À propos de calcul réparti : un de mes algorithmes préférés

Michel Raynal [1]

Le calcul réparti : de quoi s'agit-il ?

L'informatique et ses applications ne se réduisent pas à l'intelligence artificielle, à l'apprentissage, aux simulations ou à la blockchain, etc. Le monde est réparti ; de plus en plus d'applications sont des applications réparties et au cœur de celles-ci, il y a les algorithmes répartis.

Calcul réparti L'algorithmique répartie naît lorsque l'on a résoudre un problème en termes d'entités physiquement réparties (entités appelées processus, processeurs, agents, capteurs, acteurs, pairs, clients, serveurs, etc.) connectées par un medium de communication. De plus, chaque entité a ses propres (paramètres d') entrées, qui sont inconnues des autres entités, alors que la solution au problème met *a priori* en jeu l'ensemble de ces entrées. Dans la suite, le terme « processus » sera utilisé pour parler de ces entités de calcul. Le medium de communication peut être une mémoire partagée (comme dans les multi-cœurs) ou un réseau à passage de messages.

Une des difficultés principales du calcul réparti provient de ce qu'il est convenu d'appeler son « environnement ». Ce dernier représente le contexte dans lequel le calcul s'exécute. Celui-ci peut être synchrone ou asynchrone. Dans le premier cas, les processus avancent au pas cadencé, alors qu'il n'y a pas d'hypothèse sur les vitesses des processus dans un contexte asynchrone (le temps physique est alors une ressource nécessaire au calcul mais ne participe pas à sa sémantique). Il y a, ensuite,

1. Université de Rennes IRISA, Inria, CNRS, France, Department of Computing, Polytechnic University, Hong Kong

les types de fautes que peuvent commettre les processus : les plus communément considérées étant le crash (arrêt inopiné et définitif du processus) et les fautes byzantines (le comportement d'un processus ne correspondant alors pas au code défini par l'algorithme qu'il est censé exécuter [17]).

Ainsi, asynchronisme, fautes, mobilité, hétérogénéïté, etc., constituent autant d'« adversaires » auxquels un calcul réparti est soumis et auxquels il doit faire face. Ces adversaires génèrent un non-déterminisme dont la maîtrise constitue l'essence du calcul réparti. En fait, dans un calcul réparti, l'environnement du calcul constitue lui-même l'une de ses entrées, entrée qui n'est sous le contrôle d'aucun processus mais que chacun d'eux subit. Alors que le calcul parallèle a pour but l'efficacité, et que le calcul dit « temps réel » considère le temps physique comme un objet de programmation, le but du calcul réparti réside dans la maîtrise de l'incertitude créée par son environnement. Bien sûr, une application répartie particulière peut mettre en jeu des aspects temps réel et des aspects répartis, mais les concepts et techniques pour les adresser ne sont pas de même nature.

Calcul réparti versus calcul parallèle Il existe parfois, dans la littérature, une confusion entre calcul parallèle et calcul réparti [27]. L'essence du calcul parallèle consiste en la décomposition d'un problème en sous-problèmes indépendants, de façon à pouvoir exploiter cette indépendance, en résolvant ces sous-problèmes simultanément sur des processeurs différents. Il s'agit d'une activité non-triviale qui met en jeu des concepts et des techniques appropriés (techniques spécifiques d'analyse de programmes, algorithmes d'ordonnancement, langages de programmation idoines, etc.).

Dans le calcul parallèle, la décomposition d'un problème en sous-problèmes est un choix de programmation alors que, dans un contexte réparti, la distribution des entités de calcul est imposée au programmeur. Exprimé de façon différente, tout problème résolu par un calcul parallèle pourrait être résolu par un calcul séquentiel (au facteur d'efficacité près), alors qu'il existe des problèmes de calcul réparti qui n'ont pas d'équivalent séquentiel : par exemple, la détection de la terminaison.

Alors que le calcul parallèle consiste à rendre les « choses » indépendantes, le calcul réparti consiste à permettre à des entités de calcul pré-existantes de se coordonner (en présence d'adversaires) de façon à ce que chaque entité puisse calculer son résultat local (qui dépend de l'ensemble des entrées et de l'environnememt imposé au calcul). Alors que le mot « indépendance » peut être naturellement associé au calcul paralléle, le mot « coordination » peut l'être au calcul réparti.

