# Simulation océanique avec MPI

**DM d'algorithmique parallèle**

Jérémie Dumas

Janvier 2010

ENS de Lyon

## Table des matières

## Introduction

On considère le scénario suivant : une grille de taille $n \times n$, où chaque case est soit vide, soit occupée par une sardine, soit occupé par un requin. Le scénario envisager consiste à faire évoluer cette grille selon des règles prédéfinies, où les poissons vont se déplacer aléatoirement sur des cases adjacentes (mais pas en diagonale), se reproduire, mourir de faim, manger des sardines, etc.

Les règles précises sont détaillées dans le sujet, donc on ne s'étendra pas dessus. L'important étant que la mise à jour s'effectue case par case, en parcourant la grille ligne par ligne, de haut en bas (chaque ligne étant elle-même parcourue de droite à gauche).

On envisage deux versions de l'algorithme de simulation : l'un séquentiel, l'autre parallèle. Pour l'algorithme parallèle, on associe à chaque processeur une bande horizontale de l'océan, qu'il se doit de mettre à jour et de communiquer à ses voisins. On commencera donc par présenter rapidement l'algorithme séquentiel, avant d'en présenter et d'analyser la version parallèle.

## 1 Algorithme séquentiel

Supposons que l'on dispose d'une primitive `update(i,j)` qui met à jour la case $(i, j)$ de la grille, selon les règles définies dans le sujet. On note que cette fonction prend soin de ne pas mettre à jour une deuxième fois un poisson qui se serait déplacé vers la droite ou vers le bas au cours de l'itération en cours. On peut alors définir un algorithme de simulation séquentiel comme suit :

**Programme 1** Simulation séquentielle

**Entrée :** Les paramètres de la simulation $(r, n, p_{requin}, p_{sardine}, A_{requin}, A_{sardine}, \ldots)$
**Sortie :** L'océan après $r$ itérations de la simulation

```
generateOcean(n, n) // Distribution initiale
Pour k = 0 à r − 1 faire
  Pour i = 0 à n − 1 faire
    // Graine aléatoire pour la mise à jour de la ligne i
    initializeSeed(seed + i + k)
    Pour j = 0 à n − 1 faire
      update(i, j)
    Fin Pour
  Fin Pour
Fin Pour
Retourner Ocean
```

**Complexité** Si on suppose que la mise à jour d'une case (fonction `update(i,j)`) a un coût $\omega$, et que la génération de l'océan initial prend un temps $\mathcal{O}(n^2)$, alors l'algorithme de simulation séquentiel présenté ici a clairement un coût $\mathcal{O}(rn^2\omega)$.

# 2 Algorithme parallèle

## 2.1 Description de l'algorithme

### 2.1.1 Découpage en bande

On a convenu que chaque processeur s'occupait d'une bande horizontale de l'océan. On effectue un découpage *homogène*, sauf pour le dernier processeur qui aura peut-être moins de lignes à mettre à jour que les autres.

Concrètement, si on a $p$ processeurs et une grille de taille $n \times n$, chaque processeur va s'occuper de $\lceil \frac{n}{p} \rceil$ lignes, sauf le dernier qui n'aura que $n \mod p$ lignes si $p$ ne divise pas $n$. On notera $\gamma = \lceil \frac{n}{p} \rceil$ la largeur usuelle d'une bande horizontale (cf. figure 1).

