An approach to correctness of data parallel algorithms

Joaquim Gabarró * Ricard Gavaldà *

Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Pau Gargallo 5, 08028 Barcelona
Spain

gabarro@lsi.upc.es gavalda@lsi.upc.es

* Research supported by the ESPRIT Basic Research Actions Program of the EC under contract No. 7141 (project ALCOM II).

Proposed running head: Correctness of data parallel algorithms.

Contact author:

Joaquim Gabarró
Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Pau Gargallo 5, 08028 Barcelona
Spain

Phone number: 34-3-401-7011
E-mail: gabarro@lsi.upc.es

Abstract: The design of data parallel algorithms for fine-grain machines is a fundamental domain in today's computer science. High standards in the specification and resolution of problems have been achieved in the sequential case. It seems reasonable to apply the same level of quality to data parallel programs.

It appears that most data parallel problems can be specified in terms of post and preconditions. These conditions characterize the overall state of the fine-grain processors in the initial and final states. In this paper:

- We present an axiomatic system to prove correctness of data parallel algorithms on single-instruction multiple-data (SIMD) machines.
- We specify some data parallel problems like tree sum, root finding, radix sorting, and dynamic memory allocation.
- With this set of axioms we prove the correctness of programs solving the problems above.

It seems that the framework to deal with data parallel problems is quite different from those dealing with problems of parallelism with multiple threads of control, like those solvable in Communicating Sequencial Processes (CSP).

1. Introduction

The study of massively parallel algorithms implemented in fine-grain machines is a fundamental domain in today's computer science. Most of the theoretical work on algorithms has used the model called Parallel Random Access Machine or PRAM [11,26], and a lot of effort has been done to find optimal and efficient PRAM algorithms to solve different problems [12,19]. However much less effort has been directed towards having a good programming methodology. This can be dangerous if we look back at what happened in the sequential case [27].

Building the Connection Machine, Hillis [15] seems to have fixed a new paradigm in programming fine-grain machines. On the other hand, active data structures [16] seem to be also an important and unexplored tool. In order to have a clear, usable, and complete program methodology for PRAM algorithms it seems unavoidable to:

1) Consider what kind of high-level instructions are needed to construct high-level programs on fine-grain machines.
2) Give an axiomatic semantics for these instructions.
3) Develop the notion of data structure adequate to this kind of machines.

This paper intends to be a first step towards points 1) and 2). Notations for synchronous parallel algorithms can be divided roughly into two groups, depending on how they describe interprocessor comunication. One is the message-passing scheme, where the language contains explicit instructions of the type **send** and **receive**. Another one is the shared-memory scheme, where it is assumed that each processor can access any variable in memory. The second scheme seems to describe computation at a higher level, and has both theoretical and practical importance: On the one hand, it has a very good theoretical model, the PRAM. On the other hand, it is the basis for many existing parallel languages (FORTRAN 90, C*). We adopt the shared-memory view for our high-level notation. Also, we focus on single-instruction, multiple data (or SIMD) algorithms; this means that there is a single instruction flow that is executed by all processors on different data items. Let us be more precise about our goals: We want to define formally a set of instructions that can express most of the SIMD algorithms described, for instance, in [12,19]. The definition should make no reference to the PRAM model or any other low-level model. And, more important, the axiomatic system must be "practical" in the sense that it can be used to prove correctness of usual data parallel algorithms.

Let us also make clear which are not our goals:

We do not claim that our instructions are sufficient for general-purpose programming. We do not discuss input/output, data typing, and other features that any practical programming language should have.

We do not claim either that the instructions can be efficiently implemented with current technology. For instance, our high-level view ignores crucial issues such as:

- Mapping between data structures and processors: Similarly, when describing formally high-level sequential languages, it is not reasonable to explain how global and local variables, procedure parameters, and dynamic memory have to be mapped into a machine's memory, that is usually a linear array.

- Scheduling of tasks to individual processors: A typical consideration that we want to exclude from our semantics is whether the parallel assignment

$$x[k+1] := x[k] + x[k-1]$$

is executed by the processor "numbered" k, $k+1$, or $k-1$, be this number physical or virtual.

Of course, ignoring these items is unrealistic with present machines and compiler technology. However, by comparison with the history of sequential programming, we expect that these problems will be less and less important in the far future. Otherwise, there is little hope of ever having high-level, machine-independent, parallel software.

Let us consider some related work. Our approach to synchronous parallelism is founded on a high level view of the PRAM as a set of active processors plus a global memory. Our specifications and axioms will be obtained keeping this idea in mind. However there are other approaches and, in principle, any one of them can be translated into an axiomatic system.

- When the model of computation is the array machine [21] it is possible to deal with arrays as whole objects. This approach has been used by Stewart [24] and Clint and Narayana [7].
- Another approach makes explicit that processors communicate through an interconnection network. In this view the statements **send** and **receive** become fundamental. The works of Bougé and Levaire [3] and Utard [25] take this approach.

These works have common points and differences. However we think that our axiomatic system is workable tool. To test it we specify and prove a variety of basic algorithms such as tree sum, root finding, radix sorting, and dynamic memory allocation.

Finally let us remark that there are works on denotational semantics of the PRAM model. Rünger and Sieber [22] take a trace-based approach [18] for the FORK language defined by Hagerup, Schmitt, and Seidl [14].

2. Preliminaries

In order to present the semantics of a SIMD language we start with an informal overview of it. It is very close to that used in [16]. We describe an abstract parallel machine that supports it and how each language construct is implemented. We do this for clarity; in the following sections, we will give a semantics not tied to any machine model.

We want to describe programs written for a machine with a large number of processor+memory cells. Every processor has an internal state bit, called the *context flag*. All processors receive a common instruction stream from a host and examine it synchronously. For each of them, the context flag determines whether the incoming instruction is actually executed or ignored. There is also a set of instructions that are executed unconditionally, i.e., even if the context flag is off. Such instructions are needed, at least, for setting and clearing context flags. We call a processor *selected* or *active* at a given moment if its context flag is on. The *context* at that moment is the set of selected processors. Note again that the next instruction in the stream will be executed identically by all processors in the context, while the rest of processors will remain idle.

Two classes of variables are considered: *Global* variables are not attached to any particular processor. One can think of them as located in the host's memory. *Local* variables are replicated in each processor. We use array notation to define and use local variables. A declaration such as

$$x : \textbf{array } [1..N] \textbf{ of proc of } \text{integer}$$

creates an integer variable belonging to each processor k, that can be referenced as $x[k]$. But note that memory is shared, in the sense that a processor can directly access another processor's memory.

Language constructs

- The main parallel construct is the for-all, denoted as

$$\textbf{for all } k \textbf{ do in parallel } S(k) \textbf{ end}$$

It indicates that statement $S(k)$ will be executed by all active processors, each with a different value of variable k. We will also use the construction

$$\textbf{for all } k : E(k) \textbf{ do in parallel } S(k) \textbf{ end}$$

to indicate that only active processors k such that condition $E(k)$ is true will actually execute $S(k)$. This statement is used for notational convenience. It can be viewed as an unrestricted for-all enclosing an if statement (described below).

- Inside a for-all we can have parallel assignments of the form $x[F_1(k)] := F_2(k)$ giving statements like

$$\textbf{for all } k : E(k) \textbf{ do in parallel } x[F_1(k)] := F_2(k) \textbf{ end}$$

This statement instructs each selected processor k to compute $F_2(k)$ and update variable x of processor $F_1(k)$. This requires some communication if $F_1(k) \neq k$. Note that expression $F_2(k)$ can be in fact independent of k (for example, it can be a global variable).

- We also have a conditional statement of the form

$$\textbf{if } E(k) \textbf{ then } S(k) \textbf{ end}$$

It indicates that all currently selected processors evaluate $E(k)$. Those that evaluate to true will execute $S(k)$ and all others are deselected during that time.

But remember that the instruction stream is still sent to all processors in the machine. This is necessary because S might contain some unconditional instructions that require activating all of them.

• Instructions that are executed unconditionally are indicated as

$$\textbf{unconditionally } S \textbf{ end}$$

Here all processors are temporarily selected for execution of S and after termination the original context is restored.

• We use two functions called **count** and **enumerate** as primitives of our language. If c is a global variable and x is a local variable, then the statement

$$c := \textbf{count}$$

will place in c the number of elements in the current context, and

$$x[k] := \textbf{enumerate}$$

will place in variable x of each selected processor its relative position in the context. That is, the selected processor with lowest value of k will receive 0, the immediate next will receive 1, etc. Hilis and Steele [16] and Blelloch [2] argue that these operations can be regarded as primitive because they can be implemented very efficiently on real machines and allow to write many algorithms concisely. We will support this opinion by showing that they have very simple and elegant definitions in our semantics.

• Finally, we inherit all constructs common in sequential programming, such as assignment to global variables, sequential composition (denoted **;**), and sequential **for** and **while** loops.

With all these constructs we can code algorithms easily, especially if we can use procedures and functions. In this paper we deal with procedures informally. However, their axioms can be translated directly from those of sequential languages.

As an example of our notation, Figure 1 presents a program that adds two N-bit numbers in $O(\log N)$ parallel steps. It is based on the "carry generate/carry propagate" algorithm as described in [19].

