Formal Refinement of BSP Programs with Early Cost Evaluation

Virginia Niculescu
Faculty of Mathematics and Computer Science
Babeş-Bolyai University
Cluj-Napoca, România
Email: vniculescu@cs.ubbcluj.ro

***Abstract*—The paper presents a method that allows formal refinement of BSP programs. A parallel program is considered as a number of cooperating parameterized processes with similar structures. The method uses parameterized pre- and post-conditions, and takes into account the data-distribution, even at the beginning of the construction process. The possibility of counting the number of communications from postconditions, allow us to make a cost evaluation even at the early stages of the design, and so it leads us to the right decisions.**

***Keywords*-parallel computation; abstraction; model; refinement;data-distribution;**

I. Introduction

In parallel computing, the wide range of possible platforms means that testing cannot do more than skim the surface of a program's behavior. Hence, it is very important, in this case, that programs be correct by construction [13]. Also a big difference in a parallel setting is that the application of a single formal construction can often build the entire parallel structure of the program.

The parallel programming model BSP – *Bulk Synchronous Parallelism* has been proved as being a successful choice [15], [2]. Because it separates communication from synchronization, it is particularly clean and simple. This separation allows us to develop a simple and formal software construction process for it.

For BSP programs, a number of software development methodologies have been proposed.

D. Skillicorn proposed a refinement calculus for BSP programs [12], which is an extension of the classic Refinement Calculus [10]. The calculus introduces some new constructions: *distribute*, *collect*, *redistribute*, with the appropriate laws. The frames are not partitioned, and the global state is used together with some location predicates.

In [3] it is presented a top-down design method of BSP programs, based on a specification formalism LOGS. BSP is seen here as a parallel programming paradigm based on variable sharing.

Another idea has been proposed by Hoare, He, and Lecomber [6], [9]: a simple semantic is given, in which each processor can act arbitrarily on any variables, with the final effect of a superstep being defined by a *merge* operator; it defines what happens if multiple assignments to the same variable are made by different processors.

BSP-Why is a tool for verifying BSP programs [7]. It is intended to be used as an intermediate core-language for verification tools (mainly condition generators) of BSP extensions of realistic programming languages such as C, JAVA, etc. BSP-Why is based on a sequential simulation of the BSP programs which allows to generate pure sequential codes for the back-end condition generator Why and thus benefit of its large range of existing provers. In this manner, BSP-Why is able to generate proof obligations for BSP programs.

In the BSP model, the partitioning of the data is a crucial issue, as opposed to the mapping of the resulting partitions to particular processors, which is irrelevant [2]. In fact, the choice of a data distribution is one of the main means of influencing the performance of the algorithm. This leads to an emphasis on problem dependent techniques of data partitioning, instead of on hardware dependent techniques that take network topologies into account. The algorithm designer who is liberated from such hardware considerations may concentrate on exploiting the essential features of the problem.

On an abstract level, BSP is defined as a distributed memory model with point-to-point communication between processors. So, a formal method for BSP program construction based on distributed-memory paradigm is appropriate. The development of shared-memory based methodologies is justified by the intense use of DRMA (Direct Remote Memory Access) primitives. Even if the BSPLib standard defines them, not all the implementations of BSP model use these primitives, so to base the development methodology on a distributed memory model with point-to-point communications is a justified idea. These kinds of communications are directly mapped into BSMP (Bulk Synchronous Message Passing) operations, and also may be easily transformed into DRMA operations if these are more efficient on a particular architecture. The proposed refinement method is based on a distributed-memory model that uses point-to-point communications. An important advantage of the method is that the effective cost model of BSP is used by the design method in order to make the right decisions.

II. Bulk Synchronous Parallelism

The BSP model was proposed by Valiant [15] as a "bridging model" that provides a standard interface between parallel architectures and algorithms. A BSP computer contains a set of processor-memory pairs, a communication network allowing inter-processor delivery of messages, and a global synchronization unit that executes collective requests for a synchronization barrier. The computation is divided into *supersteps* separated by global synchronization steps, and packets sent in one superstep are assumed to be delivered at the beginning of the next superstep.

The properties of architectures are captured by four parameters. These are: the raw speed of the machine (which can be ignored by expressing the remaining parameters in its units), the number of processors – p, the time required to synchronize all processors – l *latency*, and the ability of the network to deliver messages under continuous load – g *gap* (which reflects network bandwidth on a per processor basis).

