HAL
open science

La Recursivite, mode de programmation distribuee

G. Florin, Ivan Lavallee

To cite this version:

G. Florin, Ivan Lavallee. La Recursivite, mode de programmation distribuee. [Rapport de recherche] RR-1536, INRIA. 1991. inria-00075026

HAL Id: inria-00075026

https://inria.hal.science/inria-00075026v1

Submitted on 24 May 2006

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

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

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Rapports de Recherche

N° 1536

Programme 1
Architectures parallèles, Bases de données, Réseaux et Systèmes distribués

LA RÉCURSIVITÉ, MODE DE PROGRAMMATION DISTRIBUÉE

Gérard FLORIN
Ivan LAVALLÉE

Octobre 1991

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

RR-1536

Recursive Distributed Programming Schemes

LA RÉCURSIVITÉ, MODE DE PROGRAMMATION DISTRIBUÉE

Gérard Florin[1] Ivan Lavallée[2]

Mots clefs : récursivité, vague récursive, appel de procédure distante, élection distribuée, état global stable, routage, plus court chemin.

Key words : recursivity, wave, remote procedure call, distributed election, global state detection, shortest path.

[1]CÉDRIC-CNAM, Centre d'Études et Recherches en Informatique ; Conservatoire National des Arts et Métiers, 292 rue St Martin, 75141 Paris Cedex 03 (FRANCE)
[2]Université Paris VIII UFR LIT, 2 rue de la liberté 93000 St Denis et action *PARADIS*-INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex (FRANCE)

Résumé

L'expression récursive des algorithmes est bien connue en algorithmique séquentielle comme étant le plus souvent élégante, mais très inefficace. L'avènement des réseaux et machines distribués conduit à revoir ce statut. En effet, il apparaît qu'après avoir explicité un certain nombre de primitives, dans le domaine de la programmation des systèmes distribués, la récursivité y est, non seulement élégante, mais aussi efficace. Du moins n'induit-elle pas des coûts d'exécution supérieurs aux algorithmes itératifs correspondants.

De plus, l'écriture récursive est unificatrice. Nombre de problèmes d'algorithmique distribuée peuvent être résolus par un algorithme récursif. Nous présentons ici trois premiers schémas simples de récursivité distribuée appliqués à quelques problèmes de contrôle. Cette méthodologie peut s'étendre a des problèmes de calcul du type produit matriciel par blocs, des problèmes d'optimisation combinatoire comme les algorithmes de Séparation et évaluation Progressive, des algorithmes du type A*, AO*, alpha/bêta, ou encore SSS*, nous y reviendrons, car alors le schéma récursif de base change.

Nous précisons donc les notions afférentes aux primitives que nous utilisons, et montrons leur implémentabilité. Après quoi, nous décrivons les schémas de principe de la récursivité distribuée, et montrons comment résoudre un certain nombre de problèmes de contrôle classiques (élection sur un anneau, diffusion sur une structure virtuelle d'arbre couvrant) et nous donnons des algorithmes nouveaux pour résoudre deux problèmes classiques du domaine, la détection d'état stable (terminaison, protocole de validation à deux phases) et le calcul des plus courts chemins d'un sommet initiateur à tous les autres, avec établissement des tables de routage en chaque nœud du réseau.

Abstract

This paper is devoted to the recursive distributed programming styles. We present three variants of recursive wawes and some related protocol examples. We show that the distributed iteration scheme (defined using asynchronous message passing) can be expressed using distributed recursivity at the same cost. This programming style doesn't support the inefficiency drawback of the single computer recursivity, because when a RPC is implemented in a distributed operating system a distributed recursive call is possible and is performed at the same cost than a RPC call. We have seen that due to the interest for the RPC scheme it is very often and efficiently implemented.

Hence the main interest of a recursive programming style can be used without restriction : the higher level of abstraction and the natural way to design a solution in numerous cases.

In the first section we present a classification of recursive wawes (recursive distributed programming schemes). The second section is devoted to the presentation of recursive programming in a sequential recursive wave. The third section is devoted to the presentation of recursive programming on a spanning tree In the fourth section, we introduce the flowding wave which induces its proper control structure (which is a tree) and we show how to use this programming style in order to solve two majors control problems. First the global quiescent state detection protocol, (which can be see also as a termination detection protocol or a two step commit one). Second we give an original solution of the one to all nodes shortest path computation.

1. Introduction

L'outil le plus utilisé en programmation distribuée jusqu'à une époque récente a été l'échange asynchrone de messages entre processus séquentiels (voir [TAN 89]). C'était la seule façon de construire des algorithmes distribués jusqu'à ce qu'on dispose de l'appel de procédures à distance (A.P.D.). L'APD devient le schéma de contrôle le plus employé car c'est l'outil idéal pour construire des protocoles basés sur le système client-serveur. Plus généralement, l'APD est l'outil privilégié pour l'écriture des systèmes opératoires distribués orientés-objet. Curieusement, il y a peu de littérature proposant l'APD comme outil général de programmation distribuée. En fait, l'APD est considéré seulement comme un outil de programmation distribuée orientée objet (voir MMS -Manufacturing Messages Standard- [TAN 89]).

En programmation séquentielle, les structures de contrôle abstraites du type "if...then...else...", "do loop..."ou "while loop..." ont joué un rôle fondamental. En programmation distribuée, il y a également besoin de structures de contrôle similaires. C'est ce que nous proposons ici. Les algorithmes présentés ne sont pas tous intrinsèquement nouveaux, c'est la manière de les exprimer qui l'est, et surtout qui présente un aspect méthodologique original.

La démarche se situe dans le courant de pensée qui considère que le programmeur, ou celui qui conçoit un algorithme, n'a pas plus à gérer les envois et réceptions de messages, que le programmeur ou le concepteur d'algorithme séquentiels n'a à gérer le chargement des registres et les adresses de sa machine ; seul importe de gérer une *communication* avec un autre processus ou un groupe de processus. L'APD fournit un outil privilégié pour ce faire. Notre démarche permet à la fois de s'abstraire des aspects trop techniques liés à l'échange de message, mais de rester à un niveau de réalisme suffisant pour avoir accès à la gestion du parallélisme du système, réseau ou machine, considéré.

Les structures de contrôle utilisées en programmation distribuée pour l'activation et la synchronisation sont déduites de structures de contrôle utilisées sur les machines séquentielles. Pour l'instant, les

outils les plus utilisés pour l'activation et la synchronisation de tâches ou de processus en calcul distribué (à part l'échange de messages et l'APD) est l'exécution en parallèle des primitives "fork, join" ou "parbegin ... parend".

Le schéma de programmation distribuée le plus intéressant et le plus général qui ait été utilisé pour construire des solutions d'une large classe de problèmes distribués (détection de terminaison, plus court chemin) est associé à la notion de calcul diffusant [DI,SC 80],[CHA 82] ou vague.

En programmation séquentielle, itération et récursivité apparaissent comme étant deux styles complémentaires de programmation. En général, la présentation récursive des algorithmes et programmes est élégante, mais inefficace, alors que c'est souvent l'inverse avec la présentation itérative.