Notation

We introduce some notations to deal with finite sets of integers. Let A be a subset of $\mathbb{N}$ such that $A = \{a_0, a_1, a_2, \ldots\}$. We assume A ordered, that is, $a_0 < a_1 < a_2 < \ldots$ Let $\#A$ be the cardinality of A. Given $k \geq 0$, we define the following subsets of A:

$$A_{<k} = \{a \in A \mid a < k\}$$

$$\mathit{first}_k A = \begin{cases} \{a_0, \ldots, a_{k-1}\} & \text{if } k < \#A \\ A & \text{otherwise.} \end{cases}$$

Given a mapping $y : A \to A'$ and two sets $B \subseteq A$ and $B' \subseteq A'$ we denote as $B \xrightarrow{y} B'$ the restriction of this mapping to the sets B and B'. Note that given two sets A and B the following conditions are equivalent

- $\mathit{first}_{min(\#A,\#B)} A \xrightarrow{y} \mathit{first}_{min(\#A,\#B)} B$ is a bijection.
- $\mathit{first}_{\#B} A \xrightarrow{y} \mathit{first}_{\#A} B$ is a bijection.

type number = **array** $[N-1..0]$ **of proc of** bit;
procedure Fast_Binary_Sum (a, b: number): **array** $[N..0]$ **of proc of** bit;

import
Tree_And (a: number): bit;
Tree_Or (b: number): bit;

var
p: number; { carry propagate }
g: number; { carry generate }
c: number; { carry }
x: **array** $[N-1..0, N-1..0]$ **of proc of** bit;

begin
{ compute the carry }
for all $i : N > i \geq 0$ **do in parallel** $p[i] := a[i] \oplus b[i]; g[i] := a[i] \wedge b[i]$ **end**;
for all $i, j : N > i > j \geq 0$ **do in parallel** $x[i, j] :=$ Tree_And $(p[i..j+1]) \wedge g[j]$ **end**;
for all $i : N > i \geq 0$ **do in parallel** $c[i] := g[i] \vee$ Tree_Or $(x[i, i-1..0])$ **end**;
{ compute the sum }
Fast_Binary_Sum$[0] := a[0] \oplus b[0]$;
for all $i : N > i > 0$ **do in parallel** Fast_Binary_Sum$[i] := a[i] \oplus b[i] \oplus c[i-1]$ **end**;
Fast_Binary_Sum$[N] := c[N-1]$
end Fast_Binary_Sum

Figure 1. Fast binary sum.

3. Semantics

In this section we present an axiomatic semantics, based on the notion of state and pre- and post- conditions for the language described. This extends the work done by Hoare [17] and Dijkstra [9] for sequential programming.

To see why this type of semantics is natural in this framework, recall that we assume a single instruction stream that is executed in perfect synchrony by all processors. This gives us two immediate advantages at the time of designing and proving programs correct:

• First, there is no need to synchronize processors explicitly. In particular, there is no danger of deadlock: the state of each processor (selected or not) is internal to it and can only be changed by the instruction stream, not by any other processor's signal. Thus, we can prove termination of these programs using the same techniques as in sequential programming.

• Second, programs are completely deterministic. There are no invisible state changes or nondeterministic choices. As a consequence, there is a precise definition of the "state" of a program at a given moment.

For these reasons, it is possible to specify the behaviour of one of these programs *completely* in terms of its initial and final state. Note that such specification is not possible for concurrent MIMD programs, because one must also specify how and when each program is ready to communicate.

States, processors, and assertions

In the sequential case the state of a computation is determined by the value of the variables. The relationship between these values is described by a predicate P. To describe the state of a SIMD algorithm we need to take care of two different facts.

• As in the sequential case we need to describe the relationship between the values of the local and global variables. This can be done by a predicate P.

• Recall that each processor can be active or inactive. Any description of the state of a program must take care of this, because the same program, started with different active processors, will have different results. We assume that processors are denoted by natural numbers; our axioms can be easily generalized if any other countable type is used. The set of active processors will be described by a set expression A.

Summarizing these two aspects, our assertions about states will be a pair of the form $\{A;\ P\}$ where A is a set of natural numbers and P is a predicate describing the relationship between the values of all program variables.

One can argue that the set of active processors A can be regarded as just another variable and handled by the predicate P. We will not adopt this view for two reasons:

- First, some axioms are much simpler if the set A is displayed clearly in each assertion and not hidden inside P.
- Second, processors should be selected or deselected on entry or exit of the block-structured statements only. In this sense, changing context flags explicitly is the parallel analog of the sequential **goto** statement. In particular, no program expression or assignment should use the context flags as variables.

Therefore, our specification of a program S will have the form

$$\{\ A;\ P\ \}\ \ S\ \ \{\ A;\ Q\ \}$$

meaning that S transforms predicate P into predicate Q if it is started with processors in A active. Note that with our program constructs the set of active processors is always the same before and after S. However, it can change during the execution of S. For example, S can contain **if** statements that deselect some processors temporarily.

We note an additional problem here. If S modifies variables used in the set expression A, it is possible that A denotes different sets before and after S. All proof rules that we give assume that *S does not modify variables used in A*. This assumption simplifies our axioms and does not reduce their power at all: If this problem appears, modify the specification above to

$$\{\ B;\ (B = A) \wedge P\ \}\ \ S\ \ \{\ B;\ Q\ \}$$

where B is a new symbol that "fixes" the initial set of active processors.

Axioms

Global Assignment. The axiom for assignment to a global variable is identical to that of the sequential case. If c is a global variable and E a global expression, then:

$$\{\ A;\ P\ \}\ c := E\ \{\ A;\ Q\ \}$$

iff P implies $domain(E) \wedge Q^c_E$. Here, Q^c_E denotes the predicate that results of the *textual substitution* of c by E in Q; see [13] for details on textual substitution. In the following axioms, conditions such as $domain(E)$ will not appear explicitly in order to lighten the notation.

Local Assignment. We present two axioms for assignment to local variables that correspond to assignment in Owner Write and Exclusive Write modes. The first axiom is a particular case of the second. We include both because parallel programs usually contain many assignments in Owner Write and we can then use a considerably simpler proof rule. Let x be a local variable. We consider that x contains a function of finite domain, so each local instance of x, $x[k]$, is in fact the application of the function in x to argument k. Thus, we can view assignments to components of x as a change of the function x.

• *Owner Write.* Let $E(k)$ be an expression that may depend on k. Let us give an axiom to formalize the idea "every active processor k does some local computation (represented by $E(k)$) and memorizes the result on its local variable $x[k]$". We call this kind of assignment Owner Write assignment. If we consider the innermost for-all statement enclosing an Owner Write assignment $x[k] := E(k)$ we have the following axiom.

$$\textbf{for all } k \textbf{ do in parallel} \cdots \{\ A;\ P\ \}\ x[k] := E(k)\ \{\ A;\ Q\ \} \cdots \textbf{end}$$

iff P implies $Q^x_{(x,A,E)}$ and (x, A, E) is the array (or function) defined as

$$(x, A, E)[k] = \begin{cases} E(k) & \text{if } k \in A \\ x[k] & \text{otherwise} \end{cases}$$

Remark that (x, A, E) describes the global modification of x and $Q^x_{(x,A,E)}$ describes the global modification of the state described by Q.

• *Exclusive Write.* The axiom for Exclusive Write assignment is slightly more complicated because it must deal with indirection between processors as in $x[F(k)] := E(k)$. If $F(k)$ and $E(k)$ are expressions that may depend on k and, as we want exclusive writes, $F(k)$ must be injective on the domain of k. After executing the assignment, we have for all l and k that $x[l] = E(k)$, if $F(k) = l$ and k is active. The processor k can be identified as $k = F^{-1}(l)$ where $F^{-1}(j) = \{k\ :\ F(k) = j\}$, so the expression $E(F^{-1}(l))$ makes sense here. Then, the assignment $x[F(k)] := E(k)$ is equivalent to $x[l] = E(F^{-1}(l))$. As k and l are dummy variables we get the following axiom

$$\textbf{for all } \ k \textbf{ do in parallel} \cdots \{\ A;\ P\ \}\ x[F(k)] := E(k)\ \{\ A;\ Q\ \} \cdots \textbf{end}$$

iff P implies

$$(F(k) \text{ as a function of } k \text{ is injective}) \ \wedge \ (Q^x_{(x,A,E,F)}).$$

Here, (x, A, E, F) is the function that is equal to x in each component that is not altered, and that has value $E(k)$ at the component altered by processor k:

$$(x, A, E, F)[k] = \begin{cases} E(F^{-1}(k)) & \text{if } F^{-1}(k) \in A \\ x[k] & \text{otherwise} \end{cases}$$

We can give a slightly stronger axiom noticing the following: to have Exclusive Write, it is enough to prove that F is injective on the domain A. Then, $F^{-1}(k) \cap A$ has at most one element even if $F^{-1}(k)$ is larger, and the kth component of x receives the value $E(F^{-1}(k) \cap A)$ if $F^{-1}(k) \cap A \neq \emptyset$.

Counting active processors. This instruction assigns to a global variable c the number of currently active processors. Thus, we have an axiom similar to that of global assignment. If $\#A$ denotes the cardinality of set A, then

$$\{\ A;\ P\ \}\ c := \textbf{count}\ \{\ A;\ Q\ \}$$

iff P implies $Q^c_{\#A}$.

Enumerating active processors. This instruction enumerates from 0 onwards the currently active processors. Each active processor receives its relative position into its own copy of a local variable. Thus, the axiom for **enumerate** is similar to local assignment in Owner Write. For every index i and a set A we define $\#A_{<i} = \#\{j \in A \mid j < i\}$. Then $\#A_{<i}$ is 0 if i is the minimum of set A, 1 if it is the second element in A, and so on. The axiom is:

$$\{\ A;\ P\ \}\ x[k] := \textbf{enumerate}\ \{\ A;\ Q\ \}$$

iff P implies $Q^x_{(x,A,\#A_<)}$. In this predicate, $\#A_<$ is interpreted as an expression that depends on one argument.

Sequential composition. The axiom for sequential composition is identical to that of the sequential case.

$$\{\ A;\ P\ \}\ S_1; S_2\ \{\ A;\ Q\ \}$$

iff there exists a predicate R such that

$$\{\ A;\ P\ \}\ S_1\ \{\ A;\ R\ \} \wedge \{\ A;\ R\ \}\ S_2\ \{\ A;\ Q\ \}$$

Conditional statement. The following holds

$$\{\ A;\ P\ \}\ \textbf{if } E(k) \textbf{ then } S(k) \textbf{ end}\ \{\ A;\ Q\ \}$$

iff

$$\{\ \{k \mid E(k)\ \wedge\ k \in A\ \};\ P\ \}\ S(k)\ \{\ \{k \mid E(k)\ \wedge\ k \in A\ \};\ Q\ \}$$

When the expression $E(k)$ does not depend on k the axiom for sequential conditional statement can be derived.