The BSP model ignores the particular topology of the underlying machine; this rules out any use of network locality in algorithm design. Thus, the model only considers two levels of locality, local (inside the processor) and remote (outside a processor), with remote access usually being more expensive than local ones.

The cost of a superstep is given by:

$$cost = w + hg + l$$

where w is the maximum local computation in any processors during the superstep, h is the maximum number of global communications into or out of any processor during the superstep.

The organization of programs as a sequence of supersteps reduces the complexity of arranging communication and synchronization. This makes it straightforward to extend techniques for constructing sequential programs to BSP programs.

III. Formal Refinement of BSP Programs

The method considers a parallel program as a number of cooperating *parameterized* processes with similar structures. A parallel program is considered to be formed of many parameterized processes $S.q(0 \leq q < p)$, which are running in parallel. There is no shared memory, and point-to-point communication is considered. Since in BSP model, a parallel machine consists of a set of processors, each with its own private memory, and an interconnection network that can route packets between processors, we may reason about BSP programs as a set of parameterized processes that communicate via message-passing.

A parameterized process is much like a procedure in sequential programming. The difference is that, instead of having only one instantiation in a sequential program, we have many instantiations in a parallel program.

In sequential programming the Hoare-triple [5]

$$\{Q\}S\{R\}$$

is commonly used to denote a formal specification of a program S. This notation expresses that if the program S starts in a state described by the predicate Q, and the program terminates, then, upon completion, the predicate R is satisfied (partial correctness).

For specifying a parallel program, both pre- and post-conditions, Q and R, are split up as conjunctions of $p(p > 0)$ local pre- and post-conditions, and a process is associated with each such pair. So, we may specify a parallel program as follows:

$$\begin{array}{l} |[\{Q\} \\ \quad par\ q : 0 \leq q < p : \\ \quad \{Q.q\} \\ \quad\quad S.q \\ \quad \{R.q\} \\ \quad rap \\ \{R\}]| \end{array}$$

A parameterized specification refers local variables, the number of processes (p), the index of the local process (q), and also elements of the distributed objects which represent the input of the problem. Therefore, *the parameters of a specification* are q, p, and a data distribution D. Many choices are possible for D, each of them having an impact on the complexity of the parallel program.

Such a specification forms the starting point for a parallel program derivation, a formal construction of parameterized processes constituting a parallel program.

Our approach for obtaining a parameterized process S from a functional specification is similar to the methods used in sequential programming. We may use the classic rules for derivation from sequential programming, and new rules for parallel composition and communication. The classic method that obtains an invariant from a specification, in a calculational style, may be used. The programs are derived by calculating the necessary conditions to maintain the invariant. Because we have parameterized specifications, we will derive parameterized invariants.

A. Communication

A parameterized process is refined into a sequence of ordinary sequential processes and communication processes. A parallel program is, thus, decomposed into layers of process instances.

All the communications occur between instances of the same parameterized communication process. Such a process is called *communication closed*. Therefore, a parallel program may be considered as being decomposed into layers, each layer being either a sequential process or a communication process. In order to respect the BSP program structure, we consider that each sequential layer is followed by a communication layer, which is ended by a global

synchronization barrier. So, a BSP superstep is formed of a computation layer and a communication layer.

Non-blocking point-to-point communications are considered, and because of this if we have to send more messages from one process to another, the order in which the messages arrive is not determined. We may impose a constraint: in a communication layer only one communication may be done between two particular processes. This constrain does not restrict the communications between processes because if more than one message has to be sent from process s to process r, then messages may be packed into one message (they form a list), since anyway BSP model assumes that the communications are completed only at the end of each superstep.

B. Program Notation

The Dijkstra notation [1] is used for our program notation, and it is enlarged with some parallel specific constructs.

Parallel composition is denoted by:

- **par** $q : 0 \leq q < p : S.q$ **rap**.

Communication is expressed by the constructs:

- $r \; ! \; \overline{e}$
  where $\overline{e}$ is a list of expressions (or only one), and the statement means that these expressions are output to process r;
- $s \; ? \; \overline{x}$
  where $\overline{x}$ is a list of variables (or only one variable), and the statement means input from process s a list of values that are assigned to $\overline{x}$.