Le problème de la vue instantanée

Afin d'illustrer les notions vues précédemment, cette section considère un problème simple et non-trivial, à savoir le calcul d'une « vue instantanée ». Ce probléme a été posé et résolu en 1993 par Elisabeth Borowski et Eli Gafni [4]. Introduit pour résoudre un problème réparti particulier (le renommage de processus), le concept de vue instantanée s'est révélé crucial à la fois pour montrer des équivalences de modèles de calcul réparti (voir par exemple [22]) et comme brique de base pour construire des objets répartis de plus haut niveau d'abstraction (voir par exemple [25]).

Modèle de calcul et définition de la vue instantanée

Modéle de calcul Le système considéré est formé de n processus séquentiels dénotés p_1, ..., p_n. L'index i constitue l'identité de p_i. Chaque processus est asynchrone au sens où chacun progresse à sa propre vitesse qui peut varier avec le temps et reste toujours inconnue des autres processus. Un nombre quelconque de processus peut crasher (on rappelle que le crash est un arrêt inopiné et définitif).

La communication se fait via une mémoire partagée, constituée de registres, que les processus ne peuvent que lire ou écrire. Ces registres sont atomiques au sens où tout se passe comme si les opérations de lecture et d'écriture sur chaque registre R avaient été exécutées les unes après les autres, définissant une séquence S_R telle qu'une opération op_1 sur R, qui se termine avant qu'une opération op_2 sur R ne débute, apparaît avant op_2 dans S_R, et toute opération de lecture de R retourne la dernière valeur écrite dans R (« dernière » au sens de la séquence S_r) [18]. Une telle séquence est « appelée linéarisation du registre » R [13].

Vue instantanée : définition Le problème de la vue instantanée est présenté, ici, comme la construction d'un objet réparti. Cet objet est défini par une seule opération dénotée write_snapshot() qu'un processus ne peut appeler qu'une seule fois [2]. Lorsqu'un processus p_i invoque l'opération write_snapshot(), il lui fournit une valeur v_i en entrée, qui est déposée dans l'objet, et obtient en résultat un ensemble de paires dénoté $view_i$. Une paire est constituée d'une identité de processus j et de la valeur v_j déposée par p_j.

L'objet est défini par les propriétés suivantes (ce qui signifie que toutes les exécutions d'un algorithme qui met en œuvre l'objet doit les satisfaire).

— Terminaison : l'invocation de write_snapshot() par un processus qui ne crashe pas termine.

— Auto-inclusion : si l'invocation de write_snapshot(v_i) par p_i lui retourne $view_i$, on a $\langle i, v_i \rangle \in view_i$.

2. Il est possible d'étendre la définition au cas où les processus peuvent appeler write_snapshot() un nombre quelconque de fois.

— Ordre : si les invocations de write_snapshot(v_i) par p_i et write_snapshot(v_j) par p_j leur retournent $view_i$ et $view_j$, on a $view_i \subseteq view_j$ ou $view_j \subseteq view_i$.
— Instantanéité : si l'invocation de write_snapshot(v_i) par p_i lui retourne $view_i$ et $\langle j, v_j \rangle \in view_i$, on a $view_j \subseteq view_i$.

La propriété de terminaison est parfois appelée « sans attente » (*wait-freedom* [11]). Elle stipule que si le processus qui l'invoque ne crashe pas durant l'invocation, l'opération write_snapshot() doit terminer quel que soit le comportement des autres processus.

Les trois autres propriétés définissent la cohérence mutuelle des ensembles rendus en résultats. L'auto-inclusion indique que tout processus voit sa participation à l'objet, la propriété d'ordre indique que les ensembles résultats sont totalement ordonnés par inclusion, et la propriété d'instantanéité indique si un processus p_i voit la valeur déposée par un processus p_j, il voit tout ce que p_j voit.