Unconditional statement. The following holds

$$\{\ A;\ P\ \}\ \textbf{unconditionally}\ S\ \textbf{end}\ \{\ A;\ Q\ \}$$

iff

$$\{\ \mathcal{D};\ P\ \}\ S\ \{\ \mathcal{D};\ Q\ \}$$

where $\mathcal{D}$ denotes the domain of processors of the program.

For-all statement. Let us consider how a for-all statement affects the state of a computation. If the following holds

$$\textbf{for all}\ k\ \textbf{do in parallel}\ \{\ A;\ P\ \}\ S(k)\ \{\ A;\ Q\ \}\ \textbf{end}$$

it also holds

$$\{\ A;\ P\ \}\ \textbf{for all}\ k\ \textbf{do in parallel}\ S(k)\ \textbf{end}\ \{\ A;\ Q\ \}$$

It seems that, in general, the for-all instruction can be considered redundant. It only declares the variable that will be used to index parallel constructs.

From another point of view the statement **for all** k **do in parallel** $S(k)$ **end** is often intuitively interpreted as "every active processor k does some easy local computation. This computation does not interfere with the computation of its neighbours". In these cases, the postcondition usually has the form $\forall k : k \in A \Rightarrow Q(k)$, and we can give a useful rule.

Given u and v in A such that $u \neq v$, the statements $S(u)$ and $S(v)$ *do not interact* if they act over different sets of variables, this means:

- $S(u)$ cannot read from any variable written by $S(v)$, and reciprocally,
- $S(u)$ and $S(v)$ cannot write into the same variable, but
- $S(u)$ and $S(v)$ can read from the same variable.

Then, the following holds

$$\{A;\ \forall\ k : k \in A \Rightarrow P(k)\}\ \textbf{for all}\ k\ \textbf{do in parallel}\ S(k)\ \textbf{end}\ \{A;\ \forall\ k : k \in A \Rightarrow Q(k)\}$$

if

$$\begin{gathered} \forall\ k : k \in A \Rightarrow \{A; P(k)\}\ S(k)\ \{A; Q(k)\} \\ \wedge \\ \forall\ u, v : u \neq v \wedge u, v \in A :\ S(u) \text{ and } S(v) \text{ do not interact} \end{gathered}$$

We would like to remark that, in some cases, programming with for-all statements having only non-interacting $S(k)$ can be a serious handicap. In these cases the previous rule is useless.

To see what kind of computation is possible with no interaction, suppose that a program has the following form, where the different $S(k)$ do not interact:

$$\textbf{for all } k \in \{1, \ldots, l\} \textbf{ do in parallel } S(k) \textbf{ end}$$

Then, any arbitrary interleaving of the statements $S(1), \ldots, S(l)$ will give us a correct computation. In CSP notation [18] we could characterize the meaning of the for-all as:

$$S(1) \| S(2) \| \ldots \| S(l)$$

In particular the sequential computation $S(1); S(2); \ldots; S(l)$ gives us a correct result.

4. Examples

As examples of use of our axioms, we present now four programs of increasing complexity:

1) Computing the sum of the elements of an array (Figure 2).
2) Finding the root of a real function (Figure 3).
3) Radix sorting an array of integers (Figure 4).
4) Dynamic allocation of processors and memory (Figure 5).

The programs for 1), 3), and 4) are taken directly from [16]. We have only added variable declarations and split long programs into procedures for clarity. Basic ideas for 2) are taken from [1].

Example 1: Tree Sum of N numbers

Program Tree_Sum in Figure 2 computes the sum of all components of array x in time $O(\log N)$ by the well known binary-tree technique. This program illustrates the use of Owner Write assignment within for-all's, and is an example of small program where considering mainly "local states" is still feasible. Let $A = [0 \,..\, N)$ be the initial set of active processors. The program can be specified as:

$$\{A;\ (x[0], \ldots, x[N-1]) = (X_0, \ldots, X_{N-1})\ \}$$
$$\text{Tree\_Sum}$$
$$\{A;\ x[N-1] = \textstyle\sum_{i=0}^{N-1} X_i\}$$

To specify the procedure Partial_Sums it is interesting to consider the set

$$active(j) = \{k \in A\ :\ (k+1) \textbf{ mod } 2^j = 0\ \}$$

such that

- $active(j) \subseteq active(j-1)$.
- $k \in active(j) \Longrightarrow k \in active(j-1)\ \wedge\ k + 2^{j-1} \in active(j-1)$.

```
procedure Tree_Sum (x : array [0..N − 1] of integer): integer;
var j: integer;
procedure Partial_Sums (i: integer);
    for all k : (k + 1) mod 2^i = 0 do in parallel
        x[k] := x[k − 2^(i−1)] + x[k]
    end
end Partial_Sums;
begin
    for j := 1 to log_2 N do
        Partial_Sums(j)
    end;
    Tree_Sum := x[N − 1]
end Tree_Sum
```

Figure 2. Sum of N numbers.

With this set let us define the invariant of the sequential for:

$$SUMS(j) \equiv (\forall\ k:\ k \in active(j)\ :\ x[k] = \sum_{i=k-2^j+1}^{k} X_i)$$

Let us prove the following:

Lemma 1: The procedure Partial_Sums verifies for $j > 0$:

$$\{A;\ SUMS(j-1)\ \}$$
Partial_Sums (j)
$$\{A;\ SUMS(j)\ \}$$

Proof. Substituting the call Partial_Sums by its body we have:

```
{A; SUMS(j − 1) }
for all k : (k + 1) mod 2^j = 0 do in parallel
    x[k] := x[k − 2^(j−1)] + x[k]
end
{A; SUMS(j) }
```

Recalling that we view restricted for-all's as a composition of one for-all and one if statement, we must show that:

for all k : $(k+1)$ **mod** $2^j = 0$ **do in parallel**
  $\{active(j);\ SUMS(j-1)\ \}$
  $x[k] := x[k-2^{j-1}] + x[k]$
  $\{active(j);\ SUMS(j)\ \}$
end

We call $e(k)$ the expression $x[k-2^{j-1}] + x[k]$ that depends on k. Then

$$(x, active(j), e)[k] = \begin{cases} x[k-2^{j-1}] + x[k] & \text{if } k \in active(j) \\ x[k] & \text{otherwise.} \end{cases}$$

By the axiom of owner write assignment we must show that

$$SUMS(j-1) \Longrightarrow SUMS(j)^x_{(x,active(j),e)}$$

As the predicate $SUMS(j)^x_{(x,active(j),e)}$ is equivalent to

$$\forall\ k:\ k \in active(j)\ :\ x[k-2^{j-1}] + x[k] = \sum_{k-2^j+1}^{k} X_i$$

we must prove

$$SUMS(j-1) \Longrightarrow (k \in active(j) \Longrightarrow x[k-2^{j-1}] + x[k] = \sum_{k-2^j+1}^{k} X_i).$$

Let $k \in active(j)$. Then $k \in active(j-1)$ and $k - 2^{j-1} \in active(j-1)$. Assuming $SUMS(j-1)$ we have

$$k \in active(j-1) \Longrightarrow x[k] = \sum_{i=k-2^{j-1}+1}^{k} X_i$$

$$k - 2^{j-1} \in active(j-1) \Longrightarrow x[k-2^{j-1}] = \sum_{i=k-2^j+1}^{k-2^{j-1}} X_i$$

and this implies $SUMS(j)^x_{(x,active(j),e)}$. The proof is done. $\square$

Lemma 2: The program Tree_Sum satisfies its specification.

Proof. As

$$\begin{aligned} SUMS(0) &\equiv ((x[0], \ldots, x[N-1]) = (X_0, \ldots, X_{N-1})) \\ SUMS(\log_2 N) &\equiv \left(x[N-1] = \sum_{i=0}^{N-1} X_i \right) \end{aligned}$$

we must prove

$$\{A;\ SUMS(0)\ \}$$
$$\text{Tree\_Sum}$$
$$\{A;\ SUMS(\log_2 N)\ \}.$$

Substituting Tree_Sum by its body we must prove

$\{A;\ SUMS(0)\ \}$
for $j := 1$ **to** $\log_2 N$ **do**
    Partial_Sums (j)
end
$\{A;\ SUMS(\log_2 N)\ \}$

But this is done by the previous lemma and the well-known axiom for sequential for. □

Example 2: Finding a root of a real function

Figure 3 presents a simple algorithm to approximate roots of real functions. Assume that we have a continuous function f such as

function f (x: real): real;

Let ϵ be a fixed positive constant. Given two points a and b where f has different signs, procedure Root_ϵ should return a point between a and b whose distance is at most ϵ from a root of f. The algorithm approximates the root using a generalization of the bisection algorithm. Repeatedly, it splits the interval $[a, b]$ into N subintervals. It evaluates the endpoints of all these intervals, identifies those that can contain a root of f and arbitrarily selects one of them to go on. To specify formally this procedure, let us define the following predicates for real numbers x and x':

$$HAS\_ROOT\ (x, x') \quad \equiv \quad (x < x') \wedge (f(x) * f(x') \leq 0).$$

The continuity of f and $HAS\_ROOT\ (x, x')$ implies that f has at least one zero in the interval $[x, x']$, but it may have more than one. We define the set of active processors as $X = [0\ ..\ N]$ with $N \geq 2$. Remark that we have at least three active processors. Now we can specify Root_ϵ as follows:

$$\{\ X;\ (a = A) \wedge (b = B) \wedge HAS\_ROOT\ (a, b) \wedge (N \geq 2)\ \}$$

```
procedure Root_ε (a, b: real; f: real_function; N: integer): real;

var  x, y       : array [0 . . N] of proc of real;
     changes    : array [0 . . N − 1] of proc of boolean;
     chosen     : [0 . . N − 1];

procedure New_Values;
     for all k : 0 ≤ k ≤ N do in parallel
          x[k] := a + k ∗ (b − a)/N;
          y[k] := f(x[k])
     end
end New_Values;

procedure Sign_Changes;
     for all k : 0 ≤ k < N do in parallel
          changes[k] := (y[k] ∗ y[k + 1] ≤ 0)
     end
end Sign_Changes;

procedure New_Interval;
     chosen := min{k | (0 ≤ k < N) ∧ changes[k]};
     a := x[chosen]; b := x[chosen + 1]
end New_Interval;

begin
     while b − a > ε do
          New_Values; Sign_Changes; New_Interval
     end ;
     Root_ε := a
end Root_ε
```