A *send* command $r!\overline{e}$ and a *receive* command $s?\overline{x}$ form a *matching pair* if and only if $r!\overline{e}$ appears in process s, $s?\overline{x}$ appears in process r, and $\overline{x} := \overline{e}$ is a syntactically legal multiple assignment.

The synchronization barrier is expressed by the statement **barrier** that has to be introduced at the end of any communication layer.

C. Proving Rules

Beside the classic axioms and the inference rules of sequential programming, some new specific rules are introduced.

For **barrier** we will not introduce any rule or axiom since that statement is implicitly added at the end of any communication layer.

The following rules and axioms are adapted from Levin and Gries [8].

For communication we have two axioms:

Input Axiom: $r :: \; \{true\} \; s?x \; \{M.x\}$
Output Axiom: $s :: \; \{P\} \; r!e \; \{P\}$

The input axiom stipulates that in isolation anything can be concluded after receiving a value, and the output axiom stipulates that sending a value does not change the state of the sending process.

In order to relate these two communication statements a *Rule of Satisfaction*, as in [8], is introduced. This rule has to be proved after each barrier and states that:

- for every receive statement there is a matching send statement and vice versa;
- for every matching pair of the form

$$r :: \; \{true\} \; s?x \; \{M.x\}$$
$$s :: \; \{P\} \; r!e \; \{P\}$$

we have $P \Rightarrow M.x[x \backslash e]$

This way deadlock becomes impossible and the deadlock-free property of BSP is satisfied.

The postcondition of the receiving process r follows the statement, even if that postcondition is satisfied only after the synchronization barrier. We allow this, because the received value is not used anywhere else in the corresponding communication process that is derived. (A process does not have the role of an intermediary in sending a value; the formal derivation excludes this and in BSP this is not justified.)

For the parallel composition we have the following *par*-rule:

Par Rule:

$$\frac{(\forall i : 0 \leq i < p : \{Q.i\} \; S.i \; \{R.i\})}{\{(\forall i : 0 \leq i < p : Q.i)\} \; par \; q : 0 \leq q < p : S.q \; rap \; \{(\forall i : 0 \leq i < p : R.i)\}}$$

The precondition of the par-statement is the conjunction of all preconditions of each process $S.q$. A similar remark holds for postcondition. The state spaces of the processes are disjoint since each process has each its own set of local variables. Execution of one process, therefore, cannot alter the state of any other process, except when communication occurs. And, in the communication layers the rule of satisfaction has to be proved.

We also use the result of Elrad and Francez [4] that says: "Any distributed program is equivalent to any of its safe decompositions into layers". Decomposition into layers is safe if all the layers are communication closed. This means that a parallel program

$$S :: \; par \; q : 0 \leq q < p : S.q \; rap$$

is equivalent to

$$S :: L_0, \ldots L_{d-1}$$

where $L_j :: \; par \; q : 0 \leq q < p : S_j.q \; rap$ are communication-closed, and d is the depth of the decomposition.

The rule of satisfaction is applied to each layer after achieving the decomposition into layers.

Example 1: An Example – Parallel Prefix

This section presents a formal derivation for a BSP program for parallel prefix problem. We make a strict separation of computation and communication layers, and we analyze the complexity of the obtained program.

The specification of the problem in terms of our notation is:

$$
\begin{array}{l}
|[k, p : int; \\
\quad x(i : 0 \leq i < p) : \text{array of int}; \\
\quad \{0 \leq k \wedge p = 2^k\} \\
\quad par\ q : 0 \leq q < p : \\
\quad\quad |[\ m : int; \\
\quad\quad\quad S.q \\
\quad\quad\quad \{R.q : m = M.0.(q+1)\} \\
\quad\quad]| \\
\quad rap \\
\quad \{(\forall q : 0 \leq q < p : m_q = M.0.(q+1))\} \\
]|
\end{array}
$$

where $M.a.b = (\odot i : a \leq i < b : x(i))$, and the operator $\odot$ is an associative operator.

We assume that the integer array x is distributed by assigning the element $x(q)$ to process q (identity distribution).

The global postcondition is:

$$R : (\forall q : 0 \leq q < p : m_q = M.0.(q+1))$$

where m_q is the variable m from process q.

