

SYSTEMATIC BUILDING OF A DISTRIBUTED RECURSIVE ALGORITHM EXAMPLE: THE SHORTEST PATH ALGORITHM

G. FLORIN*, R. GÓMEZ** AND I. LAVALLÉE**

Abstract. Designers of distributed algorithms must deal with a variety of issues including sequential algorithms design, communication protocols, fault tolerance. The distributed design must also include a proof step of the whole algorithm features. This paper gives a new scheme for the design of distributed algorithms. In this approach the design step is performed simultaneously with the proof step. Our distributed design method is mainly based on parallel recursive schemes, but recursivity is used in a distributed environment, so we use two existing and widely available tools: remote procedure call, and the PAR instruction parallel execution of threads.

1. Introduction

Designers of distributed problems must bring matter to several issues. The questions at issue include the classical problems of sequential algorithm design (for each sequential process running on a given processor). Moreover there are synchronization features to consider. Designers must specify communication protocols managing cooperation between sequential processes.

There are four major problems which can be considered:

- The communication protocols in order to manage the cooperation between processes.
- The fault tolerance which must be studied carefully.
- Performance problems, or dead line time constraints, in-time critical applications.
- Both formal specification solutions and proof.

Received October 22, 1994.

Key words: distributed algorithms, recursivity, wave, remote procedure call, shortest path

The paper [6] proposes a new scheme for the construction of algorithms founded on distributed recursivity.

In sequential programming the recursive design of algorithms is elegant but not very efficient, but recursive design in distributed algorithms is both elegant and efficient.

Moreover, the systematic use of the remote procedure call, (RPC) provides a high level of designing which avoids managing many problems of synchronization and communication.

Lamport, Shostak and Pease's solution of the Byzantine general problem [7] is, as far as we know, the first example of distributed recursive programming. However, since that paper is not concerned with programming style, the solution given there is not easy to follow.

In this paper our purpose is to show, with these concepts, how to generate a parallel/distributed program from the proof of an algorithm. We envisage that the generated program is to be elegant and efficient as possible.

This paper is organized as follow: the first part introduces the recursive distributed concept. The second part explains the different steps of the formal proof. And in the last part we give an example of the proof, namely the shortest path algorithm, and we show how to generate the code from the proof.

2. Recursive distributed programming scheme (RDPS)

In this section we introduce the basic recursive distributed programming scheme.

2.1. Recursive waves

A recursive wave in this paper is defined by a procedure. This procedure, during its execution, calls n (or none) concurrent remote executions of itself. Hence executing a recursive wave leads to a tree such that:

- the root is associated with the first execution of the procedure.
- the root and the nodes are associated with blocked procedures waiting for the termination of all the respective parallel recursive calls.
- the leaves are associated with the active execution of the procedure; or with the ended execution of a procedure that did not make any recursive call.

Moreover a recursive wave defined by a procedure is often integrated in a set of procedures. These procedures encapsulate a data structure, as for example in an ADA package (but also as in the modular, or as in the object oriented approaches). The data structure includes, for example, the current processor identifier, a set of neighbour's identifiers, etc.

In this paper we assume that all communications and synchronizations between the successive remote executions are defined by the remote procedure call scheme. RPC: for example the transmission of control and input parameters, and the procedure exit (return to the calling procedure associated with the transmission of the output parameters). There are no complementary data or messages exchanged between the remote execution of the procedure on different processors.

On the other hand, there can be a need for communications between the different executions of the same procedure on a given processor. These communications are achieved by using shared variables.

2.2. Specification of the general scheme

We will describe features that will help to develop the proof and the construction of distributed recursive algorithms. We use two approaches for this: programming and mathematical scheme.

2.2.1. Programming scheme

We consider concurrent execution of n procedures, therefore a programming control structure for the parallel execution of n threads on the same processor must be available. We assume that the following instruction **par**, similar to the OCCAM, can be used.

```
par i in <domain> do
  <INSTRUCTION BLOCK>;
enddo;
```

For each parameter value in a discrete set, ($\langle \text{domain} \rangle$), a thread associated with the instruction block is activated on the processor executing the **par** instruction. This instruction is terminated when all activated threads have terminated and the next instruction is executed.

In order to define the location of remote procedure execution we consider the following syntax for a RPC execution of a given procedure, (named `<pcd_name>`) on a remote processor (named `<prcs_id>`).

```
<pcd_name> (<parameter_list>) on <prcs_id>;
```

In Figure 1 we present the general description of a recursive wave. The instruction blocks represent any kind of sequence.

```
type processor_identifier is....; - Type processor name
i: processor_identifier;
procedure recursive_wave ({parameters}) is
<DECLARATION BLOCK>
begin
  <INSTRUCTION BLOCK A>
  if { condition } then - stop/exit of the recursive wave
    <INSTRUCTION BLOCK B>
    par i in < DOMAIN> do - execution of concurrent threads
      <INSTRUCTION BLOCK C>
      recursive_wave ({ parameters }) on i; - RPC execution
      <INSTRUCTION BLOCK D>
    enddo;
    <INSTRUCTION BLOCK E>
  endif;
  <INSTRUCTION BLOCK F>
end recursive_wave;
```

Figure 1: General description of the recursive wave

The structure is defined in an ADA-like specification language. As we often use the notion of processor groups we have to manage sets. Therefore we consider that a package implementing a type set (setof) is available with its associated operations (union, intersection,...).

The execution of this recursive wave builds a tree of active processes on the network, and this tree is oriented by the relation caller-called which define a father-son relationship and the root of this tree is the initiator of the computation.

2.2.2. Mathematical scheme

The previous definition of a recursive wave, induces that the recursive parallel called R can be defined as a set of four functions:

$$R = \{L, S, G, H\}.$$

These four functions are many to many ones.

Function L

This function defines the set of local data to each site. In each site i , there can be several simultaneous executions of the recursive procedure.