Figure 3. Root of a real function.

$$\text{Root\_}\epsilon\ (a, b, f, N)$$
$$\{\ X;\ (A \le \text{Root\_}\epsilon \le B) \wedge (\exists x \in [\text{Root\_}\epsilon, \text{Root\_}\epsilon + \epsilon] \text{ such that } f(x) = 0)\ \}$$

To prove the correctness of the algorithm, we introduce the following predicate:

$$GOOD\_INTERVAL\ (a, b, A, B) \equiv ([a, b] \subseteq [A, B]) \wedge HAS\_ROOT\ (a, b).$$

Later we will prove the following:

$$\{\ X;\ (a = A) \wedge (b = B) \wedge HAS\_ROOT\ (a, b) \wedge (N \ge 2)\ \}$$

while $b - a > \epsilon$ **do**
    New_Values; Sign_Changes; New_Interval
end

$$\{\ X;\ GOOD\_INTERVAL\ (a, b, A, B) \wedge (b - a \le \epsilon)\ \}$$

Then, notice that $GOOD\_INTERVAL\ (a, b, A, B) \wedge (b - a \le \epsilon)$ implies that

$$(a \in [A, B]) \wedge (\exists x \in [a, a + \epsilon] \text{ such that } f(x) = 0).$$

Finally, by the axiom of sequential assignment we have

$$\{\ X;\ (a \in [A,B]) \wedge (\exists x \in [a, a+\epsilon] \text{ such that } f(x)=0)\ \}$$
$$\text{Root\_}\epsilon := a$$
$$\{\ X;\ (\text{Root\_}\epsilon \in [A,B]) \wedge (\exists x \in [\text{Root\_}\epsilon, \text{Root\_}\epsilon+\epsilon] \text{ such that } f(x)=0)\ \}$$

That implies the postcondition of Root_ϵ.

To prove the specification of the **while** loop, we choose the invariant

$$I \quad \equiv \quad ([a,b] \subseteq [A,B]) \wedge \mathit{HAS\_ROOT}\ (a,b) \wedge (N \geq 2).$$

We would like to prove $\{\ X;\ I\ \}$ New_Values; Sign_Changes; New_Interval $\{\ X;\ I\ \}$. We do it in two steps.

Lemma 3: The following holds

$$\{\ X;\ I\ \} \text{ New\_Values; Sign\_Changes; New\_Interval } \{\ X;\ [a,b] \subseteq [A,B]\ \}.$$

Proof. Starting from the postcondition $\{\ X;\ [a,b] \subseteq [A,B]\ \}$ and applying the axiom of assignment backwards we obtain the following

$$\begin{gathered}
\{\ X;\, p = \min\{k \mid 0 \leq k < N \wedge f(a+k*(b-a)/N) * f(a+(k+1)*(b-a)/N) \leq 0\} \\
\wedge\, [a+p*(b-a)/N, a+(p+1)*(b-a)/N] \subseteq [A,B]\} \\
\text{New_Values;} \\
\{\ X;\ p = \min\{k \mid 0 \leq k < N \wedge y[k]*y[k+1] \leq 0\} \wedge [x[p], x[p+1]] \subseteq [A,B]\} \\
\text{Sign_Changes;} \\
\{\ X;\ p = \min\{k \mid 0 \leq k < N \wedge \mathit{changes}[k]\} \wedge [x[p], x[p+1]] \subseteq [A,B]\} \\
\text{New_Interval} \\
\{\ X;\ [a,b] \subseteq [A,B]\}.
\end{gathered}$$

We need to prove

$$I \Longrightarrow\ p \text{ is defined } \wedge [a+p*(b-a)/N, a+(p+1)*(b-a)/N] \subseteq [A,B].$$

As $N \geq 2$ and $a < b$ we can divide the interval $[a,b]$ in N subintervals

$$[a, a+\frac{b-a}{N}], [a+\frac{b-a}{N}, a+2*\frac{b-a}{N}], \ldots, [a+(N-1)\frac{b-a}{N}, b].$$

By construction, for every $0 \leq k < N$ we have $[a+k*\frac{b-a}{N}, (k+1)*\frac{b-a}{N}] \subseteq [a,b]$. As f, a, b verify $\mathit{HAS\_ROOT}\ (a,b)$ we have that

$$\exists k : 0 \leq k < N : f\big(a+k*\frac{b-a}{N}\big) * f\big(a+(k+1)*\frac{b-a}{N}\big) \leq 0$$

therefore the p given above is defined and

$$[a+p*(b-a)/N, a+(p+1)*(b-a)/N] \subseteq [A,B]$$

and this concludes this lemma. □

By similar techniques we can prove:

Lemma 4: The following holds

$$\{ X;\ I \} \text{ New\_Values; Sign\_Changes; New\_Interval } \{ X;\ \mathit{HAS\_ROOT}\ (a, b) \}.$$

When we have $\{ X;\ P \}\ S\ \{ X;\ Q \}$ and $\{ X;\ P \}\ S\ \{ X;\ Q' \}$, we can conclude $\{ X;\ P \}\ S\ \{ X;\ Q \wedge Q' \}$. As the preceding instructions do not modify N we have

$$\{ X;\ I \}$$

$$\text{New\_Values; Sign\_Changes; New\_Interval}$$

$$\{ X;\ ([a, b] \subseteq [A, B]) \wedge \mathit{HAS\_ROOT}\ (a, b) \wedge (N \geq 2) \}.$$

As $(a = A) \wedge (b = B) \wedge \mathit{HAS\_ROOT}\ (a, b) \wedge (N \geq 2)$ implies I, and $I \wedge (b - a \leq \epsilon)$ implies $\mathit{GOOD\_INTERVAL}\ (a, b, A, B) \wedge (b - a \leq \epsilon)$ we have proved the specification of the while, formally:

Lemma 5: The following holds

$$\begin{gathered} \{ X;\ (a = A) \wedge (b = B) \wedge \mathit{HAS\_ROOT}\ (a, b) \wedge (N \geq 2) \} \\ \textbf{while } b - a > \epsilon \textbf{ do } \text{New\_Values; Sign\_Changes; New\_Interval } \textbf{end} \\ \{ X;\ \mathit{GOOD\_INTERVAL}\ (a, b, A, B) \wedge (b - a \leq \epsilon) \} \end{gathered}$$

Finally, let us consider total correctness. To prove it, we must find an integer function of the program variables such that 1) decreases at each iteration, and 2) when it is zero, the condition of the **while** statement is false. Note that the quantity $b - a$ is divided by N at each iteration. Then, it is easy to verify that the function $\lfloor \log_N((b - a)/\epsilon) \rfloor$ satisfies these two conditions. In particular, with the initial values $a = A$ and $b = B$, the algorithm halts after $\lfloor \log_N((B - A)/\epsilon) \rfloor$ iterations.

Example 3: Radix Sort

Figure 4 presents a parallel implementation of Radix Sort. This algorithm provides us with the occasion to deal formally with the axioms for **count**, **enumerate**, and exclusive write. Let us introduce the following notations:

If V is a vector, $\mathit{PERM}(V)$ denotes the set of vectors that are permutations of V.

Let A be the domain of array x, that is, the set $\{0, \ldots, N - 1\}$.

A specification of Radix_Sort (in fact, of any sorting program) is then:

$$\begin{gathered} \{ A;\ (x = X) \} \\ \text{Radix\_Sort} \\ \{ A;\ (x \text{ is ordered}) \wedge x \in \mathit{PERM}(X) \} \end{gathered}$$

```
procedure Radix_Sort
    (var x: array [0 . . N − 1] of proc of {0, . . . , N − 1});

var y: array [0 . . N − 1] of proc of {0, . . . , N − 1};
    c, j: integer;

procedure Enum_Zeros (i : integer);
    for all k : (x[k] mod 2^i) < 2^(i−1) do in parallel
        c := count;
        y[k] := enumerate
    end
end Enum_Zeros;

procedure Enum_Ones (i : integer);
    for all k : (x[k] mod 2^i) ≥ 2^(i−1) do in parallel
        y[k] := enumerate + c
    end
end Enum_Ones;

procedure Exchange;
    for all k do in parallel
        x[y[k]] := x[k]
    end
end Exchange;

begin
    for j := 1 to ⌊log(maxint)⌋ + 1 do
        Enum_Zeros(j);
        Enum_Ones(j);
        Exchange
    end
end Radix_Sort
```

Figure 4. Radix Sort.

To show that Radix_Sort satisfies this specification, we must express that, after the jth iteration, vector x is already sorted if only the j less significant bits of its elements are considered. So, let us denote with "x **mod** i" the following vector

$$x \ \mathbf{mod}\ i = [x[0] \ \mathbf{mod}\ i,\ x[1] \ \mathbf{mod}\ i,\ \ldots,\ x[N-1] \ \mathbf{mod}\ i].$$

And finally, define predicate $SORTED(j)$ to be

$$SORTED(j) = ((x \ \mathbf{mod}\ 2^j) \text{ is ordered}) \wedge (x \in PERM(X)).$$

It is easy to see that the precondition to Radix_Sort implies $SORTED(0)$, and that

$$\begin{aligned} SORTED(\lfloor \log(\text{maxint}) \rfloor + 1) &= ((x \textbf{ mod } \text{maxint}) \text{ is ordered}) \wedge (x \in PERM(X)) \\ &= (x \text{ is sorted}) \wedge (x \in PERM(X)). \end{aligned}$$

so it is exactly the postcondition. Therefore, using the axiom of the sequential for, we must show that for $j > 0$ we have

$$\begin{array}{l} \{\ A;\ SORTED(j-1)\ \} \\ \quad \text{Enum\_Zeros}(j); \\ \quad \text{Enum\_Ones}(j); \\ \quad \text{Exchange} \\ \{\ A;\ SORTED(j)\ \} \end{array}$$

We do this in the following lemmas. Informally, Enum_Zeros(j) and Enum_Ones(j) divide the elements of x into two subsets and enumerate in y each of them. Let us give a name to the elements that each procedure enumerates at iteration j.