Contrairement aux objets traditionnels issus du calcul séquentiel tels que les piles ou les files, l'objet vue instantanée n'a pas de spécification séquentielle, ce qui veut dire qu'il n'est pas possible de décrire tous ses comportements corrects à l'aide de séquences. En fait il est facile de montrer que la propriété d'instantanéité peut être ré-écrite de la façon suivante :

$$(\langle i, v_i \rangle \in view_j) \wedge (\langle j, v_j \rangle \in view_i) \Rightarrow (view_i = view_j).$$

D'où l'on conclut que si $view_i = view_j$, les invocations de write_snapshot(v_i) par p_i et write_snapshot(v_j) par p_j apparaissent comme si elles s'étaient produites simultanément (il n'y a aucun moyen de placer l'une avant l'autre sans violer la spécification de l'objet). [3]

Un algorithme (réparti) pour la vue instantanée

L'algorithme décrit à la figure 1 construit un objet réparti « vue instantanée ».

Mémoire partagée Celle-ci est constituée de deux tableaux tels que seul p_i peut écrire dans l'entrée i de chacun d'eux, alors que tout processus peut en lire toutes les entrées. Leurs identificateurs sont en lettres majuscules.

— $VALUES[1..n]$ est un tableau initialisé à $[\bot, \cdots, \bot]$ où $\bot$ est une donnée de contrôle indiquant que l'entrée correspondante n'a pas encore été écrite. $VALUES[j]$ contiendra ensuite la valeur déposée par le processus p_j dans l'objet « vue instantanée ».
— $LEVEL[1..n]$ est un tableau initialisé à $[n+1, \cdots, n+1]$. L'entrée $LEVEL[i]$, en décroissant par pas de un, va permettre au processus p_i de progresser dans le calcul de $view_i$ en fonction de la progression des autres processus, comme expliqué ci-dessous.

3. Il y a de nombreux objets répartis qui n'ont pas de spécification séquentielle. Un des exemples les plus connus est la validation atomique (*atomic commit)* en présence de fautes par crash de processus [6, 28].

```
operation write_snapshot(v_i) is
(L1) VALUES[i] ← v_i ;
(L2) repeat LEVEL[i] ← LEVEL[i] − 1 ;
(L3)        for each j ∈ {1,...,n} do level_i[j] ← LEVEL[j] end for ;
(L4)        set_i ← {x | level_i[x] ⩽ level_i[i]}
(L5) until (|set_i| ⩾ level_i[i]) end repeat ;
(L6) let view_i = { ⟨x, VALUES[x]⟩ | x ∈ set_i } ;
(L7) return(view_i)
end operation.
```

FIGURE 1. Algorithme qui réalise l'operation write_snapshot() (code du processus p_i)

Mémoires locales Chaque processus p_i gère une variable locale set_i et un tableau $level_i[1..n]$ qui lui sert à mémoriser la valeur lue de $LEVEL[1..n]$. Leurs identificateurs sont en lettres minuscules, indexés par l'identité du processus.

Principe sous-jacent et description de l'algorithme Considérons le tableau $LEVEL[1..n]$ comme une structure de données mesurant la progression des processus descendant (de manière asynchrone) un escalier de $n+1$ marches. Initialement, tous les processus sont à la marche $n+1$, ce qui est codé dans l'initialisation du tableau $LEVEL[1..n]$.

Quand un processus p_i invoque write_snapshot(v_i), il dépose sa valeur v_i dans $VALUES[i]$ (ligne L1), puis rentre dans une boucle dans laquelle il commence par descendre à la marche suivante (ligne L2). Il lit ensuite le contenu du tableau $LEVEL[1..n]$ (ligne L3), tableau dans lequel sont consignées les progressions des autres processus. Il est important de remarquer que les lectures des entrées de $LEVEL[1..n]$ se font une à une de façon asynchrone (en d'autres termes la lecture du tableau n'est pas atomique). Avec les valeurs approchées qu'il a obtenues sur la progression des autres processus, le processus p_i calcule ensuite l'ensemble set_i des processus qu'il perçoit à une marche de numéro plus petit ou égal au numéro de la marche à laquelle il se trouve lui-même (ligne L4). Si cet ensemble contient un nombre de processus inférieur au numéro de la marche à laquelle il se trouve, p_i progresse au pas suivant de l'itération (le prédicat de la ligne L5 est alors faux) dans laquelle il descendra à la marche suivante (ligne L2). Dans le cas contraire, p_i voit qu'un nombre de processus égal ou supérieur au numéro de la marche $level_i[i]$ à laquelle il se situe (prédicat de la ligne L5) ont progressé jusqu'à une marche inférieure ou égale à la sienne (ligne L4). Le processus p_i retourne alors en résultat l'ensemble de paires mettant en jeu ces processus (lignes L6 et L7). On dit alors que le processus p_i s'est arrêté à la marche de numéro $level_i[i]$.