To each execution of the procedure is associated a data set. In a recursive environment the same data can take different values according to the execution level. Let DL_i^j be the local data set of the execution level j on the site i . This set DL_i^j is built from two subsets of data:

1. THE SITE DATA: the data local to i and global to the executions of the procedure: named DS_i ;
2. THE EXECUTION DATA: the data proper to each execution: local to i and used only at the execution level j : named DE_i^j .

Therefore we can say:

$$DL_i^j = DS_i \cup DE_i^j.$$

During a procedure execution the values of the local data can be modified by the local code. L is the function that represents any of these modifications. It depends on two parameters: the values of local variables modified at the execution level k , DL_i^k ; and the input parameters values of the procedure, called VP .

Thus we define the function L as:

$$DL_i^j = L(DL_i^k, VP).$$

Function S

The par instruction activates a set of threads. The function S creates the set of indices of these threads, (i.e. the discrete set named <domain> in the

general scheme). The set is calculated with the local values, DL_i , of the present active procedure; and the input parameters values, VP . Then we define the function S as:

$$\{< \text{DOMAIN} >\} = S(DL_i, VP).$$

Function G

It gives the set of parameters of the RPC, which is going to be sent to the site k from the site i . This set is computed with the local data of i , the input parameters values, and the receiver site identification, k . So G is a function of three parameters:

$$\{\text{PARAMETER VALUES}\} = G(DL_i, VP, k).$$

Function H

It computes the return value before the end of the recursive. This value will be sent to the calling procedure. This function represents the last step in the recursive wave. It is computed with the local values, the input parameters values, and the set of values collected from the son's sites, H_{sons} . We are going to define H as a three parameters function:

$$< \text{RETURN VALUE} > = H(DL_i, VP, H_{sons}).$$

H_{sons} is a set, made by the results of the sons of the site i . If i is a leaf then H_{sons} is an empty set.

2.2.3. Example of the mathematical scheme

In 2.1 we said that the execution of a recursive wave leads to a tree. Here we present a general example of that tree, and the relationship between this tree and the functions defined above.

Figure 2 presents a tree generated by a recursive wave, named *recre*, with three parameters, initiated at the site i , (the root in the tree).

Following the description of a recursive wave, the site i executes a RPC on his neighbours: j , k , l . Each RPC has different parameters values: $\{u, v, w\}$ for j , $\{u', v', w'\}$ for k , and $\{u'', v'', w''\}$ for l . The set of neighbours is generated

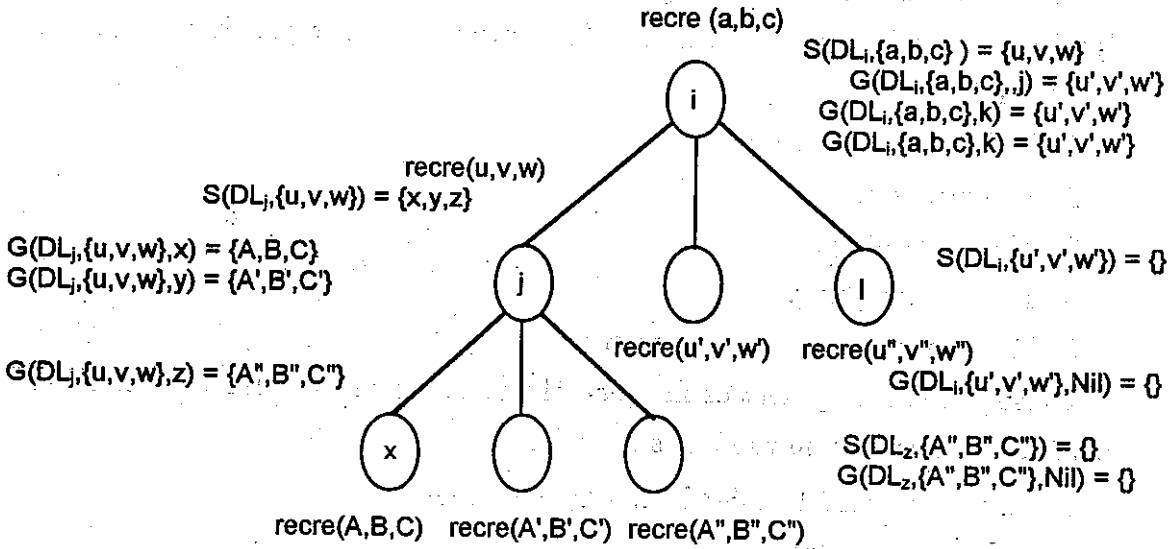


Figure 2: Example of the mathematical scheme.

by the function S, and the function G gives the value parameters of the RPC for each site.

Since sites k, l do not perform any recursive call, these sites are two leaves of the tree. When k and l end their computations, they send the results to i , (their father in the tree). Each result is calculated by means of the function H.

The site j executes a RPC on the sites x, y, z calculated by S, with the value parameters computed by G: $\{A, B, C\}$ for x , $\{A', B', C'\}$ for y , and $\{A'', B'', C''\}$ for z . Later, they send their results to the site j .

2.2.4. Formal description

We use the set of functions defined in Subsection 2.2.3 and the general description of a recursive wave (Figure 1) in order to create a formal description of a recursive wave. This formal description is presented in Figure 3.

The lines of code have been numbered, because they will be referenced in the next section for the explanation of the general description development.

3. General description of algorithm's development

In this section we will show the steps for the development of a recursive distributed algorithm.

One of our objectives is the generation of the code from the proof of the algorithm. In this way we divide the description in five steps, these steps are in a special order.

1. Definition of the local values.
2. The visit strategy.
3. The descending.
4. The backtracking.
5. The coherence of the local values.