- $small(j) = \{\ k \in A\ :\ x[k] \textbf{ mod } 2^j < 2^{j-1}\ \}$.
- $big(j) = A \setminus small(j)$.

We have the following easy properties:

- $[0\ ..\ N) = small(j) \cup big(j)$.
- $\emptyset = small(j) \cap big(j)$.
- $k \in small(j), k' \in big(j) \Longrightarrow x[k] \textbf{ mod } 2^j < 2^{j-1} \leq x[k']$.
- $k \in small(j) \Longrightarrow x[k] \textbf{ mod } 2^j = x[k] \textbf{ mod } 2^{j-1}$.
- $k \in big(j) \Longrightarrow x[k] \textbf{ mod } 2^j = 2^{j-1} + x[k] \textbf{ mod } 2^{j-1}$.

Procedure Enum_Zeros(j) deals with the elements of $small(j)$. It counts the elements in $small(j)$ and enumerates them. To deal with the number of elements we introduce the predicate

$$COUNTED_{small}(j, c) \equiv (c = \#small\ (j)).$$

To deal with the enumeration we consider a mapping $small(j) \xrightarrow{y} [0\ ..\ \#small\ (j))$ which is the (unique) increasing bijection defined as $y[k] = \#small(j)_{<k}$. We describe y by the predicate

$$\begin{aligned} &ENUMERATED_{small}(j, y) \equiv \\ &\qquad (small\ (j) \xrightarrow{y} [0\ ..\ \#small\ (j)) \text{ is the increasing bijection}) \equiv \\ &\qquad (\forall k \in small(j) : y[k] = \#small(j)_{<k}). \end{aligned}$$

We can now specify procedure Enum_Zeros(j).

Lemma 6: Procedure Enum_Zeros(j) satisfies the following specification:

$$\{\ A;\ \mathbf{true}\ \}$$
$$\text{Enum\_Zeros}(j)$$
$$\{\ A;\ COUNTED_{small}(j,c) \wedge ENUMERATED_{small}(j,y)\ \}$$

Proof. Substituting Enum_Zeros(j) by its definition we obtain:

$\{\ A; \mathbf{true}\ \}$
for all $k : (x[k]$ **mod** $2^j) < 2^{j-1}$ **do in parallel**
  $c :=$ **count**;
  $y[k] :=$ **enumerate**
end
$\{\ A;\ COUNTED_{small}(j,c)\ \wedge\ ENUMERATED_{small}(j,y)\ \}$

When we go inside the **for all** statement the set of active processors is *small(j)*. Then we have

for all $k : (x[k]$ **mod** $2^j) < 2^{j-1}$ **do in parallel**
  $\{\ small(j); \mathbf{true}\ \}$
  $c :=$ **count**;
  $y[k] :=$ **enumerate**
  $\{\ small(j);\ COUNTED_{small}(j,c)\ \wedge\ ENUMERATED_{small}(j,y)\ \}$
end

To deal with the statement $y[k] :=$ **enumerate** we need $(y, small(j), \#small(j)_<)$ defining the mapping

$$(y, small(j), \#small(j)_<)[k] = \begin{cases} \#small(j)_{<k} & \text{if } k \in small(j) \\ y[k] & \text{otherwise.} \end{cases}$$

As $ENUMERATED_{small}(j,y)^y_{(y,small(j),\#small(j)_<)} \equiv \mathbf{true}$, by the axiom of enumeration we get

$\{small(j);\ \mathbf{true}\}$
$y[k] :=$ **enumerate**
$\{small(j);\ ENUMERATED_{small}(j,y)\}$

As $COUNTED_{small}(j,c)^c_{\#small(j)} \equiv \mathbf{true}$, by the axiom of counting we get

$$\{small(j);\ \mathbf{true}\}\ c := \mathbf{count}\ \ \{small(j);\ COUNTED_{small}(j,c)\}.$$

and the lemma is done. □

Procedure Enum_Ones(j) enumerates the elements of $big(j)$ starting from the number $\#small(j)$. Rather than introducing a predicate giving us the enumeration of $big(j)$ we would like to enumerate all the elements of $[0\ ..\ N) = small(j) \cup big(j)$.

- The enumeration of $small(j)$ can be done as above. We consider the (unique) increasing bijection $small(j) \xrightarrow{y} [0\ ..\ \#small(j))$ defined as $y[k] = \#small(j)_{<k}$.

- To enumerate $big(j)$ we start from $\#small(j)$. We take the (unique) increasing bijection $big(j) \xrightarrow{y} [\#small\ (j) \ .\ .\ N)$ defined as $y[k] = \#small(j) + \#big(j)_{<k}$.

To describe this enumeration we introduce the predicate

$$
\begin{aligned}
&ENUMERATED(j, y) \equiv \\
&\quad (small\ (j) \xrightarrow{y} [0\ .\ .\ \#small\ (j)) \text{ is the increasing bijection} \\
&\quad \wedge\ big(j) \xrightarrow{y} [\#small\ (j)\ .\ .\ N) \text{ is the increasing bijection}) \equiv \\
&\quad (\forall k \in small(j) : y[k] = \#small(j)_{<k}) \wedge (\forall k \in big(j) : y[k] = \#small(j) + \#big(j)_{<k}).
\end{aligned}
$$

As

$$
\begin{aligned}
&ENUMERATED(j, y)^{y}_{(y, big(j), c+\#big(j)_{<})} \equiv \\
&\qquad COUNTED_{small}(j, c) \wedge ENUMERATED_{small}(j, y)
\end{aligned}
$$

by the same techniques as in the lemma above we obtain

Lemma 7: Procedure Enum_Ones(j) satisfies the following specification:

$$
\begin{gathered}
\{\ A;\ \ COUNTED_{small}(j, c) \wedge ENUMERATED_{small}(j, y)\ \} \\
\text{Enum\_Ones}(j) \\
\{\ A;\ \ ENUMERATED(j, y)\}
\end{gathered}
$$

As Enum_Zeros(j) and Enum_Ones(j) do not modify the array x, the above lemmas give

Lemma 8: The sequential composition of Enum_Zeros and Enum_Ones satisfies:

$$
\begin{gathered}
\{\ A;\ \ SORTED(j-1)\ \} \\
\text{Enum\_Zeros}(j); \\
\text{Enum\_Ones}(j) \\
\{\ A;\ \ SORTED(j-1) \wedge ENUMERATED(j, y)\ \}
\end{gathered}
$$

To deal with procedure Exchange we consider the mapping y^{-1} where y is defined by $ENUMERATED(j, y)$. This mapping $[0\ .\ .\ N) \xrightarrow{y^{-1}} [0\ .\ .\ N)$ verifies:

- $[0\ .\ .\ \#small\ (j)) \xrightarrow{y^{-1}} small(j)$ is the increasing bijection.
- $[\#small\ (j)\ .\ .\ N) \xrightarrow{y^{-1}} big(j)$ is the increasing bijection.

Let us specify and prove procedure Exchange.

Lemma 9: Procedure Exchange satisfies:

$$\{\ A;\ SORTED(j-1) \wedge ENUMERATED(j,y)\ \}$$
$$\text{Exchange}$$
$$\{\ A;\ SORTED(j)\ \}$$

Proof. Substituting the procedure call by its body we come to

$$\{\ A;\ SORTED(j-1) \wedge ENUMERATED(j,y)\ \}$$
$$x[y[k]] := x[k]$$
$$\{\ A;\ SORTED(j)\ \}$$

In order to deal with the statement $x[y[k]]{:=}\ x[k]$ we need to consider the array (x, A, x, y). We have

$$(x, A, x, y)[k] = x[y^{-1}[k]] = x \circ y^{-1}[k].$$

By the axiom of exclusive write we have

$$\{A;\ SORTED^{x}_{(x,A,x,y)}(j)\}\ x[y[k]]{:=}\ x[k]\ \{A;\ SORTED(j)\}$$

As

$$SORTED^{x}_{(x,A,x,y)}(j) \equiv SORTED^{x}_{x\circ y^{-1}}(j) \equiv$$
$$((x \circ y^{-1} \mathbf{mod}\ 2^{j}) \text{ is ordered}) \wedge (x \circ y^{-1} \in PERM(X)))$$

we need to prove

$$SORTED(j-1) \wedge ENUMERATED(j,y) \Longrightarrow$$
$$((x \circ y^{-1} \mathbf{mod}\ 2^{j}) \text{ is ordered}) \wedge (x \circ y^{-1} \in PERM(X)))$$

First we prove

$$SORTED(j-1) \wedge ENUMERATED(j,y) \Longrightarrow (x \circ y^{-1} \in PERM(X))).$$

As $SORTED(j-1)$ is assumed x is a permutation of X. $ENUMERATED(j,y)$ implies that y^{-1} is a bijection. Therefore $x \circ y^{-1}$ is a permutation of X.

Secondly we prove

$$SORTED(j-1) \wedge ENUMERATED(j,y) \Longrightarrow ((x \circ y^{-1} \mathbf{mod}\ 2^{j}) \text{ is ordered}).$$

We can rewrite $((x \circ y^{-1} \mathbf{mod}\ 2^{j})$ is ordered) as

$$(\forall\ k,k' \in A\ \wedge\ k < k' \Longrightarrow x \circ y^{-1}[k]\ \mathbf{mod}\ 2^{j} \leq x \circ y^{-1}[k']\ \mathbf{mod}\ 2^{j}).$$

There are three cases:

Case 1: Both k and k' are in $[0\ ..\ \#small(j))$. As we assume $ENUMERATED(j,y)$ the mapping $[0\ ..\ \#small\ (j)) \xrightarrow{y^{-1}} small(j)$ is the increasing bijection and

- $k < k' \Longrightarrow y^{-1}[k] < y^{-1}[k']$.
- $k, k' \in [0 \; . \; . \; \#small(j)) \Longrightarrow y^{-1}[k], y^{-1}[k'] \in small(j)$.
- $y^{-1}[k] \in small(j) \Longrightarrow (x[y^{-1}[k]] \textbf{ mod } 2^{j-1} = x[y^{-1}[k]] \textbf{ mod } 2^{j})$ and similarly for k'.

