Penser, modéliser et maîtriser le calcul

Parallélisme asynchrone et calcul réparti

Michel RAYNAL

raynal@irisa.fr

IRISA, Université de Rennes, France

Table des matières

- Le calcul concurrent/distribué
- Des modèles de calcul distribué
- De la calculabilité distribuée

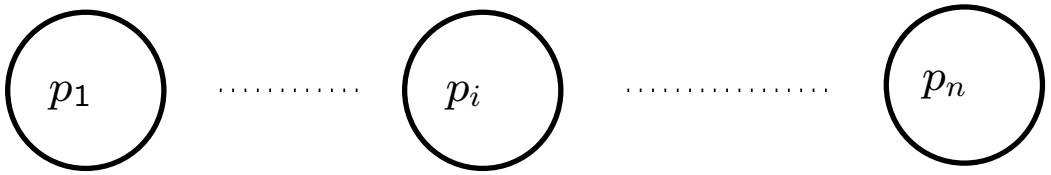
Le calcul concurrent (asynchrone)

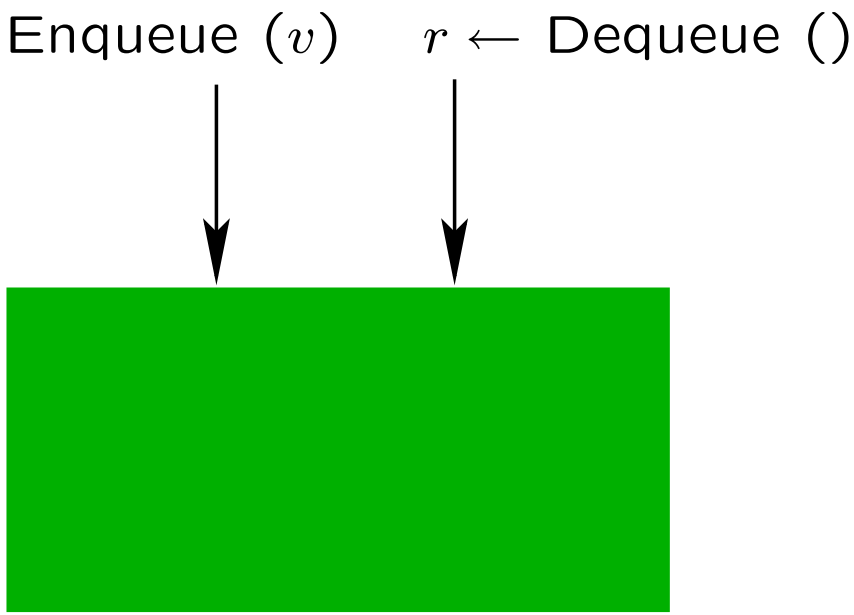
- Des processus (machines de Turing) asynchrones
- Une mémoire partagée : lecture et écriture atomiques
- Certificat de naissance :

  Co-operating Sequential Processes E.W. Dijkstra, *Programming languages*, Academic Press, pp. 43-112, 1965

Pb fondamental : construire un objet concurrent

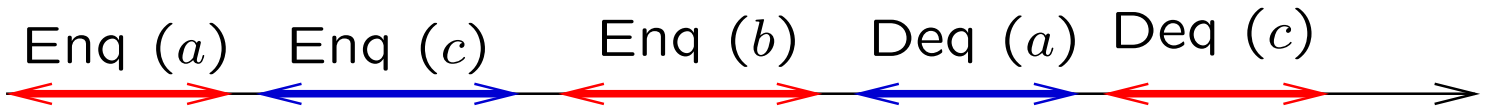
Un objet accédé par des processus concurrents



