments are carried out for each domain/range pair of the function. Thus, the soundness result for data-parallel assignment carries over to **Rule syn.**

Rule Data-Parallel is simpler than **Rule syn** for two reasons: (i) a data parallel array assignment does not explicitly specify the method of data transfer between parallel threads and thus obviates the need for the $Transfer$ function and related operators; and (ii) a process model requires the specification of data partitioning details which are used in the construction of correctness proofs. In contrast, it is possible to separate the concerns of proof and efficiency in the data-parallel model; partition details are crucial to the realisation of efficient computations but correctness arguments are independent of the memory organisation (see **Rule Data-Parallel** [9] [20]).

References

1. Apt K. R., Francez N., de Roever W. P.: A proof system for communicating sequential processes. ACM Trans. Programming Languages Systems,. **2** (3) (1980) 359–385.
2. Bouge L., Le Guyadec Y., Virot B., Utard G.: On the expressivity of a weakest precondition calculus for a simple data-parallel programming language. Parallel Processing: CONPAR 94 -VAPP VI, eds: B. Buchberger & J. Volkert, LNCS 854, Springer-Verlag, pp 100–111, 1994.
3. Clint M.: Program proving: coroutines, Acta Informatica, **2** (1) (1973) 50–63.
4. Clint M., Narayana K. T.: Programming structures for synchronous parallelism, Parallel Computing 83, eds: F. Feilmeier, J. Joubert, U. Schendel, North-Holland, pp 405–412, 1984.
5. Dahl O.-J.: Verifiable Programming, Prentice-Hall International, 1992.
6. Dijkstra E. W.: A Discipline of Programming, Prentice-Hall, 1976.
7. FORTRAN 90 International Standard, ISO : IEC 1539 : 1991.
8. Francez N.: Program Verification, Addison-Wesley, 1992.
9. Gabarro J., Gavalda R.: An approach to correctness of data parallel algorithms, Journal of Parallel and Distributed Computing, **22** (1994) 185–201.
10. Gerbessiotis A. V., Valiant L. G.: Direct bulk-synchronous parallel algorithms, Journal of Parallel and Distributed Computing, **22** (1994) 251–267.
11. Gries D.: The Science of Programming, Springer-Verlag, 1981.
12. Hoare C. A. R.: An axiomatic basis for computer programming, Comm. ACM, **12** (10) (1969) 576–580.
13. Hoare C. A. R.: Communicating Sequential Processes, Prentice Hall, 1985.
14. Jifeng H., Miller Q., Chen L.: Algebraic laws for BSP programming, Euro-Par 96 Parallel Processing, Vol. 2, eds: L. Bouge, P. Fraigniaud, A. Mignotte, Y. Robert, LNCS 1124, Springer-Verlag, pp 359–368, 1996.
15. Jones C. B.: Tentative steps towards a development method for interfering programs, ACM Trans. Programming Languages Systems,. **5** (4) (1983) 596–619.
16. Jones C. B.: Systematic Software Development using VDM (2nd edn.), Prentice Hall, 1990.
17. Levin G., Gries D.: Proof techniques for communicating sequential processes, Acta Informatica, **15** (1981) 281–302.
18. Owicki S., Gries D.: An axiomatic proof technique for parallel pograms, Acta Informatica, **6** (1976) 319–340.
19. Stewart A.: An axiomatic treatment of SIMD assignment, BIT, **30** (1990) 70–82.
20. Stewart A.: Reasoning about data-parallel array assignment, Journal of Parallel and Distributed Computing, **27** (1) (1995) 79–85.
21. Valiant L. G.: A bridging model for parallel computation, Comm. ACM, **33** (8) (1990) 103–111.