We split the global postcondition into p local postcondition, by taking into account the distribution. So, the local postconditions are:

$$R.q : m = M.0.(q+1)$$

The relation $(\wedge q : 0 \leq q < p : R.q) \equiv R$ holds, and this means that the **par**-rule may be applied.

The local invariants may be obtained by introducing a variable $t, 0 \leq t \leq k$:

$$
\begin{array}{l}
P.q : P0.q \wedge P1.q \\
P0.q : 0 \leq t \leq k \\
P1.q : 0 \leq q < 2^t \Rightarrow m = M.0.(q+1)
\end{array}
$$

The progress is obtained by increasing the variable t:

$$
\begin{array}{ll}
& P1.q : (t := t+1) \\
\equiv & \{\text{definition}\} \\
& 0 \leq q < 2^{t+1} \Rightarrow m = M.0.(q+1) \\
\equiv & \{\text{range splitting}\} \\
& P1.q \wedge (2^t \leq q < 2^{t+1} \Rightarrow m = M.0.(q+1)) \\
\\
& m = M.0.(q+1) \\
\equiv & \{\text{definition of } M, \text{range splitting}, \\
& \quad M.a.c = M.a.b \odot M.b.c \text{if } 0 \leq a < b < c \leq p\} \\
& m = M.0.(q-2^t+1) \odot M.(q-2^t+1).(q+1)
\end{array}
$$

The value $M.0.(q-2^t+1)$ is, on account of $P1$, known in process$(q-2^t)$, while the value $M.(q-2^t+1).(q+1)$ is unknown. This suggests us to strengthen P with invariant $P2$, which ensures that the value of $M.(q-2^t+1).(q+1)$ is recorded in m for all processes q with $q \geq 2^t$.

$$
\begin{array}{l}
P.q : P0.q \wedge P1.q \wedge P2.q \\
P2.q : 2^t \leq q \Rightarrow m = M.(q-2^t+1).(q+1)
\end{array}
$$

We analyze now the progress for the invariant $P2$:

$$
\begin{array}{ll}
& P2.q : (t := t+1) \\
\equiv & \{\text{definition}\} \\
& 2^{t+1} \leq q \Rightarrow m = M.(q-2^{t+1}+1).(q+1) \\
\equiv & \{\text{range splitting}\} \\
& 2^{t+1} \leq q \Rightarrow m = M.(q-2^{t+1}+1).(q-2^t+1) \odot \\
& \qquad\qquad M.(q-2^t+1).(q+1)
\end{array}
$$

The term $M.(q-2^t+1).(q+1)$ is known on account of $P2$, and the term $M.(q-2^{t+1}+1).(q-2^t+1)$ is known in process $q-2^t$.

So, there are three kinds of processes: each process q with $0 \leq q < 2^t$ has trivially restored $P1$ and $P2$, each process q with $2^t \leq q < 2^{t+1}$ has to restore $P1$ using the value m of process $q-2^t$, and finally each process $2^{t+1} \leq q < 2^k$ has to restore $P2$ using the value of process $(q-2^t)$. Therefore, in iteration t, process q should engage in a receiving from process $(q-2^t)$ and in a sending to process $(q+2^t)$, if these exist.

The variable m in process q is initialized with the element of x that it is distributed to process q: $x(q)$, and so the invariant holds at the beginning.

We use the following notation:

$$\overline{M}.q.t = \begin{cases} M.0.(q+1), \text{if } q < 2^t \\ M.(q-2^t+1).(q+1), \text{otherwise} \end{cases}$$

The parameterized process $S.q$ may be now defined:

$$
\begin{array}{l}
S.q :: \\
|[\ t, aux : int \\
\quad t := 0; m := x(q); \\
\quad \{P.q\} \\
\quad do\ t \neq k \rightarrow \\
\quad\quad C0.q; \\
\quad\quad \{2^t \leq q < 2^k \Rightarrow aux = \overline{M}.(q-2^t).t\} \\
\quad\quad S0.q; \\
\quad\quad \{P.q\} \\
\quad od \\
]|
\end{array}
$$

Each iteration is formed of a communication process $C.q$, and a computation process $S.q$.