Le présent travail est consacré à la programmation distribuée récursive. Nous présentons trois variantes de vagues récursives et quelques algorithmes basés sur ces variantes. Nous montrons que le schéma d'itération répartie (défini à partir de l'échange asynchrone de messages) peut être facilement exprimé en utilisant une forme récursive sans que cela induise un surcoût en terme de messages. Cette programmation n'a pas à supporter la lourdeur de la gestion de la vague qu'on rencontre dans le modèle d'échange de messages car, lorsque l'APD est implémenté dans un système distribué, un appel récursif distribué est possible et est exécuté au même coût qu'un simple APD.

Vu son intérêt, l'APD est maintenant très répandu dans les différents systèmes distribués, il est donc naturel d'en étendre l'usage.

L'intérêt majeur d'un style de programmation récursif peut alors être utilisé sans restriction, c'est à dire : un haut niveau d'abstraction et une façon simple et naturelle pour élaborer les solutions dans de nombreux cas.

Dans la première section, nous donnons une classification de vagues récursives.

Dans la deuxième section nous donnons un exemple de vague récursive distribuée séquentielle : un protocole d'élection sur un anneau.

La troisième section est consacrée à la présentation d'un exemple de programmation récursive d'une vague distribuée et parallèle sur une structure de contrôle en arbre couvrant : une diffusion.

Dans la quatrième section, nous introduisons la notion de vague inondante, laquelle génère sa propre structure de contrôle (qui est un arbre) et nous montrons comment utiliser cette vague inondante pour résoudre deux problèmes de contrôle majeurs (et génériques), à savoir :

- la détection d'état global stable (terminaison, protocole de validation à deux phases, synchronisation, etc.) ;

- la détection des plus courts chemins d'un nœud à tous les autres (établissement des tables de routage) ;

2. Schéma de base de programmation distribuée récursive

Nous introduisons ici le schéma de base de la structure de contrôle d'un algorithme distribué récursif. Les caractéristiques essentielles du modèle virtuel d'exécution considéré dans cette section sont :

- le niveau de parallélisme utilisé durant l'exécution (séquentielle ou parallèle);

- la topologie virtuelle de contrôle sous-jacente utilisée par le système distribué (anneau, arbre couvrant), ou l'absence d'une telle topologie.

2.1. Présentation informelle des vagues récursives

Il y a plusieurs façon d'introduire la récursivité en environnement distribué. Nous ne considérerons ici, dans un premier temps, que les cas les plus simples. Tout d'abord, une vague récursive est une procédure. Cette procédure appelle durant son exécution n autres exécutions concurrentes d'elle même sur d'autres processeurs. Cette vague récursive induit un arbre d'exécutions de procédures tel que :

- la racine de l'arbre est associée à la première exécution de la procédure ;

- la racine et les nœuds non feuilles sont associés à des procédures dont l'exécution est en attente de terminaison des appels récursifs qu'elles ont lancés ;

- les feuilles sont associées aux exécutions actives de la procédure ; avant ou après l'exécution d'appels récursifs, ou encore associées avec une exécution terminale sans appel récursif.

De plus, une vague récursive définie par une procédure est le plus souvent intégrée dans un ensemble de procédures. Ces procédures sont associées à une structure de données comme par exemple un paquetage ADA, ou encore comme en approche "objet". La structure de données comprend par exemple l'identificateur du processeur courant, les identificateurs de tous les voisins ... Nous utilisons cette approche par la suite.

hypothèses

Nous supposons dans tout ce qui suit que toutes les communications et synchronisations entre les différentes exécutions récursives sont définies par le principe de l'APD (transmission des paramètres de contrôle et d'entrée, retour à la procédure appelante, du contrôle et des paramètres de sortie). Il n'y a aucun autre message de donnée, ni signal échangé durant toute l'exécution de la procédure, sur la même couche logicielle.

2.2. Spécification

Puisque nous supposons possible l'exécution d'un APD sur n procédures, nous devons disposer d'une structure de contrôle d'exécution concurrente de n tâches sur un même processeur, au sein d'une même procédure. Nous considérerons donc par la suite, disposer de l'instruction par définie ci-après et ayant la même sémantique qu'en OCCAM :

par <paramètres> **in** <domaine> **do**

bloc d'instructions ;

```
        end do ;
```

A chaque valeur prise par le paramètre sur le domaine est associée une tâche parallèle exécutée sur le processeur courant. L'instruction par est terminée lorsque toutes les tâches parallèles qu'elle a activées sont terminées.

Le calcul se poursuit alors en séquence par l'exécution de l'instruction suivante.

Afin de rester cohérent avec l'esprit de l'appel de procédure distante, nous considérerons aussi la structure suivante permettant de définir à la fois une procédure (nom : <proc_name>), et la localisation de l'exécution, le nœud du réseau, le processeur sur lequel elle s'exécute (on <proc_id>) :

```
<proc_name> ( <parameter_list> ) on <proc_id>;
```

D'autres règles peuvent être ajoutées précisant, suivant le contexte, l'exécution de cette procédure. Dans ce qui suit, un identificateur de processus est généralement associé à une variable qui prend ses valeurs dans un domaine déterminé (le groupe de processeurs).

L'algorithme suivant est spécifié en quasi_ADA. Comme nous utiliserons souvent la notion de groupe de processeur, nous devons gérer cet ensemble. Nous considérerons donc un paquetage implémentant un type ensemble (setof) et les opérations qui lui sont associées (union, intersection...).

La spécification (i.e. définition) de la programmation de la vague récursive distribuée est alors :

```
type id_de_processeur is;      -- Définition de type pour un nom de
                               -- processeur
procedure vague_récursive ( <paramètres>) is
< déclarations> ;
i : id_de_processeur;
groupe_de_processeurs : setof id_de_processeur;

begin
        -- Le bloc d'instructions représente ici tout ensemble
        -- séquentiel d'instructions exécutées localement, y compris
        -- en parallèle
< bloc d'instructions A > ;
```

```
        -- La structure de contrôle if ... then ... else suivante est
    -- celle relative à la fin de la récursivité .
    if condition then
        < bloc d'instructions B > ;
        -- L'instruction par provoque l'exécution en parallèle, sur --
chaque processeur du groupe, un appel récursif de la
        -- procédure vague_récursive.
        par i in groupe_de_processeurs do
            <bloc d'instructions C > ;
            -- f(<paramètres> représente l'opération effectuée sur
    -- les paramètres par la première partie de la
            -- procédure.
            vague_récursive ( f ( <paramètres> ) ) on i ;
            <bloc d'instructions D > ;
        enddo ;
        <bloc d'instructions E>
    endif
    < bloc d'instructions F > ;
    end vague_récursive
```

Nous allons maintenant présenter trois sortes de vagues récursives.