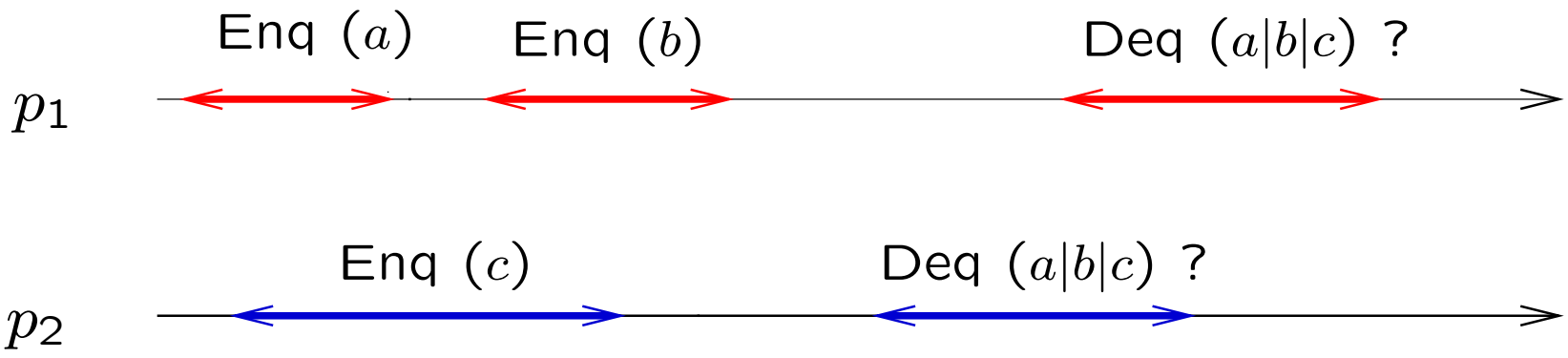


Sequentiel vs Concurrent (1)

SEQUENTIEL:



CONCURRENT:

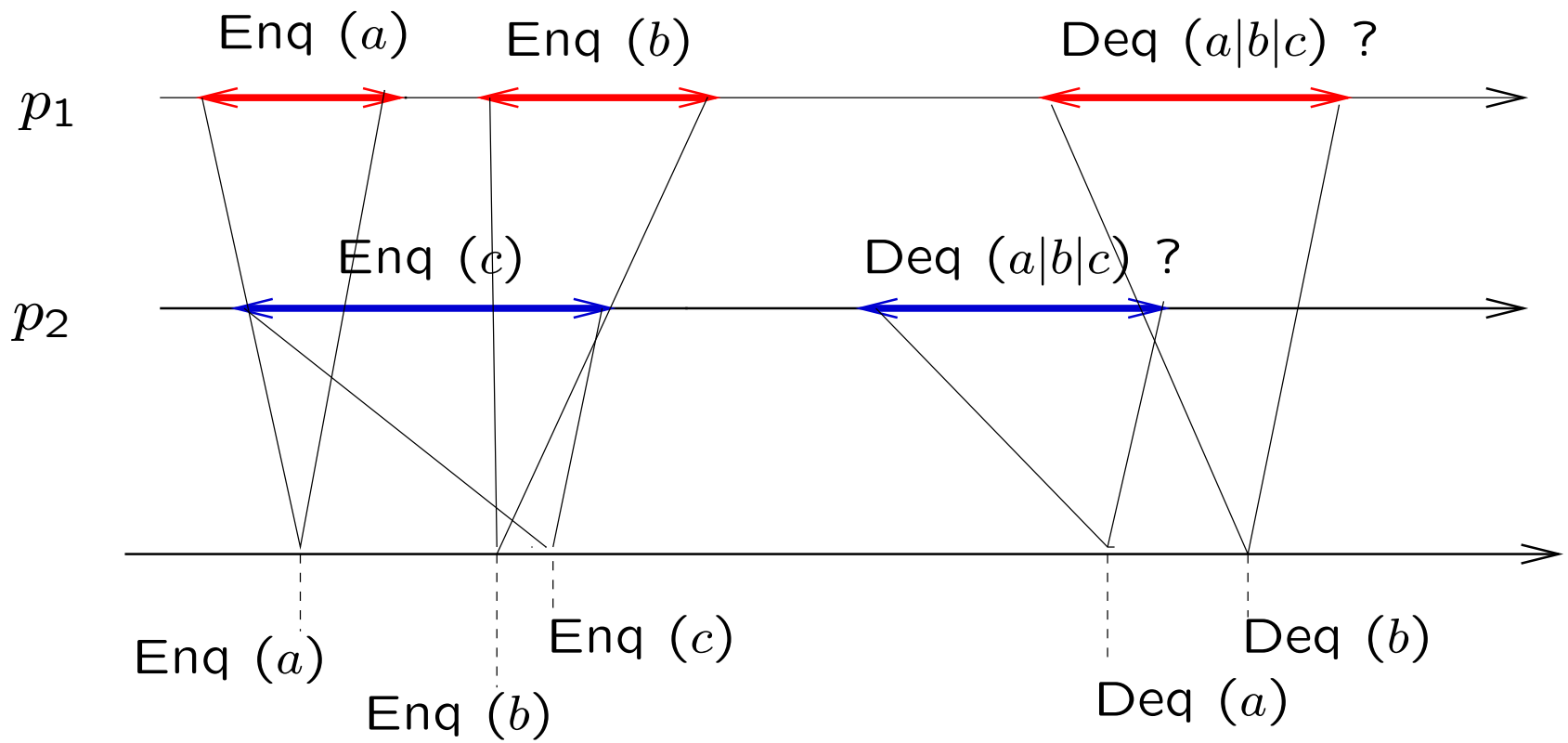


Le concept d'atomicité

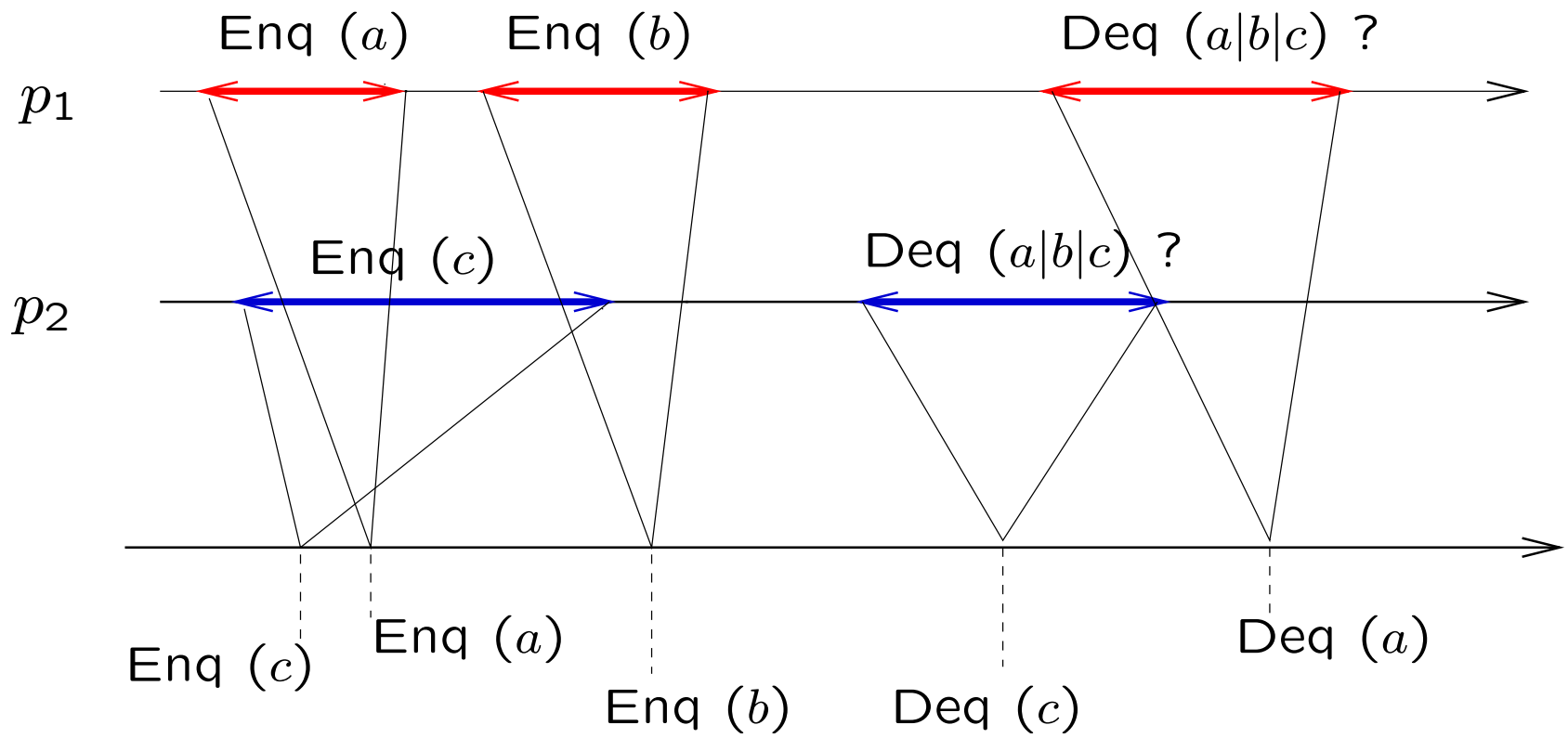
- Aussi appelé Linéarisabilité
- Propriété de sûreté
- C'est ce qui permet de raisonner séquentiellement

- Herlihy M.P. and Wing J.M., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990

Sequentiel vs Concurrent (2)



Sequentiel vs Concurrent (3)



Le calcul concurrent (classique)

- Problèmes: Exclusion mutuelle, etc.
- Propriétés de vivacité : (absence de) interblocage, famine
- Applications : allocation de resources, coordination
- Outil de base : verrou (sémaphore, moniteur, etc.)