D'un point de vue plus formel (et donc plus précis), en dépit de l'asynchronisme de chaque processus et du crash possible d'un nombre quelconque d'entre eux, l'algorithme maintient invariante la propriété suivante :

$$\forall x \in \{1, \cdots, n+1\}, \text{ au plus } x \text{ processus s'arrêtent aux marches } y \leqslant x.$$

Illustration avec deux cas extrêmes Le lecteur intéressé pourra trouver des démonstrations de cet algorithme dans [4, 25]. Pour en comprendre le fonctionnement plus en détail, nous explicitons ici deux cas extrêmes en ce qui concerne la dimension « asynchronisme ». Dans les deux cas, on considère que k processus invoquent write_snapshot() et qu'il n'y a pas de crash.

Le premier cas est celui d'une exécution séquentielle. Les k processus $p_{i_1}, \cdots, p_{i_k}$ invoquent l'opération write_snapshot() l'un après l'autre, à savoir $p_{i_{x+1}}$ n'invoque write_snapshot($v_{i_{x+1}}$) qu'après que p_{i_x} ait retourné sa vue. Il est facile de voir que, dans ce cas, p_{i_1} descend toutes les marches de l'escalier, stoppe à la marche 1, et retourne la vue $\{\langle i_1, v_{i_1}\rangle\}$. Lorsque, ensuite, p_{i_2} invoque write_snapshot(v_{i_2}), marche après marche, il descendra de la marche $n+1$ jusqu'à la marche 2 avant de retouner la vue $\{\langle i_1, v_{i_1}\rangle, \langle i_2, v_{i_2}\rangle\}$; et ainsi de suite jusqu'à p_{i_k} qui descendra jusqu'à la marche k avant de retouner la vue $\{\langle i_1, v_{i_1}\rangle, \cdots, \langle i_k, v_{i_k}\rangle\}$.

Le second cas est celui d'une exécution synchrone. Les k processus $p_{i_1}, \cdots, p_{i_k}$ exécutent conjointement les lignes L1 et L2. Une fois ces lignes terminées, ils exécutent conjointement les lignes L3 et L4. Il est facile de constater qu'ils obtiennent tous les mêmes valeurs dans le tableau $LEVEL[1..n]$ qui est tel que $|\{x : LEVEL[x] \leqslant n\}| \leqslant k$. Si $k = n$, tous les processus retournent alors l'ensemble $\{\langle i_1, v_{i_1}\rangle, \cdots, \langle i_k, v_{i_n}\rangle\}$. Si $k < n$, les k processus passent à l'itération suivante dans laquelle ils exécutent conjointement les lignes L2 et L3, etc. Ceci jusqu'au moment où les k processus se retrouvent sur la même marche k ; chacun retourne alors $\{\langle i_1, v_{i_1}\rangle, \cdots, \langle i_k, v_{i_k}\rangle\}$.

Courte discussion La vue instantanée est en fait un problème de la famille des problèmes d'accord réparti : chaque processus possède une valeur d'entrée et l'ensemble des processus doivent se mettre d'accord sur les valeurs qu'ils rendent en résultat, chaque type d'accord définissant un problème spécifique.

En plus de ses utilisations comme brique de base (à la manière des piles en calcul séquentiel par exemple) pour construire des abstractions réparties de plus haut niveau, l'algorithme de vue instantanée possède d'autres qualités de première classe, telles que définies dans [1], à savoir concision, non-trivialité, efficacité et élégance [4].