During the development of the general proof scheme we will make reference to the code described in the last subsection.

```

1 type processor_identifier is....; - Type processor name
2 DS; -Site local data set
3 i: processor_identifier;
4 procedure recursive_wave (VP: parameters value, Hi: Result ) is
5 DE; - Execution local data set
6 begin
7   DL=A(VP, DL) - < INSTRUCTION BLOCK A >
8   if { condition } then - stop of the recursive wave
9     DL=B(VP, DL) - <INSTRUCTION BLOCK B >
10    par i in S(DL, VP) do - S: set of indices threads
11      DL=C(VP, DL) - <INSTRUCTION BLOCK C >
12      recursive_wave (G(DL, VP, i), Hi) on i; - G: parameters function
13      DL=D(VP, DL) - <INSTRUCTION BLOCK D >
14    enddo;
15    DL=E(VP, DL) - <INSTRUCTION BLOCK E >
16  endif;
17  DL=F(VP, DL) - <INSTRUCTION BLOCK F >
18  R=H(DL, VP, Hi) - H: backtrack function
19 end recursive_wave;
```

Figure 3: General description of the Recursive Wave

3.1. First step: Definition of the local values set

The first step is to define the local value set. This definition is concerned with the next arguments:

- Definition of the site data: DS_i
- Definition of the execution data: DE_i^j
- Definition of the input parameters

These arguments will lead us to a construction of the declaration part of a recursive wave code. Following the code presented in Figure 3 it is easy to see that the first step of the proof is concerned with the lines 2,3,4 and 5 of this code.

3.2. Second step: The visit strategy

The designer of a distributed algorithm is concerned first with the choice of a visit strategy to different sites. For example the visit can be performed according to a predefined spanning tree, [5]; or it must be such that each path, without circuits, from a root site to any reachable site represents a path of the recursive tree.

So the goal in this step is to show that the distributed algorithm realizes the designed visit strategy.

From the previous concepts we can see that it is necessary to:

- Define a run way of the algorithm coherent with the goal of the calculus.
- Show that the pattern of the recursive calls and the stop test induce the expected visit strategy.
- Show that the termination condition is correct, i.e. the visit strategy doesn't generate an infinite tree.

We can see that these arguments are concerned with Lines 8, 9 and 10 of the code presented in Figure 3. More specifically, they are related to the definition of "condition" at Line 8 and the definition of the function S at Line 10.

3.3. Third step: The descending

The recursive calls induces a network traversal which supports a distributed computation. This distributed computation is done in two steps: the first is called the *descending*, and the second *backtracking*.

The descending traversal of the network generates a tree. In the last section we give arguments to calculate the different identifier sites to which the RPCs are to be sent; but we also need to know the parameters values of these RPCs.

Within each procedure execution on a vertex, we perform operations with local data and input parameters values of the procedure. After performing these operations we send a RPC, with new parameters values, according to the

background traversal and the result of the perform on local data. After some recursive calls, the descending stops. The last vertices reached by a RPC are the leaves of the recursive calls tree.

The proof of the “descending” is concerned with the following arguments:

- The presentation of an assertion, and its demonstration for all different levels of the tree.

- The criterion of correctness of the calculated parameters (it is different in each algorithm).

- The stopping criterion.

Thus, we obtain the proof of some properties in all visited vertices, and then particularly for the leaves of the tree. We can see that these properties are concerned with the lines 12, 13 and 14 of the formal description of the wave; especially with the definition of the function G at Line 13.

3.4. Fourth step: The backtracking

The backtracking of a recursive wave is done from leaves to the root; it goes through the father, and the father of the father, etc. This way induces a result collect of the recursive calls, and a local computation with the local data and the collected results.

When the computation on local parameters induces that any other consequent call has to be performed; then the current process is a leaf of the recursive calls tree, and the recursive backtrack can begin. The proof of the properties of this backtrack is obtained according to the following arguments:

- from the leaves we prove that a property is true on all leaves which have the same father;

- proving the passing way from the vertex of depth p , to one of depth $p - 1$, in the recursive call tree.

This step is related to the function H . If we take a look at the formal description code, we can see that this function is related to the line 18 specifically and to the lines 16, 17 and 19, too.

3.5. Fifth step: Coherence of the local values

Every time that a procedure is executed in a site, the local values of that site are modified. We must take care of this modifications, because they can change the final result of the algorithm.

In Section 2.2.2 (Mathematical Scheme) we defined DL_i^j as the set of local data of the execution j on the site i . Each time when a procedure is executed on a site, the local data are modified, but we can't know the sequence of the different modifications. Then we are obligate to synchronize and/or serialize the different modifications of the data.

For this purpose we can use the next arguments:

1. RPC management¹.
2. The mutual exclusion.
3. Stamping the modifications of the local data.
4. The algorithm's nature.

The selection of the solution depends on the problem. For example, in the shortest paths problem, the serialization is implied by the algorithm.

4. Example: The shortest path problem

In this section we will apply all concepts in the previous section to the shortest path problem. In view of a better understanding of the proposed solution we will show the informal consideration that have been taken. Next we will proceed with the development of the proof, instead of finding a recursive solution to this problem. We will follow the different steps introduced in Section 3.

4.1. Informal considerations

In a computer network each edge has a certain cost, (for example induced by the state of the messages in the buffer). A typical problem is to find the path between two vertices with the minimum cost possible.

Here the problem consists in the computation of the shortest paths from a vertex r called root, (which is the initiator of the calculus), to all vertices of a

¹This management is made by a lower level software.

valued graph. In order to build the shortest paths tree, eventually we want to obtain all single paths of the graph from the root to all others.

Many solutions have been proposed, [4, 3] to the shortest path problem. Our solution is based on an implicit enumeration method. This means that the basic algorithm potentially scans all possible rooted paths.

On the other hand we use two optimizations in order to avoid scanning unnecessary paths.

Our first optimization avoids building elementary circuits, sending RPCs to the caller. The second optimization uses the Bellmann's principle [1]: it computes the path length and then selects the shortest.