Le calcul distribué (réparti) 1

- Le calcul réparti apparaît lorsque
  - ⋆ Résoudre

    un problème en termes d'entités (processus, agents, capteurs, pairs, acteurs, nœuds, processeurs, etc.), (aspect parallélisme)
  - ⋆ Dans un contexte où

    chaque entité : connaissance partielle des multiples paramètres mis en jeu dans le problème (aspect distribution)
  - ⋆ Certificat de naissance :

    Time, clocks and the ordering of events in a distributed system, Leslie Lamport, CACM, 21(7):558-565, 1978

Le calcul distribué (réparti) 2

- Un aphorisme célèbre (dû à Leslie Lamport)

  A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable

- Important : la notion de défaillance !

- Le calcul réparti : maîtriser l'incertitude

L'incertitude est créée par

- La multiplicité des lieux de contrôle
- Localité
- Asynchronisme
- Les fautes et défaillances
- Mobilité
- Capacité de calcul de chaque entité (e.g., batterie, bande passante, etc.)
- Dynamicité
- Etc., etc.

Modèles de calcul réparti

- Mémoire partagée vs passage de messages
- (Synchrone vs) Asynchrone
- Modèles de fautes
  - ⋆ Entités de calcul, entités de communication
  - ⋆ Fiable, Crash, Fautes byzantines

Calcul sans attente (wait-free)

- Tout processus qui ne crashe pas
  - ⋆ termine ses opérations
  - ⋆ quel que soit le comportement des autres processus
- Pas de verrou !
- Sans attente = absence de famine en dépit de crashs

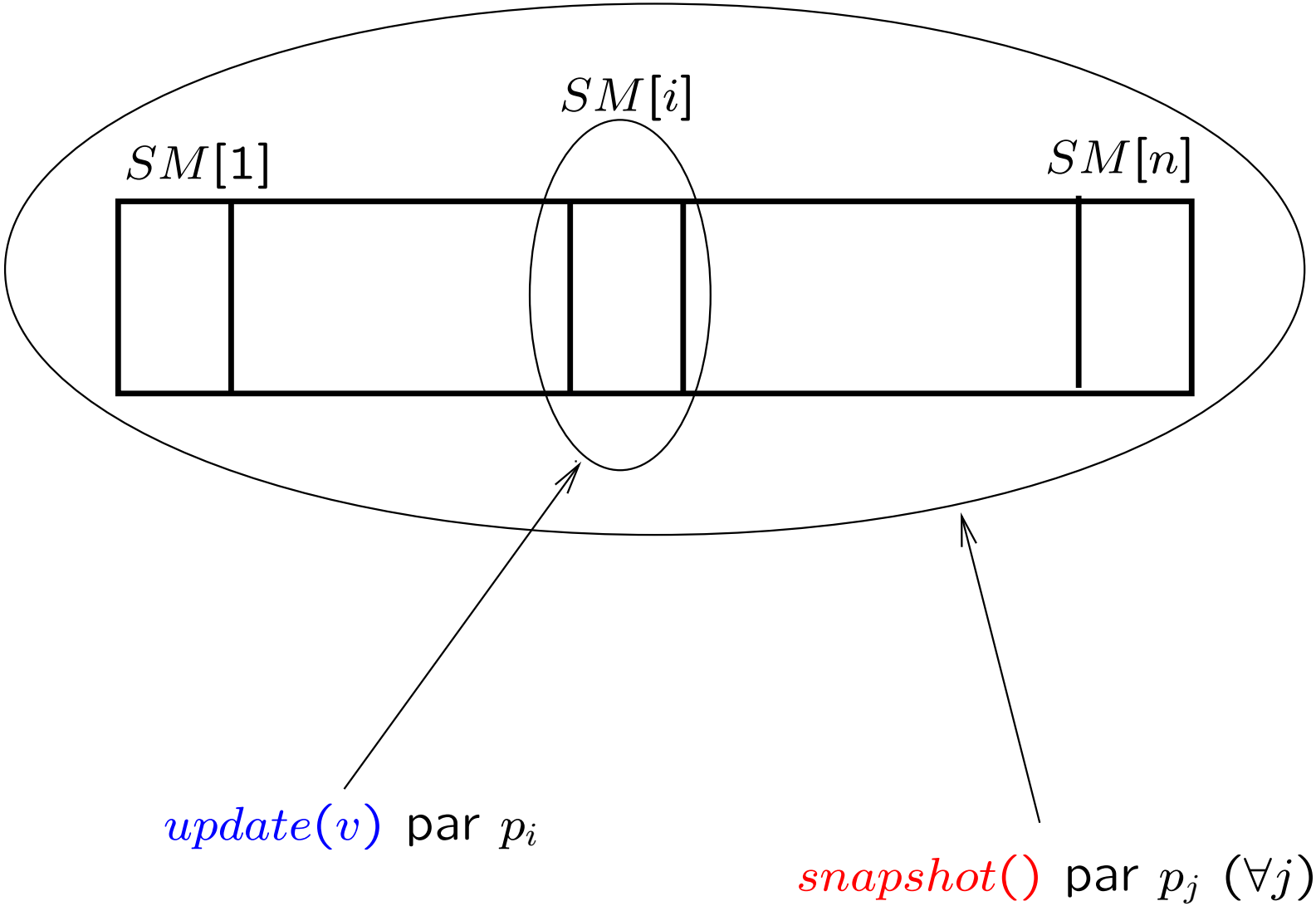
- Lamport L., Concurrent Reading and Writing. *Communications of the ACM*, 20(11):806-811, 1977

Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991

Que peut-on résoudre ainsi ?

- Modèle
  - ⋆ comportement : asynchrone
  - ⋆ communication : lecture/ecriture atomique
  - ⋆ fautes : crash de processus
- Que peut-on résoudre ?
  - ⋆ Beaucoup de problèmes (exemple : instantané)
  - ⋆ Malheureusement pas tous (“consensus number”)

Calcul d'un instantané (snapshot)



"Obstruction-free" solution

Champs : $SM[i] \ = \ (SM[i]..val, SM[i].sn)$.

operation update (v)

$sn_i \leftarrow sn_i + 1$; $SM[i] \leftarrow (v, sn_i)$.

operation snapshot()

while $true$ **do**

$A_i \leftarrow$ lecture asynchrone du tableau;

$B_i \leftarrow$ lecture asynchrone du tableau;

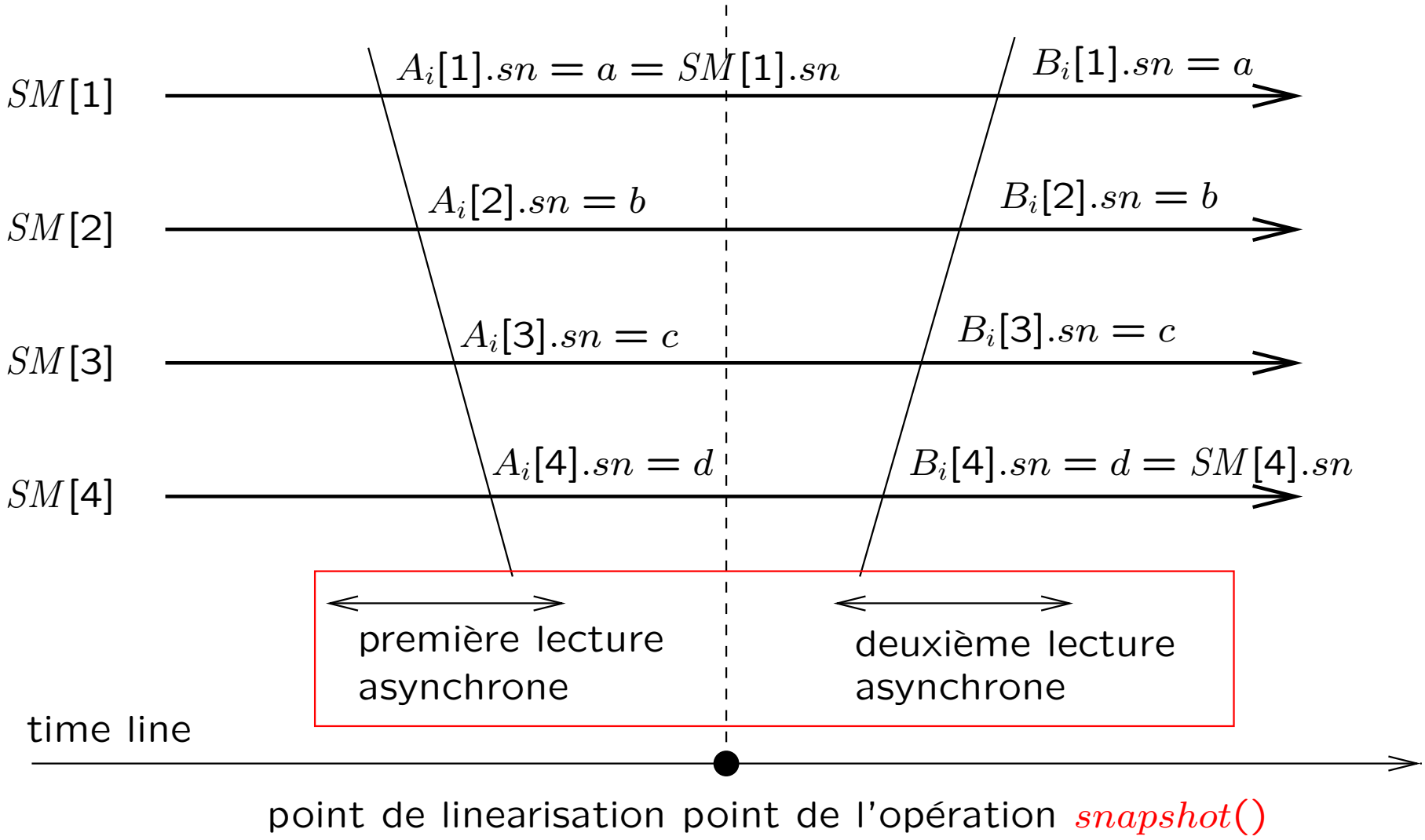
if $(\forall j : \ A_i[j].sn = B_i[j].sn)$