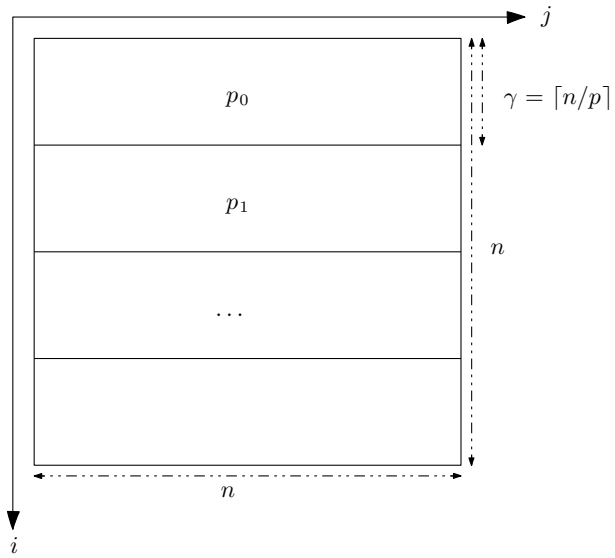


FIGURE 1: Illustration du découpage en bandes horizontales

### 2.1.2 Ordre de parcours

Pour la version parallèle de l'algorithme, il y a également plusieurs choix d'implémentations à faire. Tout d'abord, on remarque que l'on n'est pas obligé de mettre à jour strictement ligne par ligne chaque bloc horizontal. En effet, pour mettre à jour une case $(i, j)$, il suffit que les cases $(i-1, j)$ et $(i, j-1)$ aient été mises à jours. On a donc plusieurs motifs (*stencils*) possibles pour la mise à jour de la grille. Cf. les figures 2 et 3.

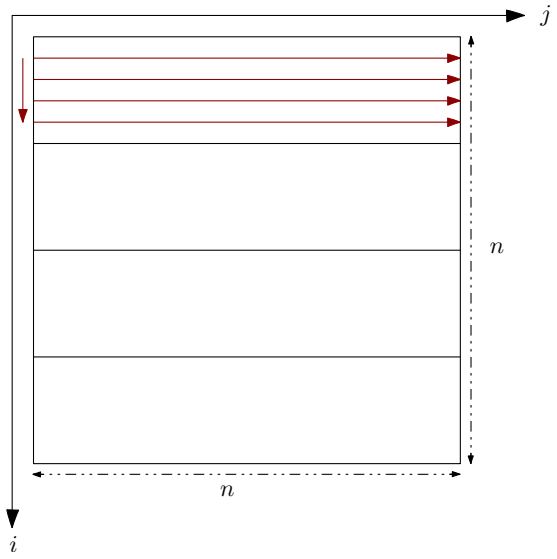


FIGURE 2: Mise à jour classique, le parcours en ligne

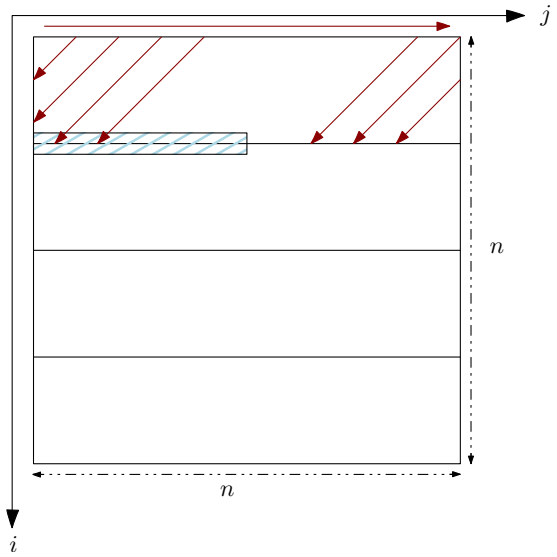


FIGURE 3: Mise à jour possible, le parcours en diagonal

Une fois qu'on a décidé de l'ordre de mise à jour, il faut se demander si chaque processeur envoie des lignes entières aux autres processeurs, ou s'il les envoie en plusieurs fois. Par exemple, un processeur $p_j$ s'occupant des lignes $i_{deb}$ à $i_{fin}$ devra faire plusieurs choses :

- Une fois que $p_j$ aura fini de mettre à jour la ligne $i_{deb} + 1$, alors les lignes $i_{deb}$ et $i_{deb} - 1$ seront correctes, et il faudra les transmettre au processeur $p_j - 1$.