3. Vague récursive séquentielle

3.1. Définition

Hors l'exécution récursive séquentielle classique sur un processeur unique, la façon la plus simple d'utiliser un appel récursif de procédure dans un système distribué est d'appeler la même procédure sur un seul autre site. En ce cas, l'arbre d'exécution est réduit à une chaîne (au sens des graphes). Il n'y a alors qu'une seule exécution active de la procédure associée a un processeur, et c'est le dernier de la chaîne (le premier étant l'initiateur du calcul).

En algorithmique distribuée, l'unique structure de contrôle se prêtant à ce type d'exécution est l'anneau (virtuel). Si on suppose qu'une telle structure est pré-existante et que chaque processeur connaît son prédécesseur et son successeur, alors la vague récursive séquentielle peut être facilement implémentée. Le schéma de principe de la vague distribuée séquentielle récursive est alors :

```
    successeur : id_de_processeur ;    -- déclaration du processeur
                                       -- successeur
    procedure  vag_rec_seq ( <paramètres>) is
        < déclarations> ;
        begin
            <bloc d'instructions A > ;
```

```
            if condition then
                < bloc d'instructions B > ;
                vag_rec_seq (f(<parameters>)) on successor ;
                <bloc d'instructions D > ;
            endif
            < bloc d'instructions E> ;
end vag_rec_sec
```

3.2. ÉLection sur un anneau

Dans ce paragraphe, le schéma de vague séquentielle récursive est adapté et spécifié en vue de l'exécution d'un protocole d'élection d'un processus sur un anneau. Cette solution, comme pratiquement toutes les autres, part de l'hypothèse que tous les identificateurs des processus sont deux à deux différents et se contente, non d'élire un processus quelconque parmi tous, mais d'élire le processus ayant le plus petit (respectivement. le plus grand) identificateur. Dans ce qui suit, l'initiateur (i.e. l'identificateur de l'initiateur) de la vague récursive est utilisé pour le test d'arrêt de récursivité. On arrête lorsque le prédécesseur de l'initiateur a été visité (on pourrait se contenter de s'arrêter lorsque l'initiateur reçoit l'appel récursif qu'il avait lui-même initialisé). Le paramètre *max* en entrée et en sortie, véhicule la valeur du plus grand identificateur rencontré jusque là, depuis le début du calcul. A tout instant, un seul exemplaire de la procédure est en cours d'exécution, sur un seul site, et la valeur de la variable *max* qu'il renvoie lors de la sortie de son appel récursif est celle du plus grand identificateur de processeur rencontré jusqu'à ce moment du calcul. Ainsi, lorsque tous les processeurs ont été visités, la valeur de la variable *max* est-elle le maximum des identificateurs des processus. Le fait que chaque identificateur soit unique implique que le *max* ainsi calculé désigne un processeur unique, qui est alors l'élu.

L'élection peut démarrer de n'importe quel site par activation de la procédure vrs_élection(ego, ego). La spécification de l'algorithme est alors :

```
ego : id_de_processeur;        -- Identificateur du site courant
succ : id_de_processeur ;      -- Successeur du processus
                                  -- courant sur l'anneau.
élu : id_de_processeur; -      -- processeur élu après terminaison

procedure vrs_élection (initiateur in , max in out:
                        id_de_processeur) is
```

```
begin
max := sup(max, ego) ;
-- Test de visite de tous les fils :
if succ ≠ initiateur then
        vrs_élection (initiateur, max) on succ
endif
-- L'élu est connu car tous les processeurs ont exécutés la
        -- procédure.
élu := max ;
end vrs_élection ;
```

4. Vague récursive sur un arbre couvrant

Après avoir traité le cas de la structure la plus simple, nous allons maintenant traiter de la vague récursive sur un ensemble de processeurs structurés en arbre couvrant (bien sûr il s'agit d'une structure virtuelle, et non physique).

La structure d'arbre couvrant est très utilisée en contrôle distribué lorsqu'il s'agit de collecter un état global ou de diffuser une information. Cette façon de faire induit une excellente utilisation des ressources en communication mais elle suppose qu'une topologie virtuelle d'arbre couvrant pré-existe et soit maintenue tout au long du calcul, ce qui suppose en fait que la topologie du réseau lui-même soit relativement stable. Pour construire un tel arbre couvrant, on peut utiliser les algorithmes [G,H,S 83], [LA, RO 86] ou [LA, LA 89].

4.1. Spécification générale

Chaque processeur connaît ici son père et ses fils dans l'arbre couvrant, seule la racine n'ayant pas de père et étant supposée initialiser le calcul (une modification simple de la procédure permettrait l'initialisation à partir de tout site, nous y reviendrons)

```
sons : ensemble id_de_processeur ;

procedure vague_ac ( <paramètres>) is
< déclarations> ;
i : id_de_processeur ;

begin
        < bloc d'instructions A > ;
        if sons ≠ empty_set  then
                < bloc d'instructions B > ;
                par  i  in sons do
                        <bloc d'instructions C > ;
                        vague_ac(f(<paramètres>))  on i ;
                        <bloc d'instructions D > ;
```

```
            enddo ;
        <bloc d'instructions E>
        endif ;
        < bloc d'instructions F > ;
    end vague_ac
```

4.2. L'algorithme récursif de diffusion

Comme exemple du style de programmation ainsi défini, nous donnons un protocole très simple de diffusion (sans gestion de fautes). En ce cas, la vague récursive sur l'arbre couvrant est uniquement utilisée afin d'envoyer des messages à un groupe de processeurs, en opérant un contrôle de flot. S'il n'y a pas de faute (franche) la vague revient à l'initiateur qui est alors assuré que tous les processeurs du groupe ont reçu et traité le message. Un nouveau message peut alors être envoyé.

Remarque 1 :
On obtient là une synchronisation dynamique sans introduire de nouvel outil algorithmique.

Remarque 2 :

Nous avons choisi de ne pas traiter dans ce rapport de la résistance aux fautes, néanmoins un traitement systématique des fautes franches est possible en utilisant un mécanisme d'exception comme celui de ADA.

```
sons : ensemble id_de_processeur ;

    procedure diffusion_ac ( message) is
    i : id_de_processeur ;

    begin
    -- Le traitement du message pourrait être fait à ce niveau (au début -- de la
procédure) mais il est préférable pour avoir des temps de
    -- réponse réalistes de lancer les traitements en parallèle.
    if sons ≠ empty_set then
        par i in sons do
            diffusion_ac (message) on i ;
        end do ;
    endif
    end diffusion_ac;
```

5. La vague récursive inondante

Dans les paragraphes précédents, nous avons vu comment mettre en œuvre une vague récursive sur les deux plus importantes topologies de contrôle distribué. Dans ces contextes, il est nécessaire qu'un précédent algorithme ait auparavant construit la-dite structure de contrôle sous-jacente, cette solution peut s'avérer efficace lorsque les topologies des réseaux considérés sont suffisamment stables.