4. Le lecteur intéressé pourra trouver une communauté d'esprit dans la conception de cet algorithme et celle de l'algorithme d'exclusion mutuelle dû à Peterson [21]. Tous deux sont fondés sur la même intuition (descente d'un escalier marche après marche) et partagent les mêmes qualités de concision et d'élégance.

Une expression répartie récursive Dans les algorithmes récursifs séquentiels ou parallèles, le paramètre de récursivité est habituellement associé à la structure des données (arbre, tableau, graphe, etc.) utilisée par l'algorithme. Il apparaît que, dans le calcul réparti récursif, ce paramètre est lié au degré de concurrence perçu par chacun des processus [10, 23, 28]. La récursivité est dirigée par la concurrence et non par la structure des données. L'algorithme précédent peut ainsi être ré-écrit dans une formulation récursive, comme indiqué dans la figure 2 dans laquelle x, tel que $n \geqslant x \geqslant 1$, est le paramètre de récursivité. Un processus invoque rec_write_snapshot(n, v_i).

```
operation rec_write_snapshot(x, v_i) is
(R1)  VALUES[x][i] ← v;
(R2)  for each j ∈ {1,...,n} do values_i[j] ← VALUES[x][j] end for;
(R3)  poss_view_i ← { ⟨j, values_i[j]⟩ | values_i[j] ≠ ⊥ };
(R4)  if (|poss_view_i| = x) then view_i ← poss_view_i
(R5)                         else view_i ← rec_write_snapshot(x − 1, v_i)
(R6)  end if;
(R7)  return(view_i)
end operation.
```

FIGURE 2. Une formulation récursive de l'opération write_snapshot() (code du processus p_i)

Le tableau $VALUES[1..n]$ est maintenant un tableau de vecteurs de taille n dont toutes les entrées $VALUES[x][j]$ sont initialisées à $\perp$. Lors du x-ième appel récursif, le processus p_i utilise le vecteur $VALUES[x]$. Il écrit dans $VALUES[x][i]$ la valeur v_i qu'il veut déposer dans la vue instantanée (ligne R1). Il lit ensuite (de façon asynchrone) les valeurs déposées dans $VALUES[x]$ (ligne R2) et calcule une vue possible pos_view_i résultant de ces lectures (ligne R3). Si cette vue contient exactement x paires, p_i voit alors exactement x processus ayant exécuté le x-ième appel récursif. Dans ce cas, la vue est correcte (du point de vue de l'ordre sur les vues) et p_i la retourne en résultat (lignes R3 et R7). Dans le cas contraire, p_i a vu moins de x processus ayant participé au x-ième appel récursif. Il invoque alors rec_write_snapshot$(x-1, v_i)$ et retournera la vue calculée par son dernier appel récursif (lignes R5 et R7).

Le lecteur, qui aura remarqué que le niveau de récursivité x correspond à la marche x de l'escalier dans l'expression non-récursive de l'algorithme, est invité à faire tourner l'algorithme récursif sur les deux cas extrêmes qui ont précédemment servis d'illustration. Une généralisation de la vue instantanée et sa relation, du point de vue de la calculabilité répartie, aux problèmes d'accord réparti sont présentées dans [7].

Du possible et de l'impossible en réparti

Le problème du consensus Le problème du consensus est le problème d'accord réparti, fondamental tant du point de vue de la calculabilité répartie que du point de vue des applications, fondées sur la répartition de machines d'état.

Le problème consiste à construire une opération appelée proposer(), que chaque processus peut appeler une fois. Cette opération prend une valeur en paramètre d'entrée (appelée « valeur proposée ») et rend en résultat une valeur (appelée « valeur décidée »). Considérant le modèle de fautes par crash de processus, résoudre le problème consiste à écrire un algorithme pour l'opération proposer() qui satisfait les propriétés suivantes :

— Terminaison : toute invocation de proposer() par un processus qui ne crashe pas termine.
— Validité : si un processus décide v, la valeur v a été proposée par un processus.
— Accord : deux processus ne peuvent pas décider des valeurs différentes.

Le terminaison indique qu'un processus qui ne crashe pas doit terminer quel que soit le comportement des autres processus (*wait-freedom* [11]). La validité relie les sorties aux entrées. La propriété d'accord est très contraigante : elle stipule qu'une seule valeur peut être décidée.