– Une fois que $p_j$ aura fini de mettre à jour la ligne $i_{fin}$, alors il faut transmettre au processeur suivant $p_j + 1$ l'info concernant les lignes $i_{fin}$ et $i_{fin} + 1$ (car la ligne a peut-être changé).

On peut faire ces envois une fois que les lignes sont complètement mises à jour (avec l'ordre de parcours normal), ou lorsque le prochain bloc de $q$ colonnes a été parcouru (avec l'ordre de parcours en diagonale). L'avantage du deuxième modèle, si la latence $L$ n'est pas trop élevée, est de permettre au processeur $p_j + 1$ de commencer sa mise à jour avant que $p_j$ n'ait complètement fini la sienne. L'avantage est plus important lorsque les blocs sont larges ($\gamma$ assez grand).

Cependant, l'envoi de message, la vérification qu'une case peut être mise à jour, et d'autres détail font que ce deuxième ordre de parcours est plus compliqué à implanter. Aussi a-t-on choisi de faire un parcours "classique" ligne par ligne, et de transmettre uniquement des lignes complètes (cf. figure 4).

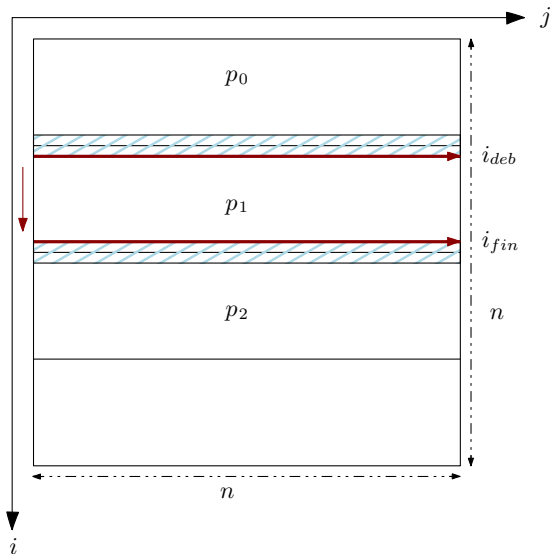


FIGURE 4: Envoi de message, $p_1$ doit partager des infos avec ses voisins.

### 2.1.3 Transmissions bloquantes et non-bloquantes

Le choix a été fait de n'utiliser que des transmissions non-bloquantes pour le calcul. Concrètement cela signifie que lors de l'envoi d'une ligne, on recopie cette ligne dans un buffer, puis on fait l'envoi non-bloquant (pas de risque de modification de la ligne par une itération ultérieure donc).

Pour ce qui est de la réception, dans le pseudo-code on présente cela comme une transmission bloquante (pour simplifier), mais cela a été implanté avec des réceptions non bloquantes. Concrètement cela signifie que les *recv* présenté ici sont implanté par des *wait* dans le code `C`, et que la réception suivante est préparée dès que possible dans nos itérations.

Ce n'est peut-être pas très clair dit comme ça, car l'implantation en `C` est plus compliquée que le pseudo-code présenté dans la section suivante, mais c'est la vie. Nous croyons en tout cas qu'il est préférable au plus possible de faire des transmissions non-bloquantes, c'est en tout cas un choix qui a été fait pour l'implantation en `C`.

## 2.2 Pseudo-code

On se donne pour faciliter l'écriture de l'algorithme une primitive `updateLine(i)`, qui s'occupe de mettre à jour entièrement la ligne $i$. Cela correspondrait aux lignes 5 à 8 du programme 1. On se donne aussi d'autres primitives :

- `receiveTopLines(src_id)` : reçoit les lignes $i_{deb}$ et $i_{deb} - 1$ du processeur $src\_id$
- `sendTopLines(dest_id)` : envoie les lignes $i_{deb}$ et $i_{deb} - 1$ au processeur $dest\_id$
- `receiveBottomLines(src_id)` : reçoit les lignes $i_{fin}$ et $i_{fin}+1$ du processeur $src\_id$
- `sendBottomLines(dest_id)` : envoie les lignes $i_{fin}$ et $i_{fin}+1$ au processeur $dest\_id$

*Remarque.* Le code en `C` se retrouve un peu complexifié par rapport à l'algorithme en pseudo-code, car il gère également le cas $\gamma < 4$, où chaque processeur s'occupe d'un petit nombre de lignes. Concrètement cela signifie que les lignes 8-9 et 15-16 se "recoupent" par rapport à l'algorithme présenté ici (d'où les deux "bloc" `first phase` et `third phase` identiques dans le code `C`).