then return $([A_i[1].val, \ldots, A_i[n].val])$

end_if

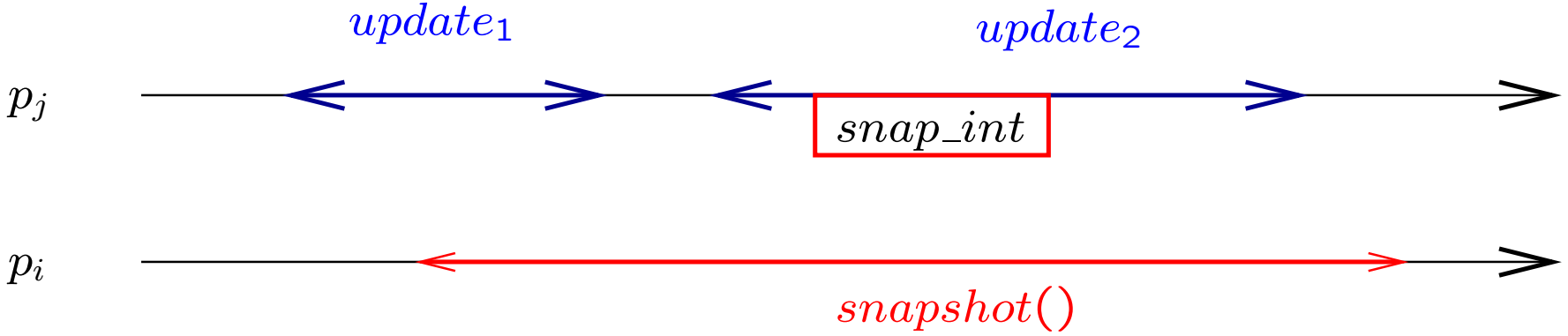
end_while.

Instantané : exemple



Solution sans attente

Chaque update() doit aider les opérations snapshot()



Algorithme (1)

Champs : $SM[i] = (SM[i]..val, SM[i].sn, SM[i].help\_array)$.

operation update (v):

$help\_array_i \leftarrow$ snapshot();