Dans les autres cas, lorsque la mise en œuvre des algorithmes ici présentés est peu fréquente, ou lorsque la topologie du réseau est susceptible d'évolutions (par exemple dans des systèmes distribués orientés objets où des objets sont créés ou détruits dynamiquement) il y a nécessité de disposer d'autres outils. Afin de pouvoir traiter ce type de problème, la vague récursive que nous présentons maintenant construit dynamiquement sa propre structure de contrôle en utilisant une stratégie que nous qualifions "d'inondation". Nous appellerons donc cette façon de faire *vague récursive inondante*

Il y a maintes façons de contrôler une inondation par messages dans un réseau (voir [TAN 89]. Tannenbaum donne un schéma très général d'une telle inondation générant une infinité de messages. Par souci d'implémentabilité, et par cohérence avec la récursivité, nous ne donnerons ici que des algorithmes générant un nombre fini d'appels récursifs.

La distinction entre les différentes stratégies de contrôle des vagues récursives inondantes tient précisément à la façon dont on limite la récursivité, c'est à dire au critère de terminaison de la vague.

Une première stratégie consiste en une limitation directe du nombre d'appels récursifs (chaque message est alors transmis exactement n fois).

Une deuxième stratégie consiste à n'exécuter une procédure sur un processeur que la première fois qu'elle est activée, mais pas lors des autres réceptions de l'appel de la même vague.

Une troisième façon est de faire transporter par l'appel récursif la liste ordonnée des identificateurs des processeurs déjà visités, le

circulant (voir [LAV 90]), la vague récursive n'étant alors envoyée qu'aux processeurs dont l'identificateur n'est pas dans la liste.

On peut aussi utiliser une combinaison de ces différentes stratégies, c'est ce qui est fait par Lamport, Shotak et Pease dans leur algorithme [L, S, M 82] en vue de résoudre le problème des généraux byzantins. Ils utilisent là une combinaison de la première et de la troisième stratégie.

5.1. Spécification de la vague récursive inondante

Nous allons maintenant spécifier un algorithme général de vague récursive inondante en n'utilisant que la troisième stratégie de terminaison. Dans un tel cas, la vague récursive inondante construit elle-même, dynamiquement un arbre couvrant du graphe des processeurs.

```
   ego : id_de_processeur ;          -- déclaration de l'identificateur du
                                     -- processeur courant
   neighbours : ensemble id_de_processeur ;
                                     -- C'est l'ensemble des voisins du
                                     -- processeur courant

   procedure
Vague_récursive_inondante(processeurs_visités:ensemble
                          id_de_processeur; <paramètres>) is
   i : id_de_processeur ;
   processeurs_à_visiter : ensemble id_de_processeur;
                                     -- Processeurs suivants à visiter

   begin
   < bloc d'instructions A > ;       -- Les processeurs visités sont des voisins
                                     -- non déjà visités
   processeurs_à_visiter := voisins ;
                                     -- Si l'ensemble n'est pas vide, l'inondation
                                     -- peut continuer
   if processeurs_à_visiter ≠ empty_set then
            < bloc d'instructions B > ;
            par i in processeurs_à_visiter do
                  <bloc d'instructions C > ;
                  Vague_récursive_inondante_3 ( f(<paramètres>) )
on i ;
                  <bloc d'instructions D > ;
            end do ;
            <bloc d'instructions E>
      endif
      < bloc d'instructions F > ;
      end Vague_récursive_inondante ;
```

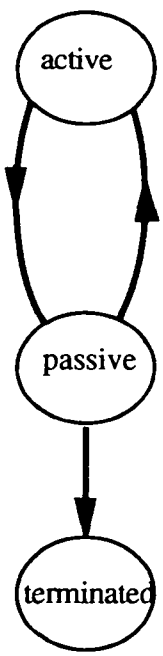
5.2. Un algorithme de détection de terminaison

Nous allons illustrer l'utilisation de la vague récursive inondante en résolvant un problème central de l'algorithmique distribuée, le problème de la terminaison.

5.2.1. La détection de terminaison

On suppose donné un système distribué à n processeurs, et pour chaque processeur deux activités : la première étant l'exécution d'un programme utilisateur, et la seconde, qui nous intéresse, la détection de la terminaison de la première.

L'algorithme utilisateur distribué est exécuté sur des processeurs utilisant des ressources locales et quelquefois envoie du travail à d'autres processeurs. Ainsi un processeur peut se trouver dans trois états : actif, passif, et terminé.



L'état passif est atteint localement par un processeur lorsqu'il a achevé son calcul en cours et que les messages qu'il a envoyé pour activer un APD ont été acquittés. Cette dernière condition peut être vérifiée en utilisant un système de compteurs, incrémentés lorsqu'on envoie un message, décrémenté lors de la réception d'un acquittement (à la manière de Dijktsra-Scholten [DI, SC 80]). On suppose que la seule façon pour un processeur d'être de nouveau activé est de recevoir un message d'activation. Un tel message ne peut être envoyé que par un autre processeur participant au calcul générateur des messages d'activation (i.e. Il s'agit d'un calcul distribué sans interaction avec le monde externe). Un processeur passe de l'état passif à l'état terminé lorsque tous les processeurs sont dans l'état passif.

L'algorithme de détection de terminaison peut être mis en œuvre par n'importe quel processeur du système distribué, dès lors qu'il s'est détecté comme étant lui même dans l'état passif, et voulant savoir s'il peut passer dans l'état terminé (on peut par exemple lancer la détection après avoir passé un certain temps en l'état

passif) ; par conséquent, plusieurs processeurs peuvent lancer simultanément un détection de terminaison. Un processeur saura qu'il a terminé dès lors qu'il saura que tous les autres processeurs ont fini et qu'il n'y a plus aucun message d'activation en transit dans le réseau.

Dans la terminologie de Halpern [HA, MO 90], lorsque tous les processeurs sont dans l'état passif, le niveau de connaissance est distribué (D-knowledge). A la fin de l'exécution de l'algorithme de terminaison, un processeur au moins a atteint l'état terminé. Il possède une connaissance exhaustive de l'état global. Cet algorithme accroît le niveau de connaissance en "quelqu'un sait" (somebody knows), noté K-knowledge. Enfin, une diffusion peut encore accroître le niveau de connaissance en "chacun sait" (everybody knows" noté E-knowledge. Nous ne nous intéressons dans ce qui suit qu'au passage de la "connaissance distribuée" à "quelqu'un sait". L'automate d'états de chaque processeur local peut alors être défini comme dans la figure ci-dessus.

Remarque 3 : Une solution du problème de la terminaison peut être utilisée pour construire des solutions à toute une classe de problèmes qui ne diffèrent que peut du problème initial, comme par exemple la détection de l'interblocage, ou plus généralement le problème de la causalité globale consistante, ou encore, dans les bases de données distribuées, le protocole de validation à deux phases (two steps commit protocol). Ainsi, on peut considérer l'état terminé comme la mise à jour d'une base de donnée distribuée, l'état actif correspondant à des mises à jour locales, et l'état passif correspondant à un manque d'activité locale. Il s'agit alors pour chaque site inactif de savoir s'il peut reporter les mises à jour qu'il a reçues.

