Chapitre 3

Un peu d'algorithmique parallèle

3.1 Introduction

Pour effectuer un calcul parallèle, il faut réussir à le découper en petites tâches et à répartir ces tâches sur les différents processeurs. Sur une machine à mémoire distribuée, il faut aussi que les *données* nécessaires aux calculs soient elles-mêmes réparties. La manière d'agencer cette répartie est parfois le problème principal du calcul parallèle.

Le cas le plus simple est celui d'un tableau de taille n réparti entre p processeurs : chaque processeur possède une portion de taille n/p du tableau. Plus précisément, le processeur de rang i possède $A\left[i\frac{n}{p} : (i+1)\frac{n}{p}\right]$. On parle alors de répartition « par bloc ».

Dans ce contexte, le motif générique de la « transformation » (*map*) est facile à paralléliser :

```
for (int i = 0; i < n; i++)
   B[i] = f(A[i], i)
```

Tous les `B[i]` peuvent être calculés indépendamment les uns des autres en parallèle : chaque processeur calcule « sa » portion du tableau B à partir de « sa » portion du tableau A. Les différents processeurs n'ont pas besoin de communiquer. Si le temps de calcul de f ne dépend pas trop de la valeur de ses arguments, alors la charge de calcul sera équitablement répartie.

3.1.1 Algorithmes de « réduction »

Le cas de la « réduction » (*reduce*) est un peu plus compliqué :

```
double sum = 0;
for (int i = 0; i < n; i++)
   sum = sum + A[i];
```

Les itérations de la boucle ne peuvent pas être exécutées en parallèle, car elles ne sont pas indépendantes (elles lisent et écrivent la même variable `Sun`). En fait, il y a des critères simples et formels (les « conditions de Bernstein ») qui permettent de décider si deux opérations $O_1; O_2$ (qui doivent être exécutées séquentiellement) peuvent en fait être exécutées en parallèle sans modifier leur résultat. Il faut :

i) Qu'aucune variable lue par O_1 ne soit écrite par O_2.

ii) Qu'aucune variable écrite par O_1 ne soit écrite par O_2.

iii) Qu'aucune variable lue par O_2 ne soit écrite par O_1.

Dans l'exemple de la réduction, les conditions ne sont pas remplies à cause de `sum`. Il faut donc légèrement modifier l'algorithme. Par exemple, on pourrait envisager la solution suivante (pour une machine à mémoire partagée) :

Algorithme S (*Somme d'un tableau*). Il s'agit de calculer la somme d'un tableau A de n éléments sur une machine où p processeurs partagent la mémoire. On suppose que n est un multiple de p pour se simplifier la vie.

S0. [Initialisation]. Allouer un tableau `Scratch` de taille p dans la mémoire partagée.

S1. [Somme locale]. Pour $0 < i < p$, le processeur $\#i$ fait : `Scratch[i]` ← `sum(A[i*n/p : (i+1)*n/p])`. Ceci est fait en parallèle par tous les processeurs, il n'y a pas de dépendance de données.

S2. [Barrière]. Attendre que tous les processeurs aient fini l'étape précédente.

S3. [Portion séquentielle]. Le processeur #0 calcule la somme du tableau `Scratch` puis l'écrit dans la variable `sum`.

S4. [Barrière]. Tous les processeurs attendent que le processeur #0 ait fini l'étape précédente.

S5. [Finalisation]. Libérer le tableau `Scratch`.

L'étape S1 nécessite un temps n/p sur chacun des p processeurs. L'étape S3 a une complexité en temps de p. Le temps total nécessaire à l'exécution de l'algorithme est donc de $n/p + p$. Ceci atteint un minimum de $2\sqrt{n}$ lorsque $p = \sqrt{n}$. Au-delà de cette limite, ajouter des processeurs *augmente* le temps d'exécution (c'est la limite classique au *strong scaling* dont il est question § 1.5.2).

Algorithme R (*Somme d'un tableau, méthode en arbre*). Mêmes hypothèses que pour l'algorithme S.

R0. [Fini ?]. Si $n = 1$, renvoyer $A[0]$.

R1. [Initialisation]. Allouer un tableau `Scratch` de taille $\lceil n/2 \rceil$ dans la mémoire partagée.

R2. [Somme]. Pour tout $0 \leq i < n/2$, faire (en parallèle) : $\texttt{Scratch}[i] \leftarrow A[2i] + A[2i+1]$. Enfin, si n est impair, faire $\texttt{Scratch}[n/2] \leftarrow A[n-1]$.