$$
\begin{array}{l}
C0.q :: \{\text{communication layer}\} \\
|[\{m = \overline{M}.q.t\} \\
\quad if\ q < 2^t \rightarrow \\
\quad\quad (q+2^t)!m \\
\quad [] \ 2^t \leq q < 2^k - 2^t \rightarrow \\
\quad\quad (q+2^t)!m; \\
\quad\quad (q-2^t)?aux; \\
\quad\quad \{aux = \overline{M}.(q-2^t).t\} \\
\quad [] \ 2^k - 2^t \leq q < 2^k \rightarrow \\
\quad\quad (q-2^t)?aux; \\
\quad\quad \{aux = \overline{M}.(q-2^t).t)\} \\
\quad fi \\
\quad barrier \\
]|
\end{array}
$$

The pairs of communication processes have the form $(q, q + 2^t), 0 \leq q < 2^k - 2^t$, where q is the sender and $q + 2^t$ is the receiver. The precondition of the sending statement is $m = \overline{M}.q.t$. The received value in process $q + 2^t$ is stored in the local variable aux and the postcondition is $aux = \overline{M}.((q + 2^t) - 2^t + 1).t = \overline{M}.q.t$. So, the rule of satisfaction is proved.

$S0.q ::$ {computation layer}
$|[\quad if\ 2^t \leq q \rightarrow$
$\quad\quad m := m \odot aux;$
$\quad fi$
$\quad ; t := t + 1$
$]|$

Cost evaluation

There are $k = \log p$ supersteps, each of them being formed of a computational process with $w_q = 2$ and a communication process with $h = 1$.
So the total cost is:

$$2 \log p + \log p * g + \log p * l$$

IV. Distributions

Data distributions have a serious impact on time complexity of parallel programs developed based on domain decomposition, which are very conveniently implemented using BSP model. Distributions are considered to be parameters of our programs, and they have to be carefully analyzed since they may considerably change the complexity of our programs.

We use $\overline{n}$ to denote the set $\{\forall i : 0 \leq i < n : i\}$.

Definition 1: $D = (\delta, A, B)$ is called a simple (one-dimensional) distribution if A is a finite set that specifies the set of data objects (an array with n elements that represent the indices of data objects), B is a finite set that specifies the set of processes, (which is usually $\overline{p}$) and, δ is a mapping from A to B.

Distributions of multi-dimensional arrays may be modeled by Cartesian distributions. In what follows, it is assumed that an $m \times n$ matrix is distributed across processes.

Definition 2: A Cartesian distribution is defined by a Cartesian product of one-dimensional distributions. The Cartesian product of two one-dimensional distributions $D0 = (\delta 0, \overline{m}, \overline{M})$, $D1 = (\delta 1, \overline{n}, \overline{N})$ is defined by:

$$D0 \times D1 = (\delta 0 \times \delta 1, \overline{m} \times \overline{n}, \overline{M} \times \overline{N})$$

where the function $\delta 0 \times \delta 1$ assigns a pair of process numbers to each array index pair.
Formally written, we have $\delta 0 \times \delta 1 = (\lambda i, j \cdot (\delta 0.i, \delta 1.j))$.

The Cartesian product of two one-dimensional distributions uses a process pair as identification for a process. Since the processes number p is set, we can consider all decomposition such that $p = M \times N$.

When the processes number is greater than the data input size, it is desirable to assign a datum to more than one process. Also, when a data object is used in more than one computation, this kind of distributions may lead to efficient algorithms [11].

Definition 3: A set distribution for n data input objects on p processes is defined by a set–valued mapping $\delta : \overline{n} \multimap \overline{p}$; $\delta.i$ represents the set of processes containing the data object with the index i.

By using the expression $q \in \delta.i$ we mean that the date with the index i was assigned to the process q; if we use simple distribution this expression will be equivalent to $q = \delta.i$.

V. Cost Evaluation

BSP model gives us a cost model that is both tractable and accurate, and can be used as a part of the design process. This formal method for BSP program construction allows us to formally evaluate the costs based on the local postconditions.

The distributions determine how the global postcondition is split up, and the local postconditions determine the number of communications and the computational work of each process. Given a program's postcondition and a distribution, an evaluation of the cost, before developing the program, is possible.

We consider that we have p processes and a data distribution $\delta : \overline{n} \rightarrow \overline{p}$. The program's postcondition is split up into p local postconditions according to the distribution δ. For each local postcondition a process, establishing it, is created and associated to.