As we assume $SORTED(j-1)$, then

$$y^{-1}[k] < y^{-1}[k'] \Longrightarrow x \circ y^{-1}[k] \textbf{ mod } 2^{j-1} \leq x \circ y^{-1}[k'] \textbf{ mod } 2^{j-1}.$$

Putting together all these results we have

$$\begin{aligned} k < k' \Longrightarrow y^{-1}[k] < y^{-1}[k'] &\Longrightarrow x \circ y^{-1}[k] \textbf{ mod } 2^{j-1} \leq x \circ y^{-1}[k'] \textbf{ mod } 2^{j-1} \\ &\Longrightarrow x \circ y^{-1}[k] \textbf{ mod } 2^{j} \leq x \circ y^{-1}[k'] \textbf{ mod } 2^{j}. \end{aligned}$$

Case 2: Both k and k' are in $[\#small \; . \; . \; N)$. In this case we have the increasing bijection $[\#small \; (j) \; . \; . \; N) \xrightarrow{y^{-1}} big(j)$. As $y^{-1}[k] \in big(j)$ we have

$$x \circ y^{-1}[k] \textbf{ mod } 2^{j} = 2^{j-1} + x \circ y^{-1}[k] \textbf{ mod } 2^{j-1}.$$

Other steps are as in case 1.

Case 3: Consider $k \in [0 \; . \; . \; \#small(j))$ but $k' \in [\#small(j) \; . \; . \; N)$. As y^{-1} is bijective we have $y^{-1}[k] \in small(j)$ but $y^{-1}[k'] \in big(j)$. Therefore

$$x \circ y^{-1}[k] < 2^{j-1} \leq x \circ y^{-1}[k'].$$

And the proof is done. $\square$

By the lemmas above we have that:

Lemma 10: The sequential composition of Enum_Zeros, Enum_Ones, and Exchange satisfies

$$\begin{array}{l} \{ \; A; \; SORTED(j-1) \; \} \\ \quad \text{Enum\_Zeros}(j); \\ \quad \text{Enum\_Ones}(j); \\ \quad \text{Exchange} \\ \{ \; A; \; SORTED(j) \; \} \end{array}$$

The lemma above and the axiom of (sequential) for statement give us the correctness of Radix_Sort.

Example 4: Dynamic processor/memory allocation

In machines with a large number of processors, it may be useful to create the high-level illusion that processors and memory are created dynamically on program demand. Figure 5 presents a program that allows an arbitrary set of processors to set up pointers to new processors, each with its own share of free memory. This program is based on a rendezvous technique: Demanding processors and free processors exchange their addresses through variables that they all know.

We present this example as an exercise in specification and refinement. First, we give an initial, very general specification of the problem. Then, we refine it adopting various decisions on implementation, until we can prove that the particular program in Figure 5 is correct.

Assume there is a set F of processors having available free memory. This information is initially stored in the array

$$\textit{free}\text{: } \textbf{array } [0..\text{MaxProc}] \textbf{ of proc of } \text{boolean}$$

Therefore the set of processors with free memory is specified initially as $F = \textit{free}^{-1}(\textbf{true})$. Note that if the precondition defines F in this way, the set F remains fixed for the rest of the program. This allows us to talk about the set of processors that were initially free even after variable *free* has been modified.

Let A be the set of processors that are active when program Parallel_Cons is called. We will assume that A is the set of processors that require extra memory.

Parallel_Cons assigns free memory to processors in A: It sets up a total mapping new : $A \rightarrow F \cup \{\textbf{null}\}$ ($new(k)$ is defined for all $k \in A$) giving to each processor $k \in A$ the address $new(k)$ of a free processor if possible, otherwise $new(k) = \textbf{null}$. More precisely:

- When not so many processors ask for extra memory, all of them get it.

$$\#A \leq \#F \Longrightarrow A \stackrel{new}{\longrightarrow} new(A) \text{ is a bijection.}$$

- When there are too many processors, a maximum number is satisfied.

$$\#A > \#F \Longrightarrow new^{-1}(F) \stackrel{new}{\longrightarrow} F \text{ is a bijection.}$$

To deal with the mapping new we introduce the following predicate

$$\begin{aligned} ALLOCATED\,(new) \equiv (&new\text{: } A \rightarrow F \cup \{\textbf{null}\} \text{ is a total mapping } \wedge \\ &\#A \leq \#F \Longrightarrow A \stackrel{new}{\longrightarrow} new(A) \text{ is a bijection } \wedge \\ &\#A > \#F \Longrightarrow new^{-1}(F) \stackrel{new}{\longrightarrow} F \text{ is a bijection }). \end{aligned}$$

After the execution of Parallel_Cons, the set of free processors has diminished. Array *free* verifies:

$$\textit{free}[k] = \textbf{true} \Longleftrightarrow new^{-1}[k]{=}\emptyset.$$

```
procedure Parallel_Cons
      (var free :    array [0..MaxProc] of proc of boolean;
        var new :    array [0..MaxProc] of proc of integer ∪ {null});

var   available : integer;
      rv : array [0..MaxProc] of proc of integer;

procedure Give_Rv;
var   required : integer;
      free_proc : array [0..MaxProc] of proc of integer;
begin
      for all k do in parallel
            required := count;
            unconditionally
                  if free [k] then
                        available := count;
                        free_proc [k] := enumerate;
                        if free_proc [k] < required then
                              free [k] := false;
                              rv [ free_proc [k] ] := k
                        end
                  end
            end
      end
end Give_Rv;

procedure Accept_Rv;
var   requestor : array [0..MaxProc] of proc of integer;
begin
      for all k do in parallel
            requestor [k] := enumerate;
            if requestor [k] < available then
                  new [k] := rv [ requestor [k] ]
            end;
            if requestor [k] ≥ available then
                  new [k] := null
            end
      end
end Accept_Rv;

begin
      Give_Rv;
      Accept_Rv
end Parallel_Cons
```

Figure 5. Dynamic processor/memory allocation.

To deal with this set we consider the following predicate, where the whole set of processors is $\mathbb{P} = [0..\mathrm{MaxProc}]$.

$$\begin{aligned}&\mathit{FREE}\ (\mathit{new}, \mathit{free}) \equiv \\ &\qquad (\mathit{free} : \mathbb{P} \to \{\mathbf{true}, \mathbf{false}\} \text{ is total } \wedge \mathit{free}[k] = (k \in F \wedge \mathit{new}^{-1}(k) = \emptyset)).\end{aligned}$$

The dynamic processor/memory allocation would meet:

Specification 11: Parallel_Cons satisfies the following specification:

$$\begin{gathered}\{A;\ \mathit{free} : \mathbb{P} \to \{\mathbf{true}, \mathbf{false}\} \text{ is total} \wedge F = \mathit{free}^{-1}(\mathbf{true})\ \} \\ \text{Parallel\_Cons} \\ \{A;\ \mathit{ALLOCATED}(\mathit{new}) \wedge \mathit{FREE}(\mathit{new}, \mathit{free})\}\end{gathered}$$

To develop any implementation we need to assume some extra hypothesis about *new*. We can implement $\mathit{new} :\ A \to F \cup \{\mathbf{null}\}$ in a variety of ways. For example, take the processors of A and F in ascending order

$$A = \{a_0, a_1, a_2, \ldots\} \qquad , \qquad F = \{f_0, f_1, f_2, \ldots\}$$

and define *new* as

$$\mathit{new}(a_0) = f_0, \mathit{new}(a_1) = f_1, \ldots$$

Of course there exists many other possible definitions. However, to develop Parallel_Cons by refinements it is better to hide as many details as possible of *new*. Let us consider implementations of *new* giving priority to the processors with small identifiers (as in the previous example). Then we consider functions *new* such that

$$\mathit{new} :\ \mathit{first}_{\#F}A \to \mathit{first}_{\#A}F \text{ is a bijection} \quad \wedge \quad \mathit{new}(A \setminus \mathit{first}_{\#F}A) = \{\mathbf{null}\}.$$

To describe this approach we introduce the predicate

$$\begin{aligned}&\mathit{LOW\_ALLOCATED}\ (\mathit{new}) \equiv \\ &\qquad (\mathit{new}\!: A \to F \cup \{\mathbf{null}\} \text{ is total } \wedge \mathit{first}_{\#F}A \xrightarrow{\mathit{new}} \mathit{first}_{\#A}F \text{ is a bijection}).\end{aligned}$$

By construction $\mathit{LOW\_ALLOCATED}(\mathit{new})$ implies $\mathit{ALLOCATED}(\mathit{new})$.

Under these hypotheses the free processors are those having large identifiers.

$$\begin{aligned}&\mathit{HIGH\_FREE}\ (\mathit{free}) \equiv \\ &\qquad (\mathit{free} : \mathbb{P} \to \{\mathbf{true}, \mathbf{false}\} \text{ is total } \wedge (\forall k \in A : \mathit{free}[k] = (k \in F \wedge \#F_{<k} \geq \#A))\end{aligned}$$

Then $\mathit{LOW\_ALLOCATED}(\mathit{new}) \wedge \mathit{HIGH\_FREE}(\mathit{free})$ implies

$$\mathit{ALLOCATED}(\mathit{new}) \wedge \mathit{FREE}(\mathit{new}, \mathit{free})$$

We can refine the specification above to the following one:

Specification 12: With the idea that the first elements of A will pair with the first elements of F we get

$$\{A;\ \mathit{free} : \mathbb{P} \to \{\mathbf{true}, \mathbf{false}\} \text{ is total} \wedge F = \mathit{free}^{-1}(\mathbf{true})\ \}$$
$$\text{Parallel\_Cons}$$
$$\{A;\ \mathit{LOW\_ALLOCATED}(new) \wedge \mathit{HIGH\_FREE}(\mathit{free})\}$$

As Parallel_Cons = Give_Rv ; Accept_Rv, we should find the right specifications for these two procedures. Give_Rv will give addresses of free processors in an array rv (rendezvous).

rv: **array** [0..MaxProc] **of proc of** integer.