5.2.2. Principe de la solution

- a) Initialisation

Lorsqu'un processeur prend l'initiative de détecter la terminaison, il exécute un appel local à la vague récursive inondante vri_terminaison. Le premier paramètre de la liste d'appel est un identificateur -unique- de vague généré à partir d'un compteur local de vague et de l'identificateur du processeur initiateur. Le deuxième

paramètre est utilisé afin d'opérer une petite optimisation qui consiste à ne pas renvoyer un appel récursif à son père dans l'arbre d'appels récursifs. Ainsi, pour l'appel lancé par l'initiateur, ce deuxième paramètre est un ensemble vide. Le troisième paramètre collecte les états (actif ou passif) de chaque processeur. L'algorithme détecte la terminaison en une seule vague. Si l'état collecté est passif, alors l'état global est terminé.

- b) Visite

Lorsqu'un appel d'activation de procédure, muni de son identificateur de vague, arrive en un processeur, il se peut que la même vague soit déjà en cours d'exécution sur le processeur en question, ou ce peut être une nouvelle vague. L'inondation doit être conduite de façon à collecter l'état des voisins. Si tous les voisins ont été détectés passifs, et si le processeur courant est resté passif durant la visite de la vague, alors l'état collecté est passif.

Remarque 4 : Aux fins d'optimisation, l'APD n'est pas envoyé par un processus à son père. Une solution plus optimisée encore éviterait à un processus d'envoyer la vague à tous ses ancêtres, c'est à dire à tous les processus qui sont sur le chemin qui le relie à l'initiateur.

La visite en cours d'un processeur par une vague est notée dans une structure de données de gestion de vagues. Quand la visite est finie (l'exécution de la procédure est terminée) l'entrée associée à la vague courante est effacée.

Ainsi, lorsqu'une vague atteint un processeur déjà en cours de visite par la même vague la procédure n'est pas activée. L'état collecté fournit la valeur courante de l'état local.

Remarque 5 : Le même processeur peut-être visité plusieurs fois par la même vague. Supposons que les processeurs P et Q soient visités parallèlement à partir d'un voisin commun, R. Si P est très rapide et exécute vri_terminaison sur Q très rapidement, Q démarre seulement sa première visite. Q répond en renvoyant sa valeur courante. Plus tard Q, qui est très lent, active vri_terminaison sur P et cette exécution s'exécute comme si P n'avait jamais été visité. Une

nouvelle visite commence alors. Une telle configuration a une très faible probabilité de survenance.

5.2.3. *L'algorithme*

- a) Gestion des vagues actives

Cette tâche sert au contrôle d'accès des vagues actives sur le processeur. Avec une sémantique de tâche de type ADA, les différentes entrées d'une tâche sont mutuellement exclusives, maintenant ainsi la cohérence de la structure de données. La structures de données *vagues_actives* peut être par exemple une liste chaînée d'enregistrements. Chaque enregistrement non vide est associé avec une vague en visite. Chaque enregistrement est composé d'un identificateur de vague et d'un booléen *cont_passif*. Ce booléen permet de vérifier qu'un processeur est resté continuement passif pendant la visite d'une vague. Il s'agit là d'un point important pour la correction de l'algorithme. Nous donnons donc une spécification informelle des différentes entrées.

```
task gestion_tâche_active is
    entry test ( vague_courante : id_vague ; visité : booleen ) ;
    entry efface (vague_courante : id_vague ) ;
    entry active ;
    entry passive ;
end gestion_tâche_active ;
```

test (vague_courante ; visité) effectue le test d'existence de la vague active par rapport à l'identificateur de vague courante. Si la vague courante est en cours de visite du processeur, il y a une entrée de la structure contenant l'identificateur de vague. La réponse du test(visité : booléen) est vraie. Sinon un enregistrement correspondant à la vague courante est créé dans la structure de données. L'identificateur de vague courante y est inscrit et l'indicateur *cont_passif* est initialisé à "vrai". La réponse du test(visité : booléen) est "faux" puisque la vague n'était pas en cours de visite du processeur.

efface(vague_courante : id_vague) efface l'enregistrement de vague courante dans la liste chaînée des *vagues_actives* . La valeur finale du booléen *cont_passif* est aussi donnée.

active doit être exécutée chaque fois qu'un processeur passif reçoit un message d'activation. *active* exécute deux opérations. Elle met la variable locale d'état, *état_local* à actif et met tous les *cont_passif* associés avec toutes les vagues en visite à *faux.*

passive doit être exécutée chaque fois qu'un processeur a terminé le travail associé à tous les messages d'activation qu'il avait précédemment reçu. Rappelons simplement que tous les messages d'activation envoyés par un processeur doivent être acquittés. La variable état_local est mise à la valeur passif.

- b) La structure de données de la vague inondante

```
ego : id_de_processeur ;   -- Identificateur du processeur courant
type vague_id is record    -- Un identificateur de vague est composé
                   -- de:
    initiator : site ;             -- Identificateur de processeur
    seq_number : integer ;         -- numéro de séquence (local)
end id_vague ;
voisins : ensemble id_de_processeur ;
                               -- Ce sont les voisins du processeur
                   -- courant
type  état_trav is  ( passif  ,  actif ) ;
                               -- actif et passif étant les deux états
                   -- locaux possibles
état_local : état_trav :=  passif ;
                               -- Variable d'état local.
```

- c) La procédure de vague récursive inondante

```
procédure vri_terminaison (
    vague_courante : id_vague ;  -- Identificateur de la vague.
    appelant : id_de_processeur ;        -- Identificateur de l'appelant.
état_collecté : état_trav ;              -- état collecté par la vague
                               ) is

voisins_à_visiter : ensemble id_de_processeur ;
    -- Voisins devant être visités.
état_voisins : array (voisins) of état_trav;
                                   -- Réponses des voisins

begin
                                   -- Si le processeur courant est actif, il est
                           -- préférable d'attendre jusqu'à ce qu'il
                   -- atteigne un état passif puisque l'algorithme
    -- est conçu pour détecter une quiescence                    -
```

```
générale des processeurs. On pourrait se                    -- contenter de
renvoyer un état courant actif                              -- sinon. Avec cette règle,
un processeur est                                           -- toujours passif en début de visite
par une                                                     -- vague.
   attente (état_local = passif ) ;     -- Teste si la même vague n'est pas déjà
                                        -- présente sur le processeur
   test( vague_courante , visité ) ;
   if not visité then
      voisins_à_visiter := voisins - appelant ;
                                                            -- évite de renvoyer l'APD
                                                            -- à l'appelant
      par i in voisins_à_visiter do
      vri_terminaison ( vague_courante , ego , état_voisins(i) ) ;
      end do ;
                                                            -- Si au moins l'un des voisins se déclare actif,
                                                            -- alors le processeur courant répond lui-même
                                                            -- qu'il est actif. Cette opération "et" est
                                                            -- exécutée après l'instruction par de façon à
      -- éviter les problèmes d'accès concurrents à                                          -
- la variable état_collecté.
      for i in voisins_à_visiter loop
            if (état_voisins(i) = active ) then
            état_collecté := active ;
            end if
      end loop
                                                            -- La vague courante termine sa visite.
                                                            -- L'identificateur de vague est effacé de la
                                                            -- structure de gestion des vagues actives Le
      -- booléen cont_passif indique si le processeura                                       -
- été -ou non- continuellement passif durant                                                 -- toute
la présence de la vague sur le site.
      efface ( vague_courante , cont_passif ) ;
      if ( not cont_passif ) then collected-state := active ;
      end if
   else
                                                            -- La vague est entrain de visiter un
                                                            -- processeur déjà en cours de visite par la
                                                            -- même vague. La valeur collectée est alors
                                                            -- égale à la valeur de l'état local.
      état_collecté := état_local ;
   end if
   end vri_terminaison ;
```