The total number of postconditions that refer to a particular datum is a measure of the number of communications of that datum.

For the datum e, the quantity $NOcc.e$ is introduced:

$NOcc.e =$ the number of local postconditions in which e occurs.

For a BSP program, we are interested in finding the number h, and this depends on the fan_in and fan_out numbers of each process q. The fan_out number of process q may be computed based on $NOcc$ of each datum that is assigned to process q:

$$fan\_out_q = (\sum i : 0 \leq i < n \wedge q \in \delta.i : (NOcc.e_i - A.e_i)\,/|\delta.i|)$$

$$A.e = \left(\sum q : 0 \leq q < p \wedge q \in \delta.e : E.q.e\right)$$

where

$$E.q.e = \begin{cases} 1, & \text{if } e \text{ occurs in the postcondition} \\ & \text{of the process } q; \\ 0, & \text{otherwise.} \end{cases}$$

For simple distributions $|\delta.i| = 1$ since $\delta.i$ is a simple value; for set-distribution $\delta.i$ represents the set of processes containing the data object with the index i. The formula is based on the division with $|\delta.i|$ because if e_i belongs

to many processes, all of them could participate to the communication as 'senders' of this date.

The fan_in number of process q can also be evaluated by counting all the data that are used in the local postcondition $R.q$ of the process q, which are not assigned to the process q.

$$fan\_in_q = \Big(\sum i : 0 \leq i < n \wedge q \notin \delta.i \wedge e_i \text{ occurs in } R.q : 1\Big)$$

From these, we have $h_q = \max(fan\_in_q, fan\_out_q)$ (or we may consider the sum $h_q = fan\_in_q + fan\_out_q$), and then $h = (\max\ q : 0 \leq q < p : h_q)$ may be computed.

By summing over all data e, the total number of communications $NCom$ will be obtained from $NOcc.e$:

$$NCom = \Big(\sum e :: NOcc.e - A.e\Big)$$

If the distribution is well balanced – perfect or homogenous – and the postconditions defined based on these distributions use the same amount of data, we may evaluate h using the formula $h = NComm/p$.

This technique of counting communications allows a comparison of the distributions on the basis of their communication overhead. The value of h is only determined by the way the program's postcondition is split up, and the distribution used. We have considered that we need only one superstep for satisfying the postcondition. But, generally, in programs construction stepwise refinement is used, and we may apply this technique to partial postconditions. So, this technique can give us a fair approximation of the communication overhead. This technique also allows us to choose the most appropriate distribution before developing the program.

The technique suits very well to BSP programs since the results that can be obtained are independent of the intercommunication network.

Example 2: **Lagrange Polynomial**

We will make cost evaluation for BSP parallel programs for the computation of Lagrange polynomial on a given value. Two variants are constructed starting by selecting different types of distribution: simple and set.

The Problem

Let $[a,b] \subset \mathbb{R}, x(i) \in [a,b], 0 \leq i < m$, such that $x(i) \neq x(j)$ for $i \neq j$ and $f : [a,b] \to \mathbb{R}$.

Lagrange interpolation polynomial is defined as:

$$(L_{(m-1)}f).x = \Big(\sum i : 0 \leq i < m : l_i.x * f.x(i)\Big)$$

where $l_i, 0 \leq i < m$ are the fundamental Lagrange interpolation polynomials:

$$l_i.x = \frac{(x-x(0))\ldots(x-x(i-1))(x-x(i+1))\ldots(x-x(m-1))}{(x(i)-x(0))\ldots(x(i)-x(i-1))(x(i)-x(i+1))\ldots(x(i)-x(m-1))}$$
$$= \frac{u.x}{(x-x(i))} \cdot \frac{1}{(x(i)-x(0))\ldots(x(i)-x(i-1))(x(i)-x(i+1))\ldots(x(i)-x(m-1))}$$

The global postcondition for the computation of the value $L_{(m-1)}f.x$ is

$$R: \ lx = (L_{m-1}f).x$$

Variant 1 – Simple Distributions

We consider the one-dimensional distribution $\delta : \overline{m} \to \overline{p}$, for the data $x(i), f(i), 0 \leq i < m$. The x value is distributed to all the processes (or x is distributed to process 0, and then communicated to all other processes, by a broadcast).