Formally a rendezvous is a bijection

$$rv :\ [0..min(\#A, \#F)) \to \mathit{first}_{\#A}F.$$

There are many possibilities to construct such a bijection. For example $rv[\#F_{<k}] = k$, such that

$$rv[0] = f_0,\ rv[1] = f_1,\ rv[2] = f_2, \ldots$$

But in fact the specification of procedure Give_Rv asks just for a bijection, any bijection will work. We introduce the predicate

$$\mathit{RENDEZ\_VOUS}(rv) \equiv (rv :\ [0..min(\#A, \#F)) \to \mathit{first}_{\#A}F \text{ is a bijection}).$$

Then we can prove that Give_Rv given in Figure 5 works.

Lemma 13: The procedure Give_Rv satisfies the following specification:

$$\{A;\ \mathit{free} : \mathbb{P} \to \{\mathbf{true}, \mathbf{false}\} \text{ is total} \wedge F = \mathit{free}^{-1}(\mathbf{true})\ \}$$
$$\text{Give\_Rv}$$
$$\{A;\ (\mathit{available} = \#F) \wedge \mathit{RENDEZ\_VOUS}(rv) \wedge \mathit{HIGH\_FREE}(\mathit{free})\}.$$

Proof. After procedure Give_Rv we have the following functions

$$\mathit{free\_proc} :\ F \to \mathbb{N} \quad \text{such that} \quad \mathit{free\_proc}[k] = \#F_{<k}.$$

We have also

$$\mathit{free} :\ F \to \{\mathbf{true}, \mathbf{false}\} \quad \text{such that} \quad \mathit{free}[k] = \mathbf{false} \text{ iff } k < \#A.$$

When we consider the overall set F the function *free* can be rewritten as

$$\mathit{free}[k] = (k \in F \wedge \#F_{<k} \geq \#A)$$

The function $rv: [0..min(\#A, \#F)) \to F$ satisfies $rv[\#F_{<k}] = k$. In fact the range of this function is $first_{\#A}F$. Then

$$rv: [0..min(\#A, \#F)) \to first_{\#A}F \text{ such that } rv[\#F_{<k}] = k \text{ is a bijection.}$$

Before concluding this proof let us remark a technical point. The procedure Give_Rv contains a statement of the form

$$\textbf{if } \mathit{free}\,[k] \textbf{ then } \cdots \mathit{free}\,[k] := \cdots \textbf{ end}$$

where the array *free* is modified inside the statement. It seems that we should have problems with the rule of the conditional because the set of active processors is modified inside the satement. However, rewriting the specification as

$$\{A;\ F = \mathit{free}^{-1}(\mathbf{true}) \wedge \cdots\} \textbf{ if } \mathit{free}\,[k] \textbf{ then} \cdots \mathit{free}\,[k] := \cdots \textbf{ end } \{A;\ F = \cdots \wedge \cdots\}$$

we can write

$$\textbf{if } \mathit{free}\,[k] \textbf{ then} \{F;\ F = \mathit{free}^{-1}(\mathbf{true}) \wedge \cdots\} \cdots \mathit{free}\,[k] := \cdots \{F;\ F = \cdots \wedge \cdots\} \textbf{ end}$$

and deal with the rule of the conditional without trouble because F remains unchanged. This finishes the lemma. □

Finally the following lemma gives us an specification of Accept_Rv.

Lemma 14: The procedure Accept_Rv satisfies the following specification

$$\{A;\ (\mathit{available} = \#F) \wedge \mathit{RENDEZ\_VOUS}(rv)\}$$
$$\text{Accept\_Rv}$$
$$\{A;\ \mathit{LOW\_ALLOCATED}(\mathit{new})\}.$$

Proof. In the procedure Accept_Rv we have the following functions

$$\mathit{requestor}: A \to \mathbb{N} \quad \text{such that} \quad \mathit{requestor}[k] = \#A_{<k}$$

In particular there is a bijection

$$\mathit{requestor}: first_{\#F}A \to [0..min(\#A, \#F))$$

The statement "$\mathit{new}\,[k] := rv\,[\mathit{requestor}\,[k]\,]$" gives a bijection

$$first_{\#F}A \xrightarrow{\mathit{requestor}} [0..min(\#A, \#F)) \xrightarrow{rv} first_{\#A}F$$

Then new is a bijection on

$$\mathit{new}: first_{\#F}A \to first_{\#A}F$$

such that $\mathit{new}[k] = rv[\#A_{<k}]$ and

$$\mathit{new}(A \setminus first_{\#F}A) = \{\mathbf{null}\}$$

This is all. □

5. Nesting and Design

We have shown that it is possible to give a formal meaning to the notations used to describe PRAM algorithms. It seems that the theory and tools required are not so different from those used in the sequential case. There is an important difference, though. The borderline between high level and low level is nowadays quite clear in sequential programming. We believe that there is a long way to go before we have such a clear distinction in parallel programming. Take as examples the points • mentioned in the Introduction. Or, concerning our notation, consider the following question: does it make sense to nest for-all statements in a program?

• If we take a low-level view, an statement such as

$$\textbf{for all } k \textbf{ do in parallel } S(k) \textbf{ end}$$

indicates that an index k is given to each processor, and the instruction S is broadcast to all. Under this approach it seems nonsense to use a piece of code like

$$\begin{array}{l} \textbf{forall } k : E_1(k) \textbf{ do in parallel} \\ \quad \textbf{for all } j : E_2(j) \textbf{ do in parallel } S(j) \textbf{ end} \\ \textbf{end} \end{array}$$

because it is rather unclear which instructions should be sent to each individual processor. Then it seems that for-all statements cannot be nested. But this fact is far from being obvious.

• Let us take another view. Suppose we have an array

$$x : \textbf{array } [0..N-1, 0..N-1] \textbf{ of proc of } \text{boolean}$$

and consider the following program:

$$\begin{array}{l} \textbf{forall } i : 0 \leq i < N \textbf{ do in parallel} \\ \quad \textbf{forall } j : 0 \leq j < N \textbf{ do in parallel} \\ \quad\quad x[i,j] := \textbf{false} \\ \quad \textbf{end}; \\ \quad x[i,i] := \textbf{true} \\ \textbf{end} \end{array}$$

In a high level this text would suffice to indicate that all the assignments $x[i,j] := \textbf{false}$ can be done safely in parallel and the same happens for assignments $x[i,i] := \textbf{true}$. Therefore, this program computes the identity matrix and makes sense.

However, it does not seem wise to nest for-all's except in simple cases like this.

```
procedure Matrix_Product
    (a, b: array [0..N − 1, 0..N − 1] of proc of integer):
    array [0..N − 1, 0..N − 1] of proc of integer;

import
    Tree_Sum (a: array [0..N − 1] of proc of integer): integer;
    { returns ∑_{0≤i<N} a[i] in time O(log N) }

var
    x: array [0..N − 1, 0..N − 1, 0..N − 1] of proc of integer;

begin
    for all i, j, k : 0 ≤ i, j, k < N do in parallel
        x[i, j, k] := a[i, j] ∗ b[j, k]
    end;
    for all i, j : 0 ≤ i, j < N do in parallel
        Matrix_Product[i, j] := Tree_Sum(x[i, 0..N − 1, j])
    end;
end Matrix_Product
```

Figure 6. Matrix product.

• There is one case where the nesting of for-all's is clearly a good program methodology. It appears when we hide the for-all statement inside a procedure. Let us consider as example the matrix product, Figure 6.

In this case the unfolding of the structure

```
forall i, j : 1 ≤ i, j < N do in parallel
    Matrix_Product[i, j] := Tree_Sum(x[i, 0..N − 1, j])
end
```

implies the nesting of two for-all's. Figure 7 shows a version of the reflexive transitive closure with a high nesting of for-all instructions.

Let us give a tentative list of proverbs that may be useful in the design and proof of parallel algorithms.

- Try to avoid unconditional instructions. We believe that unconditionals are very low level statements because they break the block structure of the program. They should be used in exceptional cases that occur asynchronously, such as dynamic memory management.
- Be careful with the nesting of for-all's. When possible, hide it into procedures. A good idea is that different procedures act over different active data structures. But trying to avoid completely the nesting of for-all's can give cumbersome programs.

type matrix = **array** $[0..N-1, 0..N-1]$ **of proc of** boolean;
procedure Transitive_Reflexive_Closure (a: matrix): matrix;

import
    Bool_Mat_Product (a, b: matrix): matrix;
    { returns the matrix product $a \cdot b$ in time $O(\log N)$ }

var
    x: matrix;

begin
    for all $i, j : 0 \leq i, j < N$ **do in parallel**
        if $i = j$ **then** $x[i, j] :=$ **true else** $x[i, j] := a[i, j]$ **end**
    end;
    for $k := 1$ **to** $\log N$ **do**
        for all $i, j : 0 \leq i, j < N$ **do in parallel**
            $x[i, j] :=$ Bool_Mat_Product $(x, x)[i, j]$
        end
    end;
    for all $i, j : 0 \leq i, j < N$ **do in parallel**
        Transitive_Reflexive_Closure$[i, j] := x[i, j]$
    end;
end Transitive_Reflexive_Closure

Figure 7. Transitive reflexive closure.

- Try to avoid $S(k)$ with complicated structure in a statement like

  for all $k : E(k)$ **do in parallel** $S(k)$ **end**

- Try to program with small for-all pieces with clearly defined global intermediate states, like

  { clearly defined global state }
  for all $k : E_1(k)$ **do in parallel** $S_1(k)$ **end**;
  { clearly defined global state }
  for all $k : E_2(k)$ **do in parallel** $S_2(k)$ **end**
  $\vdots$

- When you have a statement

  for all $k : E(k)$ **do in parallel** $S(k)$ **end**

  with a local structure of $S(k)$ try to express the global state after the execution of the for-all as a union of local states. As an example we can take some well-known algorithms for numerical simulation.

- When you have a statement

$$\textbf{for all } k : E(k) \textbf{ do in parallel } S(k) \textbf{ end}$$

and different $S(k)$ interact heavily, try to give directly the global state.

6. Conclusions

Parallel machines are becoming cheaper and more flexible everyday. It is almost certain that the design of large software systems and the development of portable languages for them will be an important issue in a few years. We have two main ways to achieve massive synchronous parallelism in programming languages [5].