In contrast with many authors, we suppose that all costs are positives and all edges are bidirectional, i.e. the valuation of (a, b) is not necessarily the same as the one of (b, a) . This case is the most realistic, but perhaps more difficult to handle.

Thus at the start of the calculus, each vertex x knows its identifier, the identifier of its neighbours (i.e. $\Gamma(x)$), and the costs of all arcs, $C(x, y)$. At the end of the computation, each vertex knows its father in the shortest paths tree, the weight of the path from r to itself, and its sons in the shortest paths tree².

In view of the formal proof of the scanning, we must establish the following properties:

- all possible paths are potentially scanned, and particularly the shortest one.
- the elimination of longer paths doesn't avoid the construction of the shortest path.
- if some path from the initiator to another vertex is longer, (i.e. with larger cost) than another path from the initiator to the same vertex; it is not necessary to continue the construction of this path since it can't become the shortest path. This is due to the fact that all costs are positives and with a convex cost function; which induces that the global optimum is a function of local costs. It is the Bellmann's principle which is an application of the Pontryagin principle to the convex functions.

²It is possible to modify this algorithm in order to obtain more information at each vertex, and especially the root could know the entire shortest paths tree.

4.2. Notations in view of the formal algorithm proof

The following notation is used in the proof:

$GR = (X, \Gamma, C)$	the weighted and directed graph
$\Gamma(x)$	the set of neighbours of the site x
r	the root
$PC(x, y)$	the cost (weight) of path (x, y)
$PC_{min}(x, y)$	the minimal path cost from x to y
$C(x, y)$	the cost (weight) of arc (x, y)
$Ch^{(j)} = \{x_0 = r, x_{i_1}, x_{i_2}, \dots, x_j\}$	the path from the root to x_j .

The path weight value can also be expressed as follow:

$$PC(Ch^{(j)}) = PC(x_0, x_j) \quad \text{the path cost from the root to } x_j.$$

5. First step of the proof: Definition of the local value set

The first step is the definition of the local value set. Like it was stated in Section 3.1, this step must follow the next arguments:

- Definition of the local data to each site.
- Definition of the local data to each execution.
- Definition of the input parameters.

5.1. Definition of the local data to each site, DS_i

In the shortest paths problem the local data set of a site i , DS_i , is composed of two subsets: a dynamic one, named DSd_i ; and a static one, named DSs_i .

The data in DSd_i is said to be dynamical because their values are modified every time that a new site is involved. Then we define:

$$DSd_i = \{mpc_i, son_i(y), father_i\}.$$

The data in the statical set, DSs_i , will not change at any moment of the execution, (the cost of the edges is the same during all the running time).

$$DSs_i = \{y \in \Gamma(i), C(i, y)\}.$$

Thus the local data set at a site i is:

$$DS_i = DSd_i \cup DSs_i.$$

$\Gamma(i)$ and $C(i, y)$ have been defined, as for the rest:

- mpc_i current minimum cost of the path from the root, r ,
to the site i
- son_i the son of the site i
- $father_i$ the father of the site i .

5.2. Definition of the local data to each execution, DE_i^j

In Subsection 2.2.2 the DE_i^j has been defined as the data proper to each execution. The local data in every site i , that change at every execution j are the sons of that site i . So we define:

$$DE_i^j = \{local_son_i\}.$$

$local_sons_i$ represents the sons of the site i at the execution j . They are all the neighbours of the site i , except his father³. Then $local_sons_i$ can be computed like:

$$DE_i^j = \Gamma(i) - callers_list(i),$$

where $callers_list(i)$ are the callers arriving to the actual site i , (here is included the father of the site).

5.3. Definition of the parameters, VP

From the previous subsections we can see that the values needed in every site at any execution are the path cost, and the callers list of a site i . Then VP , the set of input values parameters, is defined like:

$$VP = \{pcost, callers_list\},$$

³A problem of coherence will be discussed later in Section 9.

where:

pcost is the cost of the path
caller_list the callers list defined previously.

5.4. Definition of the function L

In Subsection 2.2.2 we defined the local data set like:

$$DL_i^j = DS_i \cup DE_i^j.$$

The initial value of the local data is:

$$mpc_i^0 = +\infty.$$

The modifications of the remaining local data will be defined later.

Then from the previous subsections we can make the next declarations of code:

```
type InRec is record
  callers_list : integer;
  pcost : string;
end record
```

```
type OutRec is record
  mpc : integer;
  father : string;
  sons : string;
end record
```

```
procedure short_path(in Wave: InRec, out Rec: OutRec) is
```

Here *short_path* is the name of the recursive wave, InRec et OutRec are the input and output parameters. This code, as all the codes introduced later, is in ADA-like specification language.

6. Second step: Formal proof of the visit strategy

In the shortest path problem the visit strategy proof is to know which nodes will be the receivers of the RPC calls. This involves the next considerations:

- Compute of the set of nodes receiving the RPC, in view of building all the paths from the root to all sites.

- Avoid the construction of the circuits, and show that this action does not affect the problem of finding the shortest path.
- Set up the stopping condition for the recursive algorithm.

6.1. Recursive building of the paths of length l

The specification of the path building is given by:

$$l = 0 \quad Ch^{(0)} = \{x_0 = r\}$$

$$l \neq 1 \quad Ch^{(1)} = \{x_0, x_{i_1} \mid x_0 \in Ch^{(0)}, \text{ and } \forall x_{i_1} \in \Gamma(x_0)\}$$

...

$$l = n \quad Ch^{(n)} = \{x_0, x_{i_1}, x_{i_2}, \dots, x_{i_{n-1}}, x_{i_n} \mid x_0, x_{i_1}, x_{i_2}, \dots, x_{i_{n-1}} \in Ch^{(n-1)}, \\ \text{and } \forall x_{i_n} \in \Gamma(x_{i_{n-1}})\}.$$

PROPERTY. All paths of length n are built in this described way⁴.