5.2.4. *Preuve de l'algorithme*

La preuve de cet algorithme sera conduite en deux étapes. Nous démontrons d'abord l'équivalence entre la terminaison globale et l'état passif collecté par une vague :

5.2.4.1. Théorème 1

Si l'algorithme distribué est terminé (i.e. chaque processeur est dans l'état passif et il n'y a aucun message en transit dans le réseau) alors l'état collecté lors de la terminaison d'une vague inondante récursive est l'état passif.

Preuve

A tout moment de l'exécution d'une vague récursive inondante, la structure virtuelle induite par les APD est un arbre dont la racine est l'initiateur. Lorsqu'un processeur reçoît la visite d'une vague, le processeur appelant de l'APD correspondant est le père dans l'arbre. La procédure est alors exécutée sur tous les autres voisins créant ainsi de nouvelles relations père-fils.

Il arrive un moment où un processeur f envoie un appel récursif à tous ses voisins v_f alors que ceux-ci sont déjà en cours de visite par la même vague (i.e. ils sont continuement passifs et sont bloqués en cet état après réception d'un APD). f est alors une feuille de l'arbre d'exécution des APD et reçoît la réponse de tous ses voisins qui ne sont pas son père. Cette réponse est l'état collecté passif. f renvoie alors à son père l'état collecté passif, et ainsi de suite lorsque le père de f a reçu toutes les réponses de ses fils.... Il y a au moins un processeur parent (i.e. le père du père du...) à la fois de f et v_f

- Dans le pire des cas ce parent est la racine de l'arbre d'appels récursifs.

Un processeur p dans la vague ne détecte son état passif ou actif qu'après avoir reçu toutes les réponses de ses fils. En particulier, soit p l'initiateur de vague, il est aussi la racine de l'arbre de récursivité, ainsi p reçoît les réponses des parents communs à f et v_f (voire de f et v_f eux-mêmes). Par conséquent, l'état du processeur p est donné par l'égalité suivante de point fixe :

$$\text{état}_p = \text{état}_p \bigwedge_{i \in \{\text{sons of } p\}} \text{état}_i \ ;$$

L'opérateur $\wedge$ étant transitif, si chaque processeur est dans l'état passif, en un temps fini, le processus p détecte son état passif, c'est à dire l'état passif de l'arbre dont il est la racine.

5.2.4.2. Théorème 2

Si l'état collecté est passif, tous les processeurs sont dans l'état passif et aucun message d'activation n'est en transit dans le réseau.

Preuve (ab absurdum)

Supposons qu'un état passif ait été collecté par un appel de la procédure vri_terminaison sur un processeur et qu'un autre processeur (soit p) soit encore actif dans le réseau. Ce processeur a nécessairement été activé par un message envoyé par un de ses voisins (soit Q), que ce soit avant, pendant, ou après la dernière visite de la vague de détection d'état passif.

- a) Il est impossible, par définition, que ce soit pendant car la gestion de la valeur de la variable booléenne cont_passif serait alors passée à faux, et par conséquent, l'état collecté ne peut être passif.

- b) Supposons maintenant que le message d'activation soit envoyé par le processeur Q antérieurement à la dernière visite de la vague. L'état collecté ne peut être passif que si le message a été entièrement traité par P (qui est alors repassé - toujours avant l'arrivée de la vague - dans l'état passif) avant l'arrivée de la vague en Q et que Q a reçu l'acquittement correspondant, ce message n'implique donc plus qu'un processeur P ou Q soit actif.

i - Si, durant ou après la dernière visite de la vague en Q, le message envoyé avant cette visite était entrain de transiter sur la ligne entre Q et P, alors il n'est pas acquitté. Donc, Q n'est pas passif et l'état collecté est : actif.

ii - Si le message d'activation envoyé de Q à P a été acquitté avant la visite de la vague en Q, deux situations peuvent survenir : soit P a fini le travail induit par le message d'activation issu de Q avant l'arrivée de la vague, soit ce travail s'est terminé durant l'exécution sur P de l'instruction attente(état_local = passif) qui est la première de l'APD. Dans les deux cas, ce message ne peut activer un processus **après** que la vague l'ait visité.

- c) Supposons que Q envoie un message d'activation à P après la visite de la vague en Q. Comme Q était passif, il ne peut envoyer de message d'activation qu'après avoir lui-même été activé par un tel message provenant d'un autre processeur (soit S) et ainsi de suite. De a) et b) on déduit qu'un tel message ne saurait avoir été émis ni avant, ni pendant la visite de la vague en S. Si un message d'activation survient après la dernière visite de la vague en S, c'est qu'il existe une chaîne (qui peut être de longueur quelconque) de messages

similaires d'activation dont aucun ne peut avoir été généré avant ou pendant la visite de la vague sur le site initiateur du message. C'est donc impossible car l'émission d'un tel message ne pourrait être qu'antérieure au passage de la dernière visite de la vague sur tous les processeurs de la chaîne et qu'il activerait un processeur après la visite de la vague.

5.3. La vague récursive inondante, solution au problème du routage par plus court chemin

Le problème des plus courts chemins d'un processeur à tous les autres est un problème majeur pour le contrôle des systèmes distribués. La résolution de ce problème est nécessaire pour pouvoir établir les tables de routage.

5.3.1. Le problème du plus court chemin d'un processeur à tous les autres

Considérons un système distribué constitué de n processeurs dont un initiateur e. Ce système distribué peut être vu comme un graphe. Les sommets du graphe sont alors les processeurs du système distribué et les lignes de communications, les arcs du graphe. Entre deux sommets, il y a donc deux arcs de sens contraire, pour les sommets a et b, on a les arcs (a, b) et (b, a). De plus à chaque arc est associé un coût, le coût de (a, b) pouvant bien sûr être différent de celui de (b, a). Cette situation semble la mieux adaptée dans le cas du routage, en effet, la charge de la file d'attente des tampons des messages en b en provenance de a, n'est pas nécessairement celle en a pour les messages provenant de b. Le problème est alors de construire une arborescence de plus courts chemins enraciné au nœud (i.e. sommet) e.