R3. [Récursion]. Invoquer l'algorithme R récursivement pour calculer la somme du tableau `Scratch` (de taille $\lceil n/2 \rceil$). Stocker le résultat dans une variable x.

R4. [Finalisation]. Libérer le tableau `Scratch` et renvoyer x.

Si on dessine le « flot » des données entre les différentes étapes de l'algorithme R, on s'aperçoit qu'il forme un arbre binaire (qui est complet si l'entrée est de taille 2^n), dans lequel l'information monte des feuilles vers la racine.

Comme il y a été fait allusion au § 1.5.2, le nombre total d'opérations élémentaires effectuées par un algorithme parallèle est son *travail*, et on le note $W(n, p)$ — parfois seulement $W(n)$ s'il ne dépend pas de p, ce qui est souhaitable —, où n désigne la taille de l'entrée et p le nombre de processeurs.

L'étape R2 nécessite $n/2$ opérations, et tout le reste est négligeable en dehors de l'appel récursif. On a donc :

$$\begin{aligned} W(n) &= \frac{n}{2} + W(n/2) && (3.1)\\ &= \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 1 && (3.2)\\ &= n \times \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{n} \right) && (3.3)\\ &\leq n && (3.4) \end{aligned}$$

Le nombre d'appel récursif est précisément de $\lceil \log_2 n \rceil$. Tant que $p \leq n/2$, alors on peut affirmer que l'étape R2 nécessite un temps $\frac{n}{2p}$ avec p processeurs. Le tout va donc prendre un temps $n/p + \log_2 p$, qui est minimal lorsque $p = n/2$ et vaut alors $1 + \log_2 n$.

De plus, dans le cas d'une machine à mémoire distribuée, des communications seront fatalement nécessaires entre les processeurs, car aucun ne possède entièrement le tableau A, donc aucun ne peut calculer `sum` tout seul. La coopération est nécessaire. La solution la plus simple à programmer « à la main » est probablement celle de l'algorithme S. Heureusement, il y a `MPI_Reduce`...

3.2 Un cas non-trivial : les somme-préfixes

Enfin, il y a des opérations encore plus complexes et plus difficiles à paralléliser. Par exemple, le tri :

```
B = sort(A);
```

Ou le calcul des « sommes-préfixes » :

```
for (int i = 1; i < n; i++)
   A[i] = A[i] + A[i - 1];
```

Il semble que chaque itération dépende du résultat de l'itération précédente, ce qui forcerait *a priori* à effectuer le calcul séquentiellement. Dans ce cas précis, ce n'est pas vrai, mais il faut *modifier l'algorithme* assez sensiblement pour pouvoir le paralléliser.

1: **function** PREFIX-SUM(A)
2: n désigne la taille de A
3: Si $n = 1$, alors **return**
4: $k \leftarrow \lceil n/2 \rceil$
5: Allouer un tableau B de taille k
6: **parallel-for** $0 \leq i < n/2$ **do**
7: $B[i] \leftarrow A[2i] + A[2i+1]$
8: Si n est impair, alors $B[k-1] \leftarrow A[n-1]$
9: PREFIX-SUM(B)
10: **parallel-for** $1 \leq i < n$ **do**
11: Si i est impair, alors $A[i] \leftarrow B[(i-1)/2]$.
12: Si i est pair, alors $A[i] \leftarrow B[i/2-1] + A[i]$.
13: **end function**

Un raisonnement similaire à l'analyse de l'algorithme R montre que $W(n) = \frac{3}{2}n + W(n/2) = 3n\left(\frac{1}{2} + \frac{1}{4} + \cdots + 1\right) \approx 3n$. La quantité totale d'opération est *trois fois plus importante* que dans le cas de l'algorithme séquentiel naïf. Par conséquent, l'accélération parallèle sera majorée par 1/3. Autrement dit, même avec 3 processeurs cet algorithme ne peut pas battre l'algorithme séquentiel naïf. Mais on peut se convaincre que tant que $p \leq n$, on va avoir un temps d'exécution parallèle de l'ordre de $3n/p + \log_2 n$.

Si on compare le flot des données dans le calcul avec le calcul de la somme, on se rend compte qu'on a cette fois affaire à un arbre binaire complet où les données montent jusqu'à la racine, puis, dans une deuxième phase, redescendent jusqu'aux feuilles.

3.3 Opérations collectives dans MPI

3.3.1 Modèles du coût de transfert d'un message

Combien de temps nécessite la transmission d'un message ? En général c'est très compliqué à prédire, car cela dépend de la topologie du réseau, de la charge des machines, du système d'exploitation, de la qualité des cables, du nombre de *switches* à traverser et de leur qualité, de la congestion du réseau, etc.