PROOF. If all paths of length $n - 1$ are built, then all paths of length n are potentially built. This is due to the fact that they are obtained by extension of all paths of length $n - 1$ and by all neighbours of their last edges.

LEMMA. According to the property, there is no path of length n omitted.

PROOF. Suppose that it is false. Then there is a path of length n which was not obtained as an extension of a path of length $n - 1$. It is absurd because all paths of length n contain one path of length $n - 1$ and, by the way all paths of length $n - 1$ were previously built.

6.2. Avoidance of circuits construction

One of the characteristics of this algorithm is that it avoids the construction of elementary circuits. We will show how to do this, and give consequences of this peculiarity.

⁴Remember that the length of a path is given by the number of edges.

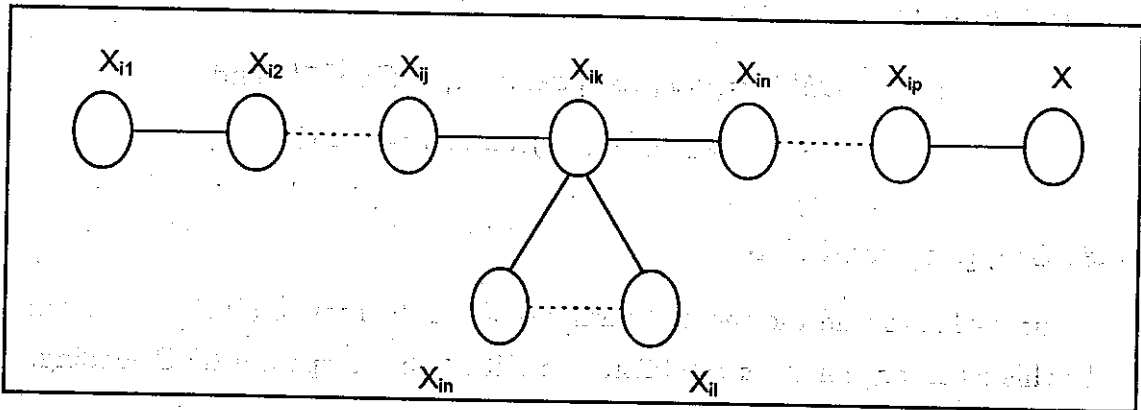


Figure 4: Example of a path containing circuits.

PROPERTY. Avoiding the construction of circuits preserves the minimum cost paths.

PROOF. Consider the path described in Figure 4. We can see that this path contains a circuit, denoted by x_{ik} , x_{il} and x_{im} .

Let $PC^{circuit}$ be the cost of the path from x_{i1} to x , including the circuit. This cost is given by:

$$PC^{circuit} = PC(x_{i1}, x_{i2}) + \dots + PC(x_{ij}, x_{ik}) + PC(x_{ik}, x_{il}) + \dots \\ \dots + PC(x_{im}, x_{ik}) + PC(x_{ik}, x_{in}) + \dots + PC(x_{ip}, x).$$

Let $PC^{no-circuit}$ be the cost of the path from x_{i1} to x , without the circuit; it is computed as follows:

$$PC^{no-circuit} = PC(x_{i1}, x_{i2}) + \dots + PC(x_{ij}, x_{ik}) + PC(x_{ik}, x_{in}) + \dots \\ \dots + PC(x_{ip}, x).$$

Then we can say that:

$$PC^{circuit} > PC^{no-circuit}$$

The circuits construction can be avoided by sending no RPCs to the previous caller,⁵ (i.e. the father in the execution tree).

⁵This avoidance is a critical point of the algorithm, and it is linked to the stopping condition.

Thus at Step n we have:

$$l = n \quad Ch^{(n)} = \{x_0 x_1 \dots x_n \mid x_0 \dots x_{n-1} \in CH^{(n-1)}, \text{ and } x_n \in \Gamma(x_{n-1}), \text{ and } x_n \neq x_{n-2}\}.$$

6.3. Stopping condition

Our goal is to find the shortest path cost from the root to all sites. We can take this as an argument to establish a condition that stops the RPC sending.

We start with the next property:

PROPERTY. Let pc_1 be the path cost of a path called *path1*, and pc_2 the path cost of another path called *path2*. If pc_1 is greater than pc_2 , it is not necessary to continue with the *path1*.

PROOF. The path obtained from *path1* can not be better than the continuation of *path2*. Hence there is not necessary to continue with the first one.

CONSEQUENCE 1. It is not useful to continue the building of a path that will reach a site, in which the cost will be greater than a previously computed one.

CONSEQUENCE 2. If the path cost needed with the current RPC is not better than a previously computed one, then the wave is stopped and the RPC activation is terminated.

Considering this, we set the next rule:

Rule:

$$\text{If } \begin{cases} PC(Ch^{(k)}) < PC(Ch^{(k-1)}) \Rightarrow \text{send the recursive call.} \\ PC(Ch^{(k)}) \geq PC(Ch^{(k-1)}) \Rightarrow \text{stop the recursive call.} \end{cases}$$

6.3.1. Proof of the algorithm termination

The only case where the algorithm never ends can be presented when the recursive calls continue without stopping it. This means that the stop condition of the recursive call $PC(Ch^{(k)}) \geq PC(Ch^{(k-1)})$ will never be true.

On the other hand the number of edges is finite and the building paths are elementary, (i.e. without circuits) then the number of building paths is finite

and at the end of each path the previous condition becomes necessary true and then the algorithm stops.

6.4. Definition of the function S

S is the function that creates the indice set containing the identifiers of the sites which will receive the RPC. From the previous sections, we define S as:

$$S(DL_i^{j-1}, VP) = \begin{cases} \{\emptyset\} & \text{if } mpc_i^j \geq mpc_i^{j-1} \\ \{\Gamma(i) - callers_list\} & \text{if } mpc_i^j \text{ otherwise,} \end{cases}$$

where mpc_i^j is the current minimum path cost from the root to the site i , calculated at j .