- One is allowing parallel assignment to simple variables. In this approach data parallelism is explicitly given by constructs such as

$$\textbf{for all } k : E(k) \textbf{ do in parallel } S(k) \textbf{ end}$$

They state explicitly which program elements can be executed in parallel and this simplifies the compiler analysis. The main idea along this work has been how to prove correctness on data parallel programs written with these statements. To do this we take assertions $\{ A \; ; \; P \}$ where we consider separately active processors and program variables. The set A denotes active processors and predicate P describes program variables. This is not only a matter of taste. Our set of axioms is easily described with specifications $\{ A \; ; \; P \}$ and more importantly, all our calculus behaves properly trough these specifications. The set of given examples proves it, therefore this framework is a workable tool. We would like to emphasize that both regular and irregular communications have been explored by this approach. The owner write axiom allow us to consider regular communications and the exclusive write assignment deals with irregular patterns. For example, radix sorting has an irregular pattern and can be proved with the set of axioms presented here.

- The other way takes variables with complex data structures as first class or atomic objects. Assignments to set or array variables represent this approach. As parallelism does not appear explicitly at statement level this approach can render the compiler analysis more difficult than the preceding one. The language SETL is a good example of this approach [23]. As we can take sets and arrays as atomic structures the classical assignment axiom seems to work. For example the set assignment instruction $S' := \{E : k \textbf{ in } S \mid C\}$ has the following axiom

$$\{ \; P \; \} \; S' := \{E : k \textbf{ in } S \mid C\} \; \{ \; Q \; \} \quad \text{iff} \quad P \text{ implies } Q^{S'}_{\{E : k \in S \mid C\}}.$$

An assignment such as $x := +/\{k \textbf{ in } S \mid C\}$ has as axiom

$$\{ \; P \; \} \; x := +/\{k \textbf{ in } S \mid C\} \; \{ \; Q \; \} \quad \text{iff} \quad P \text{ implies } Q^{x}_{\sum_{\{k \in S \mid C\}} k}.$$

Our feeling about this approach is the following. Even if the axioms are simpler than ours, their application over concrete examples could be cumbersome. This will happen because axioms does not deal explicitly with parallelism. However these two ways are conceptually close because in both cases it is possible to reason about correctness in terms of pre and post conditions. Remark that real languages take aspects from both.

We would like to consider the possible application of this work to real programming languages. We could consider whether these same proofs could be made using FORTRAN 90 or C*. It could be possible, at least ideally, to axiomatize these languages and develop programs and formal proofs together. However the approach we propose here is in a preliminay stage. For the moment, we have shown that we can reason with assertions $\{ A;\ P \}$ avoiding any mention to the computational behaviour. Recall that, as Dijkstra pointed out in [10], reasoning in terms of operational semantics is a "tremendous waste of mental effort". However we would like to note that our approach serves to clarify and explain some constructs in concrete parallel languages. As example, we consider the send construct "$[index\ ]x = y$" in C* [8]. The *send* operation distributes the elements of an array y, throughout memory according to the pattern of addresses in the array *index*. With the constructs we use, it is coded as:

$$\textbf{for all}\ \ k\ \textbf{do in parallel}\ x[\ index\ [k]\ ] := y[k]\ \textbf{end}$$

Considering the axiom of exclusive write we obtain:

$$\textbf{for all}\ \ k\ \textbf{do in parallel}\ \{\ A;\ P\ \}\ x[\ index\ [k]\ ] := y[k]\ \{\ A;\ Q\ \}\ \textbf{end}$$

iff P implies ($index\ [k]$ as function of k is injective) $\wedge$ $(Q^x_{(x,A,y,index)})$ where

$$(x, A, y, index)[k] = \begin{cases} y[\ index\ ^{-1}(k)\ ] & \text{if } index\ ^{-1}(k) \in A \\ x[k] & \text{otherwise} \end{cases}$$

To understand accurately how the send construct works, we take an example given in chapter 10 of the C* Programming Guide [8]. We consider three arrays x, y, and *index* of five components, indexed 0..4. The set of active processors is $A = \{0, 1, 2\}$. Initially the value of x is undefined, $y = [0, 10, 20, 30, 40]$, and $index = [1, 3, 0, 4, 2]$. After the send operation the value of x is $(x, A, y, index) = [20, 0, \quad , 10, \quad]$. Remark that $x[2]$ remains undefined because $index^{-1}(2) \notin A$ but $x[3] = 10$ because $index\ ^{-1}(3) \in A$. The axiom for exclusive write gives us a clear mathematical explanation of the following comment in [8]. *In a send operation, the inactive position cannot send a value, but active positions can send to it.*

We consider now the limits of our approach. In addition to data parallelism there are other paradigms for massive parallelism. One of them is associative computing. The term "associative computing" seems to have two main meanings. One is related with networks working with real numbers [20]. Our programming model seems to be very different. The second one is represented by models having "matching" mechanisms. The *tuple space* and the **in(s)** operation of LINDA are good examples [4]. This approach differs from ours in two aspects. First it is *asynchronous* and second it is *nondeterministic*. It would be possible to extend our formalism to deal with nondeterminism but it seems clearly inadequate to consider asynchronicity. The problem here, as in concurrent programming, is that the notion of "program state" is not clearly defined when different processes run asynchronously. Approaches dealing with *temporal logic* such as UNITY [6] or *traces* such as CSP [18] could be more adequate.

Acknowledgement. We thank Luc Bougé for pointing out an error in an earlier version.

7. References

1. Akl, S.G.: *The Design and Analysis of Parallel Algorithms.* Prentice-Hall 1989.
2. Blelloch, G.: *Vector Models for Data–Parallel Computing.* MIT Press 1990.
3. Bougé, L.; Levaire, J.-L.: Control structures for data-parallel SIMD languages: semantics and implementation. *Future Generation Computer Systems,* **8**, 363–378 (1992).
4. Carriero, N.; Gelerntner, D.: *How to Write Parallel Programs.* MIT Press 1990.
5. Chandy, K.M.; Kesselman, C.: *IEEE Software.* November 1991, 11–20.
6. Chandy, K.M.; Misra, J.: *Parallel Program Design.* Addison–Wesley 1989.
7. Clint, M.; Narayana, K.T.: Programming structures for synchronous parallelism. In *Int. Conf. "Parallel Computing 83"*, 405–412. M. Feilmeier, J. Joubert, and U. Schendel (editors). Elsevier (North–Holland) 1984.
8. *C* Programming Guide,* Version 6.0.2, June 1991. Thinking Machines Corporation, Cambridge, Massachusetts.
9. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, **18**, 453–457 (1975). Reprinted in *Programming Methodology, a Collection of Articles by Members of WG2.3.* D. Gries (editor). Springer-Verlag 1978.
10. Dijkstra, E.W.: On the cruelty of really teaching computing science. *Communications of the ACM*, **32**, 1398–1404 (1989).
11. Fortune, S.; Wyllie, J.: Parallelism in random access machines. In *Proc. 10th ACM Symp. on Theory of Computing*, 114–118 (1978).
12. Gibbons, A.; Rytter, W.: *Efficient Parallel Algorithms.* Cambridge University Press 1988.
13. Gries, D.: *The Science of Programming.* Springer-Verlag 1981.
14. Hagerup, T.; Schmitt, A.; Seidl, H.: FORK a high-level language for PRAMs. *Future Generation Computer Systems,* **8**, 379–394 (1992).
15. Hillis, D.W.: *The Connection Machine.* MIT Press 1985.

16. Hillis, D.W.; Steele, G.L.: Data parallel algorithms. *Communications of the ACM*, **29**, 1170–1183 (1986).

17. Hoare, C.A.R.: An axiomatic basis for computer programming. *Communications of the ACM*, **12**, 576–580 (1969). Reprinted in *Essays in Computer Science*, 45–58. C.A.R. Hoare and C.B. Jones (editors). Prentice-Hall 1989.

18. Hoare, C.A.R.: *Communicating Sequential Processes.* Prentice–Hall 1985.

19. Karp, R.M.; Ramachandran, V.: Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science* (vol. A), 869–941. J. van Leeuwen (editor). Elsevier and MIT Press 1990.

20. Kohonen, T.: *Self–Organization and Associative Memory.* Springer–Verlag 1989.

21. van Leeuwen, J.; Wiedermann, J.: Array processing machines: an abstract model. *BIT*, **27**, 25–43 (1987).

22. Rünger, G.; Sieber, K.: A trace based denotational semantics for PRAM Language FORK. Internal Report, Univ. Saarbruecken–Univ. Kaiserslautern, 1/92.

23. Schwartz, J.T.; Dewar, R.B.K.; Dubinsky, E.; Schonberg, E.: *Programming with Sets and Introduction to SETL.* Springer–Verlag 1986.

24. Stewart, A.: An Axiomatic treatement of SIMD assignment. *BIT*, **30**, 70–82 (1990).

25. Utard, G-E.: Un système axiomatique pour les langages massivement parallèles SIMD. Rapport de DEA, LIP-ENS Lyon 1991.

26. Wyllie, J.C.: The Complexity of Parallel Computation. PhD Thesis. Cornell University 1979.

27. Yourdon, E.: *Writings of the Revolution.* Yourdon Press 1982.

Biographies

JOAQUIM GABARRÓ received the M.S. degree in physics from Universitat Autònoma de Barcelona in 1975. He is Doctor in mathematics (thèse de III cycle) from Université Pierre et Marie Curie, Paris VI, in 1981 and Doctor in computer science from Universitat Politècnica de Catalunya in 1983. Co-author of the books *Structural Complexity* vols. I and II (Springer Verlag, 1988,1990). He is currently associate professor of computer science at Universitat Politècnica de Catalunya. His reseach interests include automata and formal languages, complexity, and parallel algorithms.

RICARD GAVALDÀ received the M.S. and Doctor degrees in computer science from Universitat Politècnica de Catalunya, in 1987 and 1992. He is currently an associate professor at that university. His research interest is theoretical computer science, with emphasis in complexity theory, all forms of parallel computing, and computational learning theory. He was a visiting researcher at the University of California at Santa Barbara from october 1991 to june 1992.