Using stepwise refinement, the following stages can be considered:

1) compute the value $u.x$;
2) compute the fundamental polynomials $l_i.x$;
3) compute the value $(L_{m-1}f).x$.

The local postconditions for the three stages are:

$$\begin{aligned} R0.q: &\ ux = u.x \wedge \\ &\ (\forall i : 0 \leq i < m \wedge \delta.i = q : xx(i) = x - x(i)) \\ R1.q: &\ (\forall i : 0 \leq i < m \wedge \delta.i = q : l(i) = l_i.x) \\ R2.q: &\ lx = (L_{m-1}f).x \end{aligned}$$

The first and the last stages represent computations of a product and a sum, so classic algorithms for combine computations (possibly using tree-like computations), may be used.

A BSP program for computing a general sum(product) of n elements may be defined using a single superstep with the cost:

$$l + n - 1 + g(n-1)$$

or using $(\log n)$ supersteps based on a tree-like computation with a cost:

$$\log n(l + 1 + 2g)$$

We will focus next on the second postcondition. The postcondition for the second stage may be rewritten in the following way:

$$R1.q: \ (\forall i : 0 \leq i < m \wedge \delta.i = q : l(i) = \frac{ux}{xx(i)} * prod.i.m)$$

where $prod.i.k = (\prod j : 0 \leq j < k \wedge j \neq i : x(i) - x(j))$.

Cost evaluation

$$\begin{aligned} &fan\_out_q = (\sum i : 0 \leq i < n \wedge q \in \delta.i : NOcc.x(i) - 1) \\ &= \frac{m}{p}(p-1) \\ &fan\_in_q = m - \frac{m}{p} = \frac{m}{p}(p-1) \end{aligned}$$

So, we may evaluate the cost as

$$S = l + \frac{m}{p}(m-1) + mg$$

Variant 2 – Set Distribution

Let $p = M \times M$ and all the processes are identified by a pair $(s,t), 0 \leq s,t < M$.

If we have a simple distribution $\delta : \overline{m} \to \overline{M}$, and M permutations $\pi_t : \overline{M} \to \overline{M}, 0 \leq t < M$, (for example $\pi_t.i = (i+t)\%M$), then we may define a set-distribution by:

$$x(i) \in O.s.t \Leftrightarrow \delta.i = \pi_t.s$$

where $O.s.t$ is the set of data elements that are assigned to the process (s,t).

For $m = 9$ and $M = 3$ the data distribution is shown in the Figure 1.

There are again three stages defined by the following local postconditions:

$$R0.s.t: \ (ux = u.x \wedge (\forall i: \quad i \in O.s.t: xx(i) = x - x(i)))$$
$$R1.s.t: \ t \neq 0 \vee (\forall i: i \in O.s.t: l(i) = l_i.x)$$
$$R2.s.t: \ (lx = (L_{m-1}f).x)$$

s\t	0	1	2
0	x_0, x_3, x_6	x_1, x_4, x_7	x_2, x_5, x_8
1	x_1, x_4, x_7	x_2, x_5, x_8	x_0, x_3, x_6
2	x_2, x_5, x_8	x_0, x_3, x_6	x_1, x_4, x_7

Figure 1. The data distribution for $m = 9, M = 3$.

Each row $(s, .)$ computes the values $l(i), \forall i: 0 \leq i < m \wedge \delta.i = s$.

We will focus again on the second stage, and we will split it into two steps: one for partial computations, and one for combining the partial computations.

The postcondition $R1.s.t$ may be rewritten as:

$$R1.s.t: t \neq 0 \vee (\forall i: i \in O.s.t: \\ l(i) = ux/xx(i) * 1/prod.i.m)$$

where $prod.i.m = (\prod j: 0 \leq j < m \wedge i \neq j: (x(i) - x(j)))$.

To compute the products $prod.i.m$ we split them in M products. Each of these subproducts corresponds to the set of elements assigned to a process.