NOTE: The function S could be define differently. If we don't want to send the parameter *callers_list*, which is a long message, it is possible to use only $\Gamma(i) - father$. In such a case the sent message size is shorter, but the average number of messages can be higher. The higher average messages of the shortest path tree algorithm is $O(\log_n)$ and if we suppose that the message size is \log_n , as in the present case, then the complexity of sending *callers_list* is, roughly speaking, $O(\log_n^2)$.

6.5. Code generation

From the definition of the function S and the previous demonstrations we can generate the following lines of code:

```

if pcost < mpc then
  union(caller_list, site_id)
  mpc = pcost
  par i in ( $\Gamma(i)$  - callers_list) do
    :
    :
  enddo.

```

NOTE: In the previous code *union* is a set function, that adds a new element to a set. Its syntax definition is

union(*<set identifier>*, *<new element>*).

7. Third step: The recursive descending scheme proof

The method enumerates a set of paths in order to find the shortest one. It is necessary to show that the proposed paths construction from the root to the different sites, includes the shortest path. Thus we have to:

- show how to calculate the cost of a path, and the minimum path cost.
- show that at the end of the descending, all the leaves will have a possible shortest path.

7.1. Path costs computation

For the root $x_0 = r$ we set

$$PC(Ch^{(0)}) = PC(x_0, x_0) = 0,$$

i. e. the path cost of the root to the root is zero. From this we can compute the path cost from the root to other sites:

$$PC(Ch^{(1)}) = PC(x_0, x_{i_1}) = PC(Ch^{(0)}) + C(x_0, x_{i_1}) \text{ for } x_{i_1} \in \Gamma(x_0),$$

$$PC(Ch^{(2)}) = PC(\{x_0, x_{i_1}, x_{i_2}\}) = PC(Ch^{(1)}) + C(x_{i_1}, x_{i_2}) \text{ for } x_{i_2} \in \Gamma(x_{i_1}),$$

and in general we have:

$$PC(Ch^{(n)}) = PC(\{x_0, x_{i_1}, \dots, x_n\}) = PC(Ch^{(n-1)}) + C(x_{n-1}, x_n). \quad (1)$$

Using (1) we will compute the minimal path cost in the following subsection.

7.2. Definition and calculus of the minimum cost

We define E_i^p , as the p^{th} value of the minimum cost waited from the site i .

Initially, at $p = 0$, the cost will be infinite to all sites except for the site 0, (remember that we want the minimum cost from the site 0, i.e. the root, to any site).

$$E_i^0 = \infty \quad \text{for } i \neq 0.$$

The site i receives a first cost value, $PC(Ch^{(j)})$, from a neighbour j . Then from the equation (1) of the last subsection, we can estimate a first path cost value from the site i to the root:

$$PC^{(1)}(Ch^{(i)}) = PC(Ch^{(j)}) + C(i, j),$$

where $PC^{(1)}(Ch^{(i)})$ is the first path cost value. Then the minimum cost expected in the site i is:

$$E_i^1 = \min[PC^{(1)}(Ch^{(i)}), E_i^0].$$

A second cost value arrives, $PC(Ch^{(k)})$, from another neighbour, (or from the father in the recursive tree). Let k be the identifier site of that neighbour, and $PC(Ch^{(k)})$ its path cost. Then we can compute a second path cost value, from the root to the site i :

$$PC^{(2)}(Ch^{(i)}) = PC(Ch^{(k)}) + C(i, k).$$

The second minimum cost waited at i will be:

$$E_i^2 = \min[PC^{(2)}(Ch^{(i)}), E_i^1].$$

Let $PC^{(m)}(Ch^{(i)})$ be the m^{th} path cost value, computed from all path costs that arrive to the site i , then we can define E_i^p as:

$$E_i^p = \min[PC^{(m)}(Ch^{(i)}), E_i^{(p-1)}].$$

PROPERTY. The function E_i^p is decreasing and monotonic in p and has 0 as a lower bound.

PROOF. Since all path costs are positive, the function E_i^p selects only between positive values, and at each time a lower value than the previous one. As there are no negative values, the possible lowest value is zero.

PROPERTY. The function E_i^p converges to $PC_{min}(r, i)$.

PROOF. It is true thank to the fixed point theorem. If the minimum path cost is computed, then this value corresponds to the lower bound.

LEMMA. With positive costs, if a path contains a circuit, it is not minimal.

PROOF. It is obvious because the path obtained with avoidance of the circuit is of lower cost (Bellman's principle).

LEMMA. *The shortest path from a site r to all sites x is obtained in a finite number of scans.*

PROOF. Any path with more than n edges is avoided because it contains a circuit, and it can't become a shortest path; on the other hand the number of vertices is finite.

CONSEQUENCE. *After the last visit of the recursive wave, the shortest path is computed.*

7.3. Definition of the function G

Since G is the parameter value function and every site needs to know the path cost and the callers set, we define:

$$G(DL_i^j, VP, k) = \{pcost, callers_list_i + i\} \text{ if } \{\Gamma(i) - callers_list_i\} \neq \{\emptyset\},$$

where i is the identifier site.

7.4. Code generation

From the G definition, and the previous subsections, we can generate the following lines of code:

```

par i in ( $\Gamma(i) - callers\_list_i$ ) do - Generated in the previous subsection
    pcost=pcost+cost(id_site,i);
    RPC(pcost, callers_list) on i;
enddo

```

8. Fourth step: Proof of the recursive backtrack

The recursive backtrack is the step where one gathers the global information concerning the shortest path tree. The recursive calls compute a semi global information in each vertex. These are the shortest path cost from the root to the current site, and its father in this shortest path. The recursive backtrack allows to collect the information created after the corresponding recursive call by the other visited sites to the current site. Thus a site knows its sons in the shortest path tree. It could be possible to collect the global information from each vertex to the root, but we prefer to collect site local informations because

global informations collected from the root are not necessarily interesting and it can take an important amount of information in each message.