On peut trouver dans la littérature des modèles précis pour des machines particulières.

Cependant, on peut quand même essayer de dire des choses générales. Il y a de nombreux modèles mathématiques, plus ou moins sophistiqués et qui décrivent plus ou moins fidèlement ce qui peut se passer.

Le plus simple consiste à considérer que le temps de transfert d'un message de S octets est $T(S) = \alpha + \beta \cdot S$. Le terme α représente la latence du réseau, des OS, de la bibliothèque MPI, etc. Le terme β lui représente le temps mis pour transférer un octet (c'est l'inverse de la bande passante). Ceci n'est vrai qu'approximativement, et surtout pour les grands messages.

3.3.2 Algorithmes pour SCATTER, GATHER

Tout d'abord, notons que *Scatter* et *Gather* sont des opérations réciproques. Par conséquent, un algorithme pour l'un devient un algorithme pour l'autre si on y inverse l'ordre et le sens des communications.

Tout d'abord, on peut dire des choses générales. Si p désigne la taille du communicateur et n la taille du tableau à disperser (resp. rassembler), alors on sait qu'un volume de données $(p-1)n/p$ doit sortir (resp. entrer) du processus racine à un moment ou à un autre ($p-1$ portions de taille n/p). Par conséquent, l'opération va prendre un temps supérieur à $(p-1) \cdot n/p\beta$. D'autre part, on sait aussi que ceci nécessite l'envoi d'un certain nombre de messages successivement, et donc prendre un temps supérieur à $\alpha \log_2 p$. En effet, si on imagine le SCATTER comme la diffusion d'une épidémie, alors au début il y a un seul malade (la « racine »). S'il a le temps d'envoyer un seul message, il peut en contaminer UN autre. Mais si ces deux malades ont de nouveau le temps d'envoyer un nouveau message, ils peuvent en contaminer DEUX autres, etc. De la sorte, le nombre de malades double à chaque étape, mais ça ne peut pas aller plus vite que ça. Il faut donc au moins $\log_2 p$ étapes qui prennent chacune un temps minoré par α.

Par conséquent, on trouve :

$$T_{\text{scatter}} \geq \alpha \log_2 p + (p-1)\frac{n}{p}\beta$$

Méthode « *flat-tree* » Pour faire SCATTER avec un message de taille n sur p processus, la solution la plus simple consiste en ce que la racine envoie sa « tranche » à chacun des autres processus de manière séquentielle.

Combien de temps ceci va-t-il prendre ? Le plus gros problème se situe-t-il dans le terme de latence ou dans celui de volume (ou les deux ?)

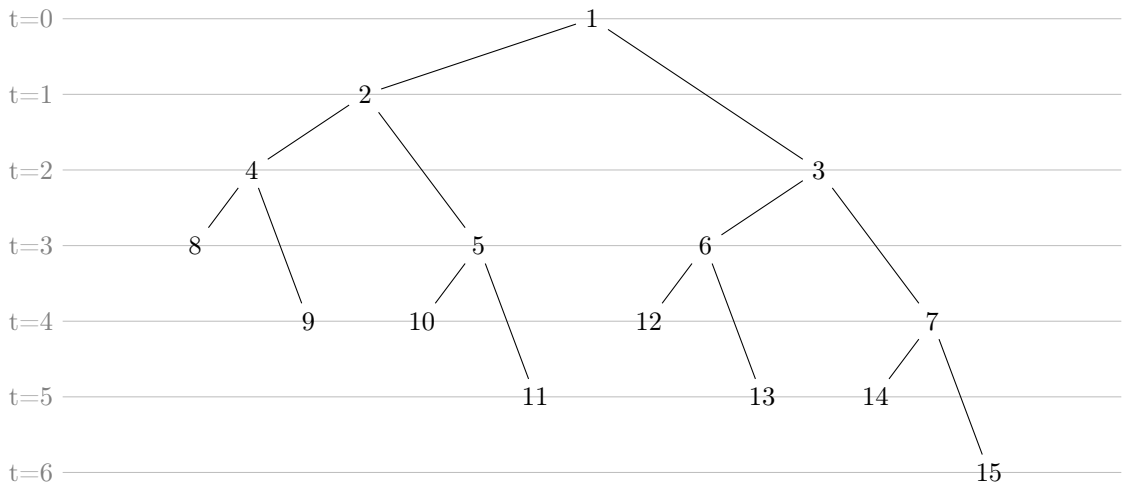
Les messages sont de taille n/p. D'après le modèle, le temps nécessaire à l'envoi d'un de ces messages est $\alpha + n/p \cdot \beta$. Le processus racine garde sa part, et il envoie $p-1$ messages aux autres avec leurs parts. Le tout prend donc un temps :

$$(p-1)\left(\alpha + \frac{n}{p}\beta\right) = (p-1)\alpha + (p-1)\frac{n}{p}\beta$$