5.3.2. Dans le cas non orienté

On trouve pour l'essentiel, dans la littérature des algorithmes considérant le graphe comme non orienté, ce qui simplifie les choses, car dans ce cas on suppose implicitement que le coût associé à (a, b) est le même que celui associé à (b, a), ce qui représente un cas particulier. Cette simplification permet alors de construire un arbre de plus courts chemins enraciné en e dans lequel, à la fin du calcul, chaque sommet connaît son *père* dans l'arborescence, mais

pas ses fils. Dans ce contexte, il suffit alors de lancer l'algorithme de plus courts chemins à partir de tous les sommets pour que, par l'argument de symétrie induit par les arètes, tout sommet connaisse tous les plus courts chemins à tout autre sommet.

l'algorithme s'écrirait très simplement récursivement dans ce cas :

a) Variables globales

```
ego : identificateur                -- Identificateur du processeur
                                    -- courant
voisins : setof identificateur ;
coûts_des_voisins : array(voisins) of coût
                                    -- coût pour atteindre le voisin
lmin : coût := max_integer;         -- valeur du meilleur chemin
connu
père : identificateur ;             -- identificateur du père dans le
plus                                -- court chemin
```

b) Tâche gérant l'accès aux variables partagées sur un processeur

```
task gérant_de_données_communes is
entry teste_et_applique (valeur_chemin, appelant, bin : boolean :=
0)

begin
   loop
         accept teste_et_applique(valeur_chemin, appelant, bin)
do
               if valeur_chemin< lmin then
                     père := appelant ;
                     lmin := valeur_chemin ;
                     bin := 1 ;
               endif
         end teste_et_applique
   endloop ;
end gérant_de_données_communes ;
```

c) Algorithme récursif de plus court chemin pour graphe non orienté

```
procedure chemin (valeur_chemin : coût, appelant :
                              identificateur) is

begin

         teste_et_applique(valeur_chemin : coût, appelant, bin)
         if bin then
               par i in voisins do
                     chemin (lmin + coûts_des_voisins , ego) on i
```

```
            end do
        endif
end chemin
```

On remarquera que dans tous les algorithmes qui résolvent ce problème, les processus sont supposés exécutés en exclusion mutuelle sur chaque site (dans [CH, MI 82] comme dans [ME, SE 79]). Ici, sur chaque site physique, plusieurs exemplaires du même processus peuvent être en cours d'exécution simultanément, seul l'accès à la variable globale partagée (sur un site) *père* doit se faire en exclusion mutuelle

5.3.3. Avec prise en compte de l'orientation

Tous les coûts associés aux arcs sont supposés positifs, dans le cas de l'établissement de tables de routage, des coûts négatifs affectés à des arcs ne pourraient avoir de sens que dans des cas bien particuliers. La structure de vague récursive fournit alors un canevas pour construire un algorithme distribué.

a) les variables globales

voisins : est un ensemble dont les éléments sont les identificateurs des processeurs directement connectés au processeur courant ;

coûts_voisins : est un vecteur indexé par les identificateurs contenus dans *voisins* et tel que coûts_voisins(j) représente le coût attribué à l'arête (i.e. la liaison) entre le processeur courant et le processeur j ;

lmin : étant la longueur, jusqu'ici calculée du plus court chemin de l'initiateur au processeur courant ;

père : variable de type identificateur contenant l'identité du prédécesseur du processeur courant dans l'arbre des plus courts chemins ;

ego : variable entière contenant l'identificateur du processeur courant ;

fils : vecteur caractéristique indexé par *voisins* et tel que fils(j) = 1 signifie que j est fils du processeur courant dans l'arbre des plus courts chemins.

b) la tâche de gestion des données

Cette tâche a trois entrées exécutées en exclusion mutuelle aux fins de préservation de la consistance des variables globales.

modifie : Cette entrée permet, après une amélioration de la valeur courante de plus court chemin, de modifier la relation fils-père, et d'enregistrer le nom du nouveau père et ceux des nouveaux fils.

change_fils : Cette entrée est appelée par le processeur courant sur son père lorsqu'il s'agit précisément de signifier que le processeur courant va changer de père. Elle efface l'information suivant laquelle le processeur appelant est fils du processeur appelé. Cette modification ne peut être faite que si la valeur courante minimale de chemin n'a pas changé entre temps. Les paramètres d'entrée de la procédure sont donc l'identificateur de l'appelant et la valeur du chemin pour laquelle l'appelant était fils de l'appelé.

Cette procédure change_fils introduit une nouveauté dans les vagues récursives dans la mesure ou elle permet de modifier des paramètres dans une procédure appelante, autrement que par la sortie de récursivité classique.

teste_et_applique : Cette entrée teste si une nouvelle activation de la procédure induit une amélioration de la valeur du chemin. Si oui, elle positionne une variable binaire *bin* à 1, enregistre la nouvelle valeur de plus court chemin dans la variable globale *lmin*, et enregistre également cette valeur dans une variable locale de la procédure appelée, soit *laux* cette variable.

Dans le graphe des processeurs, e est l'initiateur et par conséquent aussi la racine de l'arbre des plus courts chemins.

```
task gérant_de_données_communes is
    entry modifie (nouveau_père, fils_local, valeur_père);
    entry change_fils (ego)
    entry teste_et_applique(valeur_chemin, lmin, laux, bin)

begin
```

```
    loop
    select
    accept modifie(nouveau_père,fils_local, valeur_père) do
          if père ≠ nil and père ≠ nouveau_père then
                change_fils (ego, valeur_père) on nouveau_père
          endif
          père := nouveau_père ;
          for i in voisins  loop
                fils (i) := fils_local(i)
          endloop
    end modifie;
    accept teste_et_applique(valeur_chemin, lmin, laux, bin)
          if valeur_chemin < lmin then
                lmin := valeur_chemin ; bin := 1 ; laux := lmin
          endif
    end teste_et_applique
    accept change_fils (i : identificateur , valeur_père) do
          if valeur_père = lmin then fils (i) := 0
    end change_fils

    endselect ;
    end loop;
end gérant_de_données_communes;
```

c) Procédure principale

Les variables locales sont les suivantes :

appelant : identificateur de l'appelant.

fils_local : vecteur caractéristique indexé par *voisins* tel que fils_local(j) = 1 quand j est fils du processeur courant dans le chemin en construction.

Les paramètres d'appel sont :

bin : variable booléenne initialisée à 0 et mise à 1 après changement de père par le processeur courant.

valeur_chemin valeur du chemin au processeur courant.

valeur_père valeur du chemin au processeur appelant.