Dans un contexte réparti synchrone, le consensus peut être résolu quel que soit le nombre de processus qui crashent. La situation est différente dans les systèmes asynchrones. Dans ceux-ci, il est impossible de résoudre le consensus avec un algorithme déterministe dès qu'un processus peut crasher (même un seul), que le medium de communication soit des registres de lecture/écriture [19] ou un système à passage de messages [9]. Ce résultat d'impossibilité constitue l'un des résultats les plus importants de la calculabilité répartie.

Contrecarrer cette impossibilité est possible si l'on restreint partiellement l'asynchronisme du système, ou si l'on enrichit le système asynchrone en fournissant aux processus des informations sur les fautes ou encore en utilisant des algorithmes probabilistes [5] (de tels algorithmes sont décrits dans [28] pour les systèmes répartis asynchrones dans lesquels des processus peuvent crasher ou commettre des fautes byzantines). Dans le cas où la communication se fait via des registres partagés, le consensus peut être résolu lorsque les registres sont accessibles par des opérations atomiques plus puissantes que le opérations de base que sont la lecture et écriture telles que compare&swap [6].

5. Algorithmes de type Las Vegas.

6. Ceci a donné naissance à la notion de *consensus number* introduite en 1991 par M. Herlihy [11]. Cette notion fournit une hiérarchie infinie qui permet de capturer le pouvoir de synchronisation des opérations matérielles en présence d'asynchronisme et de crash de processus. Le lecteur intéressé pourra consulter [30].

De la mémoire partagée au passage de messages Une question de base importante est la suivante : peut-on construire l'objet informatique le plus simple, à savoir un registre lire/écrire, dans un système asynchrone à passage de messages, dans lequel des processus peuvent crasher ?

La réponse est « non » si la moitié ou plus des processus peuvent crasher (et « oui » dans l'autre cas) [2]. Un tutoriel sur ce qu'il est possible de réaliser et ce qu'il n'est pas possible de réaliser dans les systèmes répartis asynchrones en présence de crash de processus est présenté dans [29].

En guise de conclusion : Un très court historique

L'algorithmique répartie trouve ses fondements dans les problèmes « d'exclusion mutuelle » [8], lorsque celle-ci doit être résolue sans pré-supposer des variables sous-jacentes atomiques (c'est-à-dire sans supposer le problème déjà résolu au niveau du matériel [14]). Alors que l'exclusion mutuelle a été initialement introduite pour résoudre les problèmes posés par l'accès aux ressources (physiques) partagées par un ensemble de processus, une étape fondamentale a ensuite été franchie avec l'introduction des registres lire/écrire non-atomiques [15, 18], du concept de « machine d'état répartie » [7] [16], et des notions de tolérance aux fautes telles que les comportements byzantins [17].

On trouvera une histoire des débuts de la programmation concurrente dans [5], et un historique récent du calcul réparti et de ses motivations dans [24]. Le lecteur intéressé trouvera dans [32] une introduction au calcul réparti fondée sur des problèmes types. En ce qui concerne les algorithmes répartis, il pourra aussi consulter les ouvrages [12, 25, 31] qui considèrent les systèmes répartis dans lesquels la communication est réalisée via une mémoire partagée et les ouvrages [3, 6, 26, 28] qui considèrent les systèmes répartis (tolérant ou non les fautes) dans lesquels la communication est réalisée par passage de messages.

Références

[1] Aigner M. and Ziegler G., *Proofs from THE BOOK (4th edition)*. Springer, 274 pages, ISBN 978-3-642-00856-6 (2010)

[2] Attiya H., Bar-Noy A., Dolev D., Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1) :121-132 (1995)

[3] Attiya H., Welch J.L., *Distributed computing : fundamentals, simulations and advanced topics, (2nd Edition)*, Wiley-Interscience, 414 pages, ISBN 0-471-45324-2 (2004)

[4] Borowsky E. and Gafni E., Immediate atomic snapshots and fast renaming. *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, pp. 41-50 (1993)