Le terme de volume est le même que dans la borne inférieure : il est donc optimal. Par contre le temps de latence est trop grand. Il est linéaire alors que la borne inférieure est logarithmique.

Pour gagner du temps, on peut exploiter du parallélisme. Cette stratégie d'organisation des communications est assez générale et s'applique à de nombreuses situations.

Méthode de l'arbre binaire Supposons que les processus soient numérotés à partir de 1 (en pratique il suffit d'ajouter un à leur rang). L'idée consiste à agencer les processus sur un arbre binaire complet. La racine est le processus 1. Le fils gauche du processus i porte le numéro $2i$ et son fils droit porte le numéro $2i+1$.



L'idée c'est que 1 transmet à 2 un message qui contient non seulement la « portion » de 2, mais aussi de tous ses descendants. Il s'agit donc d'un message dont la taille est environ la moitié de n. Ensuite, 1 envoie à 3 un message qui contient sa « portion » ainsi que celle de ses descendants. Dès réception, 2 et 3 peuvent faire la même chose, en envoyant des messages de taille $\approx n/4$ à leurs descendants.

A priori, le premier et le dernier processus à recevoir leur portion sans en ré-emettre sont ceux qui se trouvent le plus à gauche et le plus à droite, respectivement. Il s'agit de ceux qui portent les numéros 2^h et $2^{h+1}-1$, où h représente la hauteur de l'arbre, c'est-à-dire le nombre maximal d'arêtes à traverser pour atteindre une feuille depuis la racine. On rappelle que $h = \lfloor \log_2 p \rfloor$.

Il faut que tous les noeuds de la forme 2^i aient reçu un message de leur père (de rang 2^{i-1}), avant qu'ils puisse en envoyer un à leur fils gauche (de rang 2^{i+1}). Le nombre de messages qui doivent transiter avant que le noeud 2^h ait reçu sa part est donc h.

Les noeuds de profondeur $i \geq 1$ (c.a.d. qui sont reliés à la racine par un chemin qui traverse i arêtes) reçoivent des messages qui contiennent :

$$N_i = \frac{p+1}{2^i} - 1$$

« portions » du tableau de départ (on rappelle qu'une portion est de taille n/p). Ceci pourrait se justifier par recurrence sur i (c'est un bon exercice qui est laissé au lecteur !). Cependant, on va se contenter de dire que les processus de profondeur i reçoivent moins de $(p-1)/2^i$ portions. En effet, la racine cherche à répartir $p-1$ portions sur les autres processus, et chaque processus transmet à ses enfants moins de la moitié de ce qu'il reçoit, vu qu'il garde sa propre portion.

Le volume total de données qui circule sur le réseau avant que 2^h ait reçu sa part est donc :

$$\begin{aligned} V &= \frac{n}{p} \sum_{i=1}^{h} N_i \\ &\leq \frac{n}{p} \sum_{i=1}^{h} \frac{p-1}{2^i} \\ &\leq \frac{n}{p} \sum_{i=1}^{h} \frac{1}{2^i} \\ &\leq (p-1)\frac{n}{p} \end{aligned}$$

Pour le noeud $2^{h+1}-1$, la différence est que tous les noeuds qui se trouvent sur le chemin qui le relie à la racine vont d'abord envoyer un message à leur fils gauche, avant d'en envoyer un à leur fils droit. Le nombre de messages envoyés avant que $2^{h+1}-1$ reçoive sa part est donc le double ($2h$), et le volume de données qui doit circuler sur le réseau est aussi le double.

Voyons maintenant le temps que cette méthode requiert pour réaliser Scatter. Le volume total qui doit transiter sur le réseau avant que le dernier processus ait reçu sa part est donc de l'ordre de $2n$, tandis que le nombre de messages qui doit circuler est de l'ordre de $2\log p$. On va donc avoir un temps d'exécution qui ressemble à :

$$T \leq 2\lfloor \log_2 p \rfloor \cdot \alpha + 2n \cdot \frac{p-1}{p} \cdot \beta.$$