**Programme 2** Simulation parallèle

**Entrée :** Les paramètres de la simulation $(r, n, p_{requin}, p_{sardine}, A_{requin}, A_{sardine}, \ldots)$
**Sortie :** L'océan après $r$ itérations de la simulation

1: `generateOcean`$(n, n)$ // Distribution initiale
2: $p \leftarrow$ `getNumberOfProc`()
3: $id \leftarrow$ `getMyIdentifier`()
4: $i_{deb} \leftarrow id \times \lceil \frac{n}{p} \rceil$ // Début de la bande horizontale
5: $i_{fin} \leftarrow \min(n, (id+1) \times \lceil \frac{n}{p} \rceil) - 1$ // Fin de la bande horizontale
6: **Pour** $k = 0$ à $r - 1$ **faire**
7: // Premières lignes
8: `receiveTopLine`$(id - 1)$ **si** $id > 0$ // Bloquant
9: `updateLine`$(i_{deb})$
10: `updateLine`$(i_{deb} + 1)$
11: `sendTopLine`$(id - 1)$ **si** $id > 0$ // Non-bloquant
12: // Lignes centrales
13: **Pour** $i = i_{deb} + 2$ à $i_{fin} - 2$ **faire**
14: `updateLine`$(i)$
15: **Fin Pour**
16: // Dernières lignes
17: `receiveBottomLine`$(id + 1)$ **si** $id < p - 1$ **et** $k > 0$ // Bloquant
18: `updateLine`$(i_{fin} - 1)$
19: `updateLine`$(i_{fin})$
20: `sendBottomLine`$(id + 1)$ **si** $id < p - 1$ // Non-bloquant
21: **Fin Pour**
22: `receiveBottomLine`$(id + 1)$ **si** $id < p - 1$ **et** $k > 0$ // Bloquant
23: **Retourner** *Ocean*

## 2.3 Analyse de l'algorithme

### 2.3.1 Complexité

Coût d'un appel à la fonction `updateLine` : $n\omega$. Pour l'envoi de message, on transmet à chaque fois deux lignes, ce qui fait un message de taille $2n$. Ainsi les fonctions `receiveTopLine`, `receiveBottomLine`, `sendTopLine` et `sendBottomLine` ont un coût $L + 2nb$, avec $L$ et $b$ fixés. On suppose ici que $p$ divise $n$.

Analysons un peu plus finement ; on va noter :

- $C(k, j)$ la date à laquelle $p_j$ commence l'itération $k$ (avant la l.8).
- $T(k, j)$ la date à laquelle $p_j$ a reçu les 2 premières lignes (après la l.8).
- $B(k, j)$ la date à laquelle $p_j$ a reçu les 2 dernières lignes (après la l. 17).

On peut établir les relations suivantes :

$$T(k,j) = \max \begin{cases} T(k,j-1) + 2n\omega + L + 2nb & \text{Fin de la transmission} \\ C(k,j) & \text{Déroulement de la boucle} \end{cases} \tag{1}$$

$$B(k,j) = \max \begin{cases} B(k-1,j+1) + 2n\omega + L + 2nb & \text{Fin de la transmission} \\ T(k,j) + (\frac{n}{p} - 2)n\omega & \text{Déroulement de la boucle} \end{cases} \tag{2}$$

$$C(k,j) = B(k-1,j) + 2n\omega \tag{3}$$

Avec les valeurs aux bornes suivantes :

$$\begin{aligned} &C(0,j) = 0 \\ &T(0,0) = 0 \\ &T(k,j) = -\infty \text{ si } j < 0 \\ &B(0,j) = T(0,j) + (\frac{n}{p} - 2)n\omega \\ &B(k,j) = -\infty \text{ si } j \geqslant p \end{aligned}$$