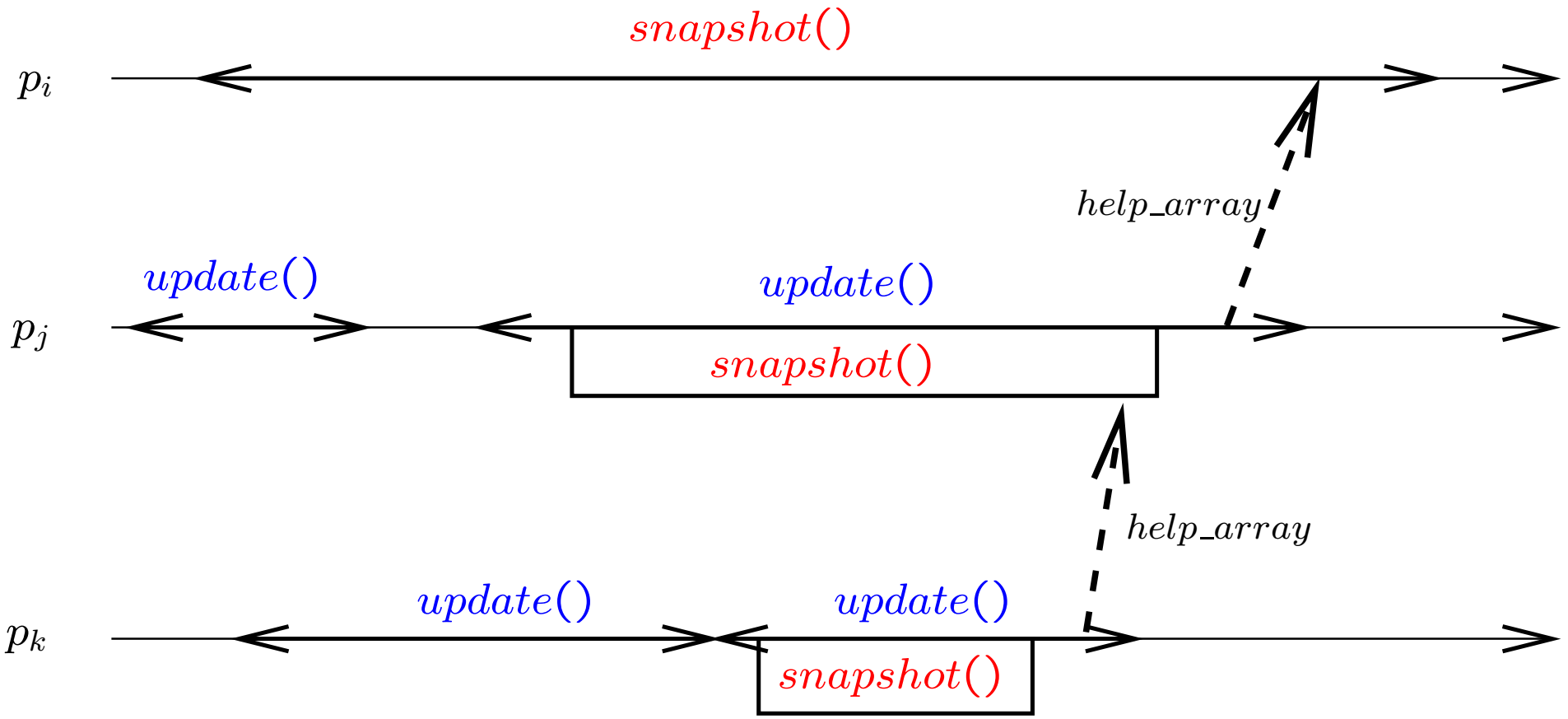
$sn_i \leftarrow sn_i + 1$;

$SM[i] \leftarrow (v, sn_i, help\_array_i)$.

Algorithme : operation snapshot()

$could\_help_i \leftarrow \emptyset$;
while $true$ **do**
  $A_i \leftarrow$ lecture asynchrone du tableau ;
  $B_i \leftarrow$ lecture asynchrone du tableau ;
  if $(\forall j : A_i[j].sn = B_i[j].sn)$
    then return $(A_i.val)$
    else for $j : 1 \leq j \leq n$ **do**
      if $(A_i[j].sn \neq B_i[j].sn)$ **then**
        if $(j \in could\_help_i)$
          then return $(B_i[j].help\_array)$
          else $could\_help_i \leftarrow could\_help_i \cup \{j\}$
      end if end if
    end for
  end if
end while

Pourquoi ça marche



deux lectures séquentielles sans ecriture

LE problème fondamental

- un objet A défini par une spécification séquentielle
- Des objets X
- Problème :

> construire une implémentation sans attente de A à l'aide de registres L/E et d'objets X

- Sans attente (wait-free) = $\forall$ nombre de crashs

Notions d'universalité

- Un objet est universel s'il permet de répondre "oui" quel que soit le nombre de processus
- Résultat fondamental (Herlihy 1991):

  L'objet consensus est un objet universel

- Notion de "consensus number" d'un objet

L'objet consensus

Chaque entité (processus, capteurs, etc.) propose une valeur et les processus corrects doivent se mettre d'accord sur la même valeur

- Propriété de sûreté :
  - ⋆ Validité : une valeur décidée est une valeur proposée
  - ⋆ Accord : Une seule valeur est décidée
- Propriété de vivacité (terminaison) : Tout processus qui ne crashe pas (stop définitif) décide

Consensus en mémoire partagée “Lire/Ecrire”

- Notion de *consensus number* d’un objet:

  Nombre maximal de processus pour lesquels on sait résoudre le consensus avec un algorithme “sans attente” (wait-freedom)

- Notion d’algorithme sans attente :

  Tout processus qui ne crashe pas doit terminer (décider)

- Sans attente = absence de famine en dépit de crash

Quelques objets de synchronisation (1)

- Test&Set $(shared)$=
  $[prev \leftarrow shared; shared \leftarrow 1;$ return $(prev)]$
- Swap $(local, shared)$= $[local \leftrightarrow shared]$

Quelques objets de synchronisation (2)

- M_to_M_Move $(shared_1, shared_2)$= $[shared_1 \leftrightarrow shared_2]$
- Compare&Swap $(shared, old, new)$=
  $[prev \leftarrow shared$;
  if $prev = old$ then $shared \leftarrow new$ fi;
  return $(prev)]$
- La paire Load_Linked, Store_Conditional

  Variantes de LL/SC dans Alpha AXP (ldl_l stl_c), IBM Power PC (lwarx/stwcx), ARM (ldrex/strex)
- Une pile, une file, etc.
- Tout object défini par une spécification séquentielle

Numéro de consensus d'un objet

Objet	NC
Registre L/E	1
Test&Set, file, pile, ...	2
...	3, 4, etc.
Compare&Swap, LL/SC, etc.	$+\infty$

Impact sur les architectures multicœurs !