So, we rewrite the products $prod.i.m$ as:

$$\begin{aligned} & prod.i.m \\ = & \{\text{range spliting } ; s = \delta.i\} \\ & (\prod t: 0 \leq t < M: (\prod j: j \in O.s.t \wedge i \neq j: (x(i) - x(j)))) \\ = & \{parprod.i.t \stackrel{not}{=} (\prod j: j \in O.s.t \wedge i \neq j: (x(i) - x(j)))\} \\ & (\prod t: 0 \leq t < M: parprod.i.t) \end{aligned}$$

Cost evaluation

For the fist step we have:

$$\begin{aligned} fan\_out_{(s,t)} &= (\sum i: i \in O.s.t: (NOcc.x_i - A.x_i)/M) \\ &= \frac{m}{M}(2M - 1 - M)/M \approx \frac{m}{M} \\ fan\_in_{(s,t)} &= \frac{m}{M} \end{aligned}$$

The costs of the first and the second supersteps are:

$$C1 = l + \frac{m}{M}\left(\frac{m}{M} - 1\right) + \frac{m}{M}g$$
$$C2 = l + \frac{m}{M}(M-1) + \frac{m}{M}(M-1)g$$

and, the total cost is:

$$2l + \frac{m}{M}\left(\frac{m}{M} + M - 2\right) + mg$$

Remarks:

- Different types of distributions lead to different algorithms.
- The second algorithm can be used in both cases: $p \leq m$, or $p > m, p = M \times M$. If $p > m$ the advantage of using the second algorithm is obvious.
- If l the cost for synchronization is big then the first variant is better.
- If we consider $h_q = fan\_in_q + fan\_out_q$, then the cost of the first variant increases considerably

$$S = l + \frac{m}{p}(m-1) + 2mg$$

and very probably $mg > l$, and so the second variant becomes better.

VI. Conclusions

BSP has shown that structured parallel programming is not only a performance win, but it is also a program construction win, especially if we add a formal method for designing.

The method presented in this paper uses parameterized processes and parameterized assertions on local variables. In this way the process construction becomes very simple. Parameterized invariants are constructed from postconditions, for program derivation.

BSP model proved to be very appropriate for problems with regular structure, and so, for problems based on domain decomposition. An important advantage of this formal method of BSP programs construction is that we can take into account the data-distribution, even at the beginning of the construction process. The possibility of evaluating costs from postconditions, allows us to make a cost evaluation even at the early stages of the design, at this leads us to the right decisions.

Acknoledgement: This work was supported by CNCSIS - UEFISCDI, project number PNII - IDEI 2286/2008

References

[1] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.

[2] Rob H. Bisseling. *Parallel Scientific Computation -A Structured Approach using BSP and MPI* Oxford University Press, 326 pg., 2004.

[3] Yifeng Chen, Jeff W. Sanders: Top-Down Design of Bulk-Synchronous Parallel Programs. Parallel Processing Letters 13(3): 389-400, 2003.

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

[5] C.A.R. Hoare. *An Axiomatic Basis for Computer Programming*. Communications of the ACM, 12(10):576-580, 1969.

[6] C.A.R. Hoare, J. He. *Unified Theories of Programming*. Prentice-Hall International, 1998.

[7] J. Fortin, F. Gava. *BSP-WHY: an intermediate language for deductive verification of BSP programs*, in Proceedings of the 4th international workshop on High-level parallel programming and applications, HLPP '10, ACM, Baltimore, Maryland, USA, pp. 35–44, 2010.

[8] G. M. Levin, D. Gries. *A Proof Technique for Communicating Sequential Processes*. Acta Informatica (15):281-302, 1981.

[9] D. Lecomber. *Methods of BSP Programming*. PhD Thesis, Oxford University Press, 1998.

[10] C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[11] V. Niculescu. *On Data Distribution in the Construction of Parallel Programs*. The Journal of Supercomputing, Kluwer Academic Publishers, 29(1): 5-25, July 2004.

[12] D.B. Skillicorn. *Building BSP Programs Using the Refinement Calculus*. In Formal Methods for Parallel Programming and Applications, IPPS/SPDP'98, volume 1388 of LNCS, pp. 790-795, 1998.

[13] D.B. Skillicorn, D. Talia. *Models and Languages for Parallel Computation*. ACM Computer surveys, 30(2): 123-136, June 1998.

[14] A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Programs*. PhD. Thesis, University of Oxford, 1998.

[15] L.G. Valiant. *A Bridging Model for Parallel Computation*. Communication of the ACM, 33(8): 103-111, August 1990.