[5] Brinch Hansen P. (Editor), *The origin of concurrent programming*. Springer, 534 pages (2002)

7. Dont la blockchain n'est qu'un avatar parmi d'autres...

[6] Cachin Ch., Guerraoui R., and Rodrigues L., *Reliable and secure distributed programming*, Springer, 367 pages, ISBN 978-3-642-15259-7 (2011)

[7] Delporte-Gallet C., Fauconnier H., Rajsbaum S. and Raynal M., t-Resilient k-immediate snapshot and its relation to agreement problems. Soumis à publication (2020)

[8] Dijkstra E.W., Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9) :569 (1965)

[9] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2) :374-382 (1985)

[10] Gafni E. and Rajsbaum S., Recursion in distributed computing. *Proc. 12th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'10)*, Springer, LNCS 6366, pp. 362-376 (2010)

[11] Herlihy M.P., Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1) :124-149 (1991)

[12] Herlihy M., Shavit N., *The art of multiprocessor programming*. Morgan Kaufmann, 508 pages, ISBN 978-0-12-370591-4 (2008)

[13] Herlihy M.P., Wing J.M, Linearizability : a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3) :463-492 (1990)

[14] Lamport L., A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8) :453-455 (1974)

[15] Lamport L., Concurrent reading while writing. *Communications of the ACM*, 20(11) :806-811 (1977)

[16] Lamport L., Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7)-558-565 (1978)

[17] Lamport L., Shostack R. and Pease M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3)-382-401 (1982)

[18] Lamport L., On interprocess communication, Part I : basic formalism. *Distributed Computing*, 1(2) :77-85 (1986)

[19] Loui M. and Abu-Amara H., Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4 :163-183, JAI Press Inc. (1987)

[20] Lynch N. A., *Distributed algorithms*. Morgan Kaufmann Pub., San Francisco (CA), 872 pages, ISBN 1-55860-384-4 (1996)

[21] Peterson G.L., Myths about the mutual exclusion problem, *Information Processing Letters*, 12(3) :115-116 (1981)

[22] Rajsbaum S., Iterated shared memory models. *Proc. 9th Latin American Symposium on Theoretical Informatics (LATIN'10)*, Springer LNCS 6034, pp. 407-416 (2010)

[23] Rajsbaum S., Raynal M., An introductory tutorial to concurrency-related distributed recursion. *Bulletin of the European Association of Theoretical Computer Science (EATCS)*, 111 :57-75 (2013)

[24] Rajsbaum S., Raynal M., Mastering concurrent computing through sequential thinking : A half-century evolution. *Communications of the ACM*, Vol. 63(1) :78-87 (2020)

[25] Raynal M., *Concurrent programming : algorithms, principles and foundations*. Springer, 515 pages, ISBN 978-3-642-32026-2 (2013)

[26] Raynal M., *Distributed algorithms for message-passing systems*. Springer, 510 pages, ISBN 978-3-642-38122-5 (2013)

[27] Raynal M., Parallel computing *vs* distributed computing : a great confusion ? *Proc. 1st European Workshop on Parallel and Distributed Computing Education for Undergraduate Students (Euro-EDUPAR), Satellite workshop of EUROPAR'15*, Springer LNCS 9523, pp. 41-53, (2015)

[28] Raynal M., *Fault-tolerant message-passing distributed systems : an algorithmic approach. Springer*, 492 pages, ISBN 978-3-319-94140-0 (2018)

[29] Raynal M., The notion of universality in crash-prone asynchronous message-passing systems : a tutorial. *Proc. 38th International Symposium on Reliable Distributed Systems (SRDS'19)*, IEEE Computer Press, pp. 334-350 (2019)

[30] Raynal M., An informal visit to the wonderful land of consensus numbers and beyond. *Bulletin of the European Association of Theoretical Computer Science (EATCS)*, 129 :168-192 (2019)

[31] Taubenfeld G., *Synchronization algorithms and concurrent programming*. Pearson Education/Prentice Hall, 423 pages, ISBN 0-131-97259-6 (2006)

[32] Taubenfeld G., *Distributed computing pearls*. Morgan & Claypool, 124 pages, ISBN 9781681733487 (2018)