Pourquoi le consensus est fondamental (1)

Le consensus permet de

résoudre le non-détermisme

créé par l'effet combiné

défaillances + asynchronisme

Modèle : asynchrone, passage de messages, crash

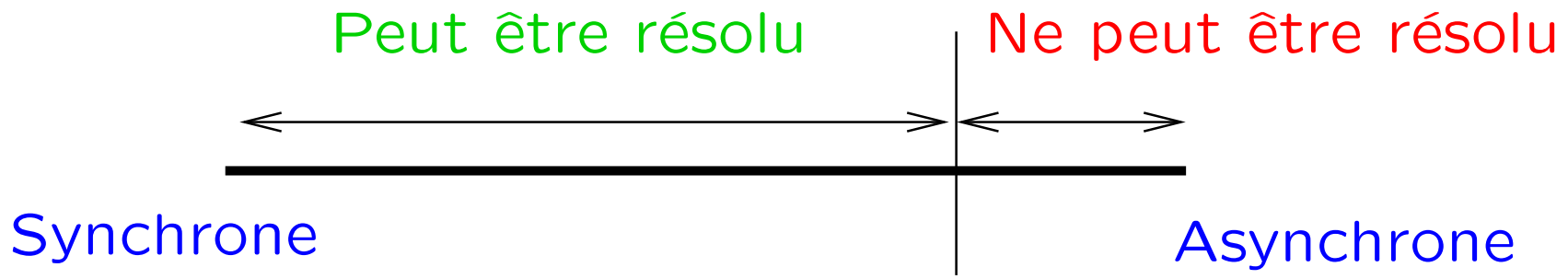
- Le consensus impossible, même si
  - ⋆ un seul processus peut être fautif
  - ⋆ faute : simple crash
  - ⋆ communications : mémoire partagée, messages
  - ⋆ consensus : binaire ou multivalué

-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

- Loui M.C., and Abu-Amara H.H., Memory Requirements for Agreement Among Unreliable Asynchronous Processes. *Par. and Distributed Computing: vol. 4 of Advances in Comp. Research,* JAI Press, 4:163-183, 1987

Systèmes à passage de messages : Que faire ?

- Et entre synchrone et asynchrone ? Frontière ?



- Le concept de détecteur de fautes (oracles)
- Notion de “plus faible oracle” pour un pb donné

L'oracle le plus faible pour le consensus

La classe Ω

- Chaque processus p_i est pourvu d'une variable locale (en lecture seule) $leader_i$
- Il existe un processus correct p_ℓ et un instant τ après lequel, pour tout processus (non crashé) on a toujours $leader_i = \ell$
- Noter : on caractérise des exécutions

Conclusion

- Algorithmes distribués = maîtriser l'incertitude créée par l'asynchronisme, les défaillances, etc.
- Durant les vingt dernières années: nombreux résultats fondamentaux (bornes inférieures, possibilité vs impossibilité), modèles précis définis, etc.
- Beaucoup reste à faire !

A propos de modèles

- Modèle de communication : mémoire vs messages
- Modèle de synchronie : du synchrone à l'asynchrone
- Modèle de fautes : des crashs aux fautes byzantines
- Mobilité, dynamicité, etc.

Quelques références

- Attiya H. and Welch J., Distributed Computing: Fundamentals, Simulations and Advanced Topics, (2d Edition), *Wiley-Interscience*, 414 pages, 2004
- Lynch N.A., Distributed Algorithms. *Morgan Kaufmann Pub.*, San Francisco (CA), 872 pages, 1996
- Raynal M., Communication and Agreement Abstractions for Fault-Tolerant Distributed Systems. *Morgan & Claypool Publishers*, to appear, 2010
- Raynal M., Une introduction à l'algorithmique distribuée des systèmes asynchrones. *Encyclopédie Vuibert* de l'informatique et des systèmes d'information, pp. 179-194, 2006 (ISBN 978-2-7117-4846-4)
- Taubenfeld G., Synchronization algorithms and and concurrent programming. *Prentice-Hall*, 423 pages, 2006