En faisant quelques substitutions on voit déjà que $C(1,j) = j(2n\omega + L + 2nb) + \frac{n}{p}n\omega$. On peut même faire mieux. Notons $U = 2n\omega + L + 2nb$ et $V = \frac{n}{p}n\omega$. On peut alors réécrire les équations (1) à (3) de la manière suivante (pour $k > 1$) :

$$\begin{aligned} C(k,j) &= \max\left(C(k-1,j+1) + U, T(k-1,j) + V\right) \\ T(k,j) &= \max\left(T(k,j-1) + U, C(k,j)\right) \end{aligned}$$

Avec cela, on démontre facilement par récurrence que $T(k,j) = C(k,j)$ pour $k \geqslant 1$, et que $C(k,j) = C(k,0) + jU$. Les équations de récurrence deviennent alors :

$$\begin{aligned} C(k,j) &= \max\left(C(k-1,j+1) + U, C(k-1,j) + V\right) \\ &= C(k-1,0) + jU + \max\left(2U, V\right) \\ &= C(1,0) + jU + (k-1)\max(2U,V) \\ &= jU + V + (k-1)\max(2U,V) \end{aligned}$$

**Conclusion** L'algorithme termine quand le dernier processeur $p-1$ a fini la $r$ ième itération. Cela correspond donc à $C(r,p-1)$, soit une complexité finale $\mathcal{O}((p-1)U + V + (k-1)\max(2U,V))$, ou encore en substituant :

$$\mathcal{O}((p-1)(2n\omega + L + 2nb) + \frac{n}{p}n\omega + (r-1)\max(4n\omega + 2(L+2nb), \frac{n}{p}n\omega))$$

### 2.3.2 Speedup

Que gagne-t-on avec un tel algorithme ? Calculons donc le speedup $S_p = \frac{T_{seq}}{T_{par}}$ obtenu :

$$S_p = \frac{prn^2\omega}{p(p-1)(2n\omega + L + 2nb) + n^2\omega + (r-1)\max(4pn\omega + 2p(L+2nb), n^2\omega)}$$

Quand $r$ devient grand, $S_p$ équivaut alors à :

$$S_p \sim \frac{n^2\omega}{\max(p(4n\omega + 2(L+2nb)), n^2\omega)}p$$

Lorsque les communications sont négligeables, on peut donc obtenir un speedup linéaire, et c'est bien. Cela dit ça reste une approximation.

**Temps mesuré :** Sur une grille de $1000 \times 1000$, avec 1000 itérations, en passant de 1 à 8 processeurs, on obtient le gain suivant :

```
$ time ./run.sh -n 1 ./simul -i 1000 -w 1000 -h 1000
real 0m53.212s
user 0m53.083s
sys  0m0.088s

$ time ./run.sh -n 8 ./simul -i 1000 -w 1000 -h 1000
real 0m13.717s
user 0m9.117s
sys  0m0.336s
```

Soit un speedup réel de 4 . . . , ce qui souligne bien que notre formule reste une approximation, ou que les communications ne sont pas forcément négligeables.

## Conclusion

Pour finir ce rapport, il convient de souligner que l'algorithme présenté, bien que simple de principe, présente quelques subtilité d'implantation, et n'est pas évident à analyser. On pourrait se demander ce qu'il adviendrait avec une mise à jour selon l'autre motif présenté, mais ça risque fort d'être difficile à analyser.

Quoiqu'il en soit, l'important est de voir qu'il y a une marge entre le speedup théorique, et le temps mesuré en pratique sur différentes configurations. Peut-être n'a-t-on pas fait assez d'itérations pour que la différence de vitesse soit conforme à la théorie ? Difficile à dire !