So our problem is to compute in each vertex the value of the shortest path from the root, the identifier of the predecessor in the shortest path, and also the list of sons in the shortest paths tree. This is the major difference with classical distributed algorithms for shortest path of non oriented graphs. In this case we are talking about paths rather than chains.

Every site i uses the following informations:

- its father in the shortest path tree,
- the minimum cost of the path from r to it.

For a site i , the missing information is the set of its sons in the shortest path tree. This information is obtained as it is shown in the next subsection.

8.1 Computation of the identifiers of the sons site

At the end of the "descending" each site knows its father in the shortest path tree, $father_i^*$; which can be different from its father in the execution tree, $father_i$.

Let x_i be a site, and $y_{i_1}, y_{i_2}, \dots, y_{i_k}, \dots, y_{i_n}$ its sons in the execution tree, (see Figure 5). Then:

$$father_{y_{i_1}} = father_{y_{i_2}} = \dots = father_{y_{i_k}} = \dots = father_{y_{i_n}} = x_i,$$

and we can establish the following rule:

RULE:

$$\text{if } father_{y_k}^* = x_i \text{ then } sons_{x_i}^* = \text{union}(y_k, sons_{x_i}^*),$$

where $sons_{x_i}^*$ are the sons of the site x_i in the shortest path tree, and union is the set function defined in 6.5.

This means that if the father's identifier in the shortest path tree of the site y_k is equal to the identifier x_i , y_k is its son in the shortest path tree.

The opposite case, ($father_{y_k}^* \neq x_i$) occurs when to the site x_i arrives a better cost from another node. This makes that x_i changes its father identifier,

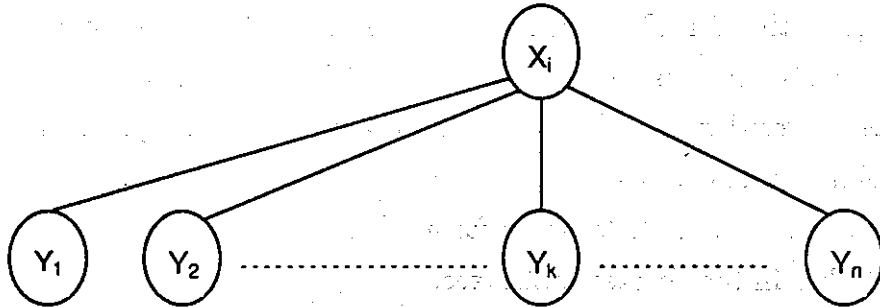


Figure 5: A site with its sons.

(in the shortest path) to the one that corresponds to the arrived improvement: this father will be different to the first one.

8.2. Definition of the function H

The function H computes and sends the result of the wave backtracking to all vertices of the execution tree. At the beginning of the section we mentioned the information required by each vertex, then the function H is given by:

$$H(DL_i^j, VP, Hsons) = \{id_site, father_i^*, sons_i^*, pcost\}.$$

We have to remember that $father_i^*$ is the father of the site i in the shortest path tree, and not necessarily the father in the execution tree.

8.3. Code generation

The calculus of site sons identifier in the shortest path tree is specified by the next code:

```

while i in (neighbour - callers_list) do
  if father(i) = ego then
    sons(i) = concat(sons(i), ego)
  endif;
enddo
  
```


9. Fifth step: The local value coherence

The local value coherence, in the shortest path problem, must be treated in the backtracking. During the recursive backtrack, a vertex must know which are its sons; however, it's difficult to know if a neighbour is the son of a vertex, because it can become son of another vertex. Then the collect of the information about the sons of a vertex is difficult.

We can have the problem of overlapping executions on global variables, due to the change of status from father to son, or the opposite. When a vertex x , with a previous father y , receives an improving visit of the recursive wave, from a vertex z then it considers that z becomes it's new father and x broadcasts the recursive wave to all its neighbours except z . In this case y which was the previous father of x receives a new instance of the wave and if it is not an improving one, y deletes the status of son for x , and if the new instance is an improving one, it executes the current algorithm. Thus we are also concerned by overlapping execution of RPC.

The only problem is that the last modification should be the good one. In order to solve this problem, a more general way is to build a total order on the transactions on a process. In our case there are two ways in order to assume this:

- a systematic solution for this problem is to give a sequence number, (a watch stamp) at each sent message or RPC. It is a systematic way but it is not necessary here;
- it is possible to use the cost paths values, to induce an order between modifications. In the following we will use this solution.

10. Algorithms' specification

The code of the recursive wave for the shortest path problem is described in Figure 6. As for the general description of the recursive wave, we use ADA-like specification language for the presentation of this code.

An example of the execution of this code is given in Figure 7. In this figure attention must be focused to the definition of the tree functions: S, G and H, and the variation of their values during the execution.

```

type InRec record
  callers: integer;
  pcost: string;
end record

type OutRec is record
  mpc: integer;
  father: string;
  sons: string;
end record

procedure short-path (in Wave: InRec, out Res: OutRec) is
  i: processor_identifier;
  ego: processor_identifier; - the identifier of the current site
  callers_list, neighbours: set of processor-identifier;
  alien: array [1..MAX_NEIGHBOURS] of OutRec;
  R: OutRec;
begin
  if Res.mpc < Wave.pcost then
    concat (Wave.callers, ego)
    Res.mpc=Res.pcost
    par i in ( $\Gamma(i)$  - callers_list) do
      Wave.pcost = Wave.pcost + cost(ego, i)
      short-path (Wave, R) on i;
      alien(i)=R;
    enddo;
  endif;
  while i in (neighbours-callers_list) do
  if alien(i).father=ego then
    R.sons=concat(R.sons, ego)
  endif;
  enddo
end short-path;

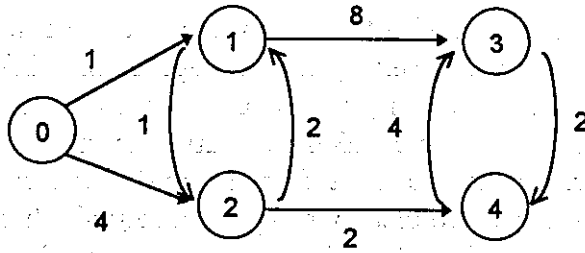
```

Figure 6: Code of the recursive solution to the shortest path problem.

11. Conclusions

We have introduced a basic method of construction for distributed algorithms. It is based on the distributed recursive wave concept. As an example of this method we presented solution for the shortest path problem, another examples can be found in [5].

a.



b.

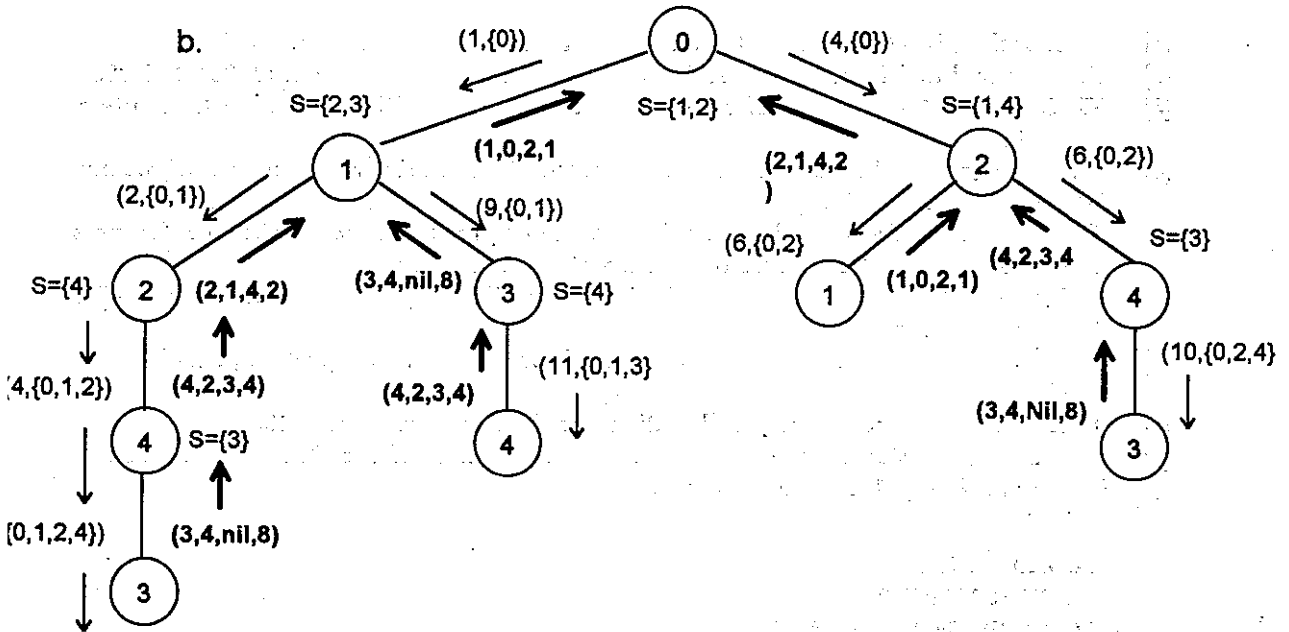


Figure 7: Example of a possible execution of the recursive wave algorithm, for the shortest path problem.

The algorithms obtained by this method are simple to be read and understood.

We have shown a general scheme for the construction of recursive distributed algorithms at the same time as we make the algorithm's proof. It is clear that each problem will be treated in a different way; we will find problems for which the first step of the scheme will take more development, and others that will require more attention in other steps of the proof scheme.

All this work has been applied in one way or other. Many algorithms have been implemented and tested down the recursive distributed concept, (see [9]).

FUTURE WORK:

The failure semantic of the RPC is a problem that was not mentioned here. We believe that one can use the exception concept, as in ADA, in order to treat the problem of faults. At the same way the management of some problems due to the semantic of RPC will be studied in a future work.

Another work that must be treated is the optimization of the algorithms which are described here. We also envisage to apply all these concepts in the area of parallel programming, that is to say on parallel computers without shared memory.

REFERENCES

- [1] D. Bersekas, R. Gallager, "Data Networks," Prentice Hall, 1987, pp. 318-322.
- [2] A. D. Birrell and B. J. Nelson, ACM Trans. on computing systems 1 (1983), 222-328.
- [3] C. C. Chen, *A distributed algorithm for shortest paths*, IEEE Trans. on Computers, Sept. (1982), 398-399.
- [4] K. M. Chandy, J. Misra, *Distributed computation on graphs: Shortest path algorithm*, CACM 25 (1982), 833-837.
- [5] G. Florin and I. Lavallée, *La récursivité, mode de programmation distribuée*, Rapport de Recherche No. 1536, INRIA-Rocquencourt, Octobre 1991.
- [6] G. Florin, R. Gómez and I. Lavallée, *Recursive distributed programming schemes*, ISADS 93, Kawasaki, Japan.
- [7] L. Lamport, R. Shostak and M. Pease, *The Byzantine Generals problem*, ACM Toplas 4 (1982), 382-401.
- [8] I. Lavallée, "Algorithmique parallèle distribuée," Hermès ed., Paris, 1990.
- [9] N. Pizigot, *La récursivité répartie*, Mémoire d'Ingénieur of the Cons Nat. des Art et Metiers (CNAM), 1992, Paris, France.

* CEDRIC, CNAM
292 RUE St MARTIN
75130 PARIS CEDEX, FRANCE

** ACTION PARADIS, INRIA
DOMAINE DE VOLUCEAU, ROCQUENCOURT BP 105
78153 LE CHESNAY CEDEX, FRANCE