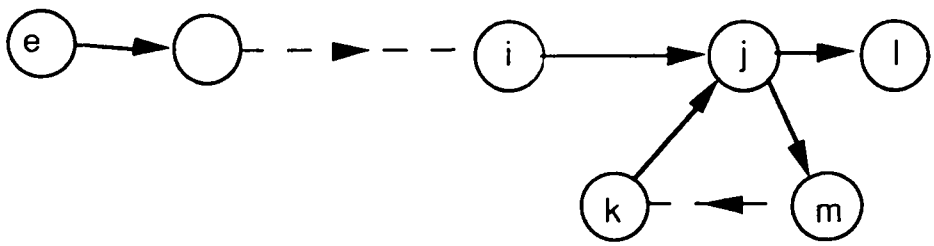
appelant identificateur du processeur appelant.

```
voisins  : setof identificateur ;
coût_ voisins  : array (voisins ) of cost  ;
ego , père : identificateur   ;
fils  : array ( voisins ) of boolean ;
```

procedure chemin1 (valeur_chemin , valeur_père : coût, appelant : identificateur , bin : **boolean**) **is**

```
fils_local : array ( voisins ) of boolean ;
laux : coût ;

begin
    bin := 0 ;
    teste_et_applique(valeur_chemin, lmin, laux, bin) ;
        if bin then
            par i in voisins - appelant do
                chemin (valeur_chemin + coût _voisins (i),
                    valeur_chemin, ego, fils_local(i)) on i
            end do
            if laux = lmin then
                modifie (appelant , fils_local, valeur_père)
            endif
        endif
end chemin1
```



Reprenons le graphe de la figure ci-dessus. Ici, e est l'initiateur et sera donc la racine de l'arbre des plus courts chemins de e à tous les autres. On peut dire alors qu'un processus sort de son exécution récursive dans deux cas :

- quand l'appel est effectué avec des paramètres n'induisant pas de modification de la valeur de l ;

- quand tous les APD activés sur tous ses voisins sont terminés.

Nous introduisons ici deux modifications importantes par rapport à l'algorithme de Chandy & Misra ;

i l'algorithme ne fonctionne qu'en une seule phase, en effet lorsque tous les APD activés par e sont terminés, l'arbre des plus courts chemins est formé (ceci est dû au fait que, contrairement à Chandy et Misra, on ne considère pas les coûts négatifs. Dans ce cas, nous avons précédemment donné une autre solution au problème sous les mêmes hypothèses voir [LAV 90]) ;

ii à tout moment du calcul tout processeur connaît son père et ses fils dans l'arbre de chemins déjà construit. En particulier,

lorsque e a terminé, les variables globales *père* et *fils* contiennent respectivement le nom du processeur père et ceux des processeurs fils dans l'arbre des plus courts chemins issus de e. Or tel n'est pas le cas dans l'algorithme de Chandy & Misra dans lequel il faudrait introduire un traitement supplémentaire dans la phase 2.

5.3.3.2. Schéma de preuve

A la fin du calcul, chaque processeur non-initiateur connaît ses successeurs et son prédécesseur (i.e. ses fils - par le vecteur fils - et son père -par la valeur de la variable père -) dans l'arbre des plus courts chemins.

L'idée de base de cet algorithme est que la vague ne se propage à partir du site courant que si la valeur du plus court chemin déjà connue est améliorée par la vague, sinon cette dernière est stoppée. Par conséquent, tant qu'il y a des améliorations de valeurs des chemins, il y a propagation de la vague, et dès lors qu'il n'y a plus d'amélioration, la vague se termine. Le nombre de processeurs dans le système étant fini, le nombre d'arètes l'est aussi. Comme il n'existe aucune chaîne infinie (i.e. ici un cycle) telle que la somme des coûts des arètes le long de cette chaîne décroisse indéfiniment (cette propriété est due au fait que tous les coûts sont positifs et qu'on cherche un coût minimal), la convexité de l'ensemble des coûts est assurée et le principe suivant lequel dans un ensemble convexe, une stratégie optimale est obtenue comme combinaison convexe de sous stratégies optimales peut s'appliquer.

6. Conclusion

Nous avons introduit ci-dessus quelques schémas simples de programmation récursive. Pour chaque schéma, nous avons donné des exemples d'utilisation dans des cas bien connus. Les algorithmes ainsi décrits sont pour certains construits sur les mêmes idées que ceux utilisant les communications explicites par messages. Il peut arriver parfois que la différence de spécification, de programmation, induise des exécutions légèrement différentes pour un algorithme qui, fondamentalement reste le même.

Pour les algorithmes de plus court chemin, à notre connaissance, cette façon d'aborder le problème est originale, et l'écriture récursive permet de gérer la simultanéité d'exécution de plusieurs instances d'une même vague sur un même site, d'une façon sommes toutes relativement simple. L'introduction d'une procédure du type change_fils induit une particularité dans la récursivité distribuée qui est que différentes exécutions de la même procédure dans la même vague, sur des sites différents peuvent communiquer entre elles. Cette façon de faire montre bien la différence conceptuelle qu'il y a dans l'utilisation de la récursivité suivant qu'on se trouve dans un contexte séquentiel ou parallèle ou distribué.

Les avantages essentiels de la solution récursive sont, le haut degré d'abstraction (plus besoin de gérer précisément les envois-réception de messages), la manière plus naturelle de construire une solution, la meilleure lisibilité. Souvent, la solution des problèmes de programmation distribuée utilise une approche du type client-serveur, la plupart du temps, synchrone. Dans ces cas, la solution récursive peut être construit au même coût d'échange d'information La gestion des pannes franches est beaucoup plus accessible par utilisation d'un mécanisme du type exception.

La récursivité apparaît comme étant un concept unificateur en programmation distribuée de contrôle et une extension naturelle du présent travail sera d'étendre ce concept à une classe de problèmes de contrôle la plus large possible, et aussi à des problèmes de calcul de même, nous travaillons à l'extension de cette méthode dans le cas d'un environnement parallèle (i.e. à mémoire partagée).

Bibliographie

[BAN 90] **Banâtre J.-P.**, La programmation parallèle, outils méthodes et éléments de mise en œuvre.

[CH, MI 82] **Chandy K.M., Misra J.**, "Distributed computations on graph : Shortest chemin algorithm", CACM, vol. 25, n°11, novembre 1982, pp. 833-837.

[CHA 82] **Chang E.J.**, Echo Algorithms, IEEE Trans. on Software eng., vol SE 8, n°4, juillet 1982.

[DI,SC 80] **Dijkstra E.W., Scholten C.S.**, Termination detection for diffusing computations, I.P.L., August 11-1980, pp. 217-219.

[HA,MO 90] **Halpern J.Y., Moses Y.**, Knowledge and Common Knowledge in a distributed environment, J.A.C.M., Vol.37, n°3, pp. 549-587, July 90.

[HE,RA 89] **Hélary J-M., Raynal M** , Un schéma abstrait d'itération répartie, T.S.I., vol. 3, pp. 259-268, 1989.

[L,S,P 82] **Lamport L., Shostak R., Pease M.**, The Byzantine Generals Problem, ACM Toplas, Vol.4, n°3, juillet 1982, pp. 382-401.

[LAV 90] **Lavallée I.**, Algorithmique parallèle et distribuée, Hermès ed., Paris, 1990.

[MAT 87] **Mattern F.**, Algorithms for Distributed termination detection, Distributed Computing (1987) 2 : pp.161-175.

[ME, SE 79] **Merlin et Segall**, A failsafe distributed routing protocol, *IEEE Trans. on Comm. vol. 27, n° 9*, Septembre 1979, pp. 1280-1287.

[TAN 89] **Tannenbaum**, Computer networks, Prentice Hall, 1989.

ISSN 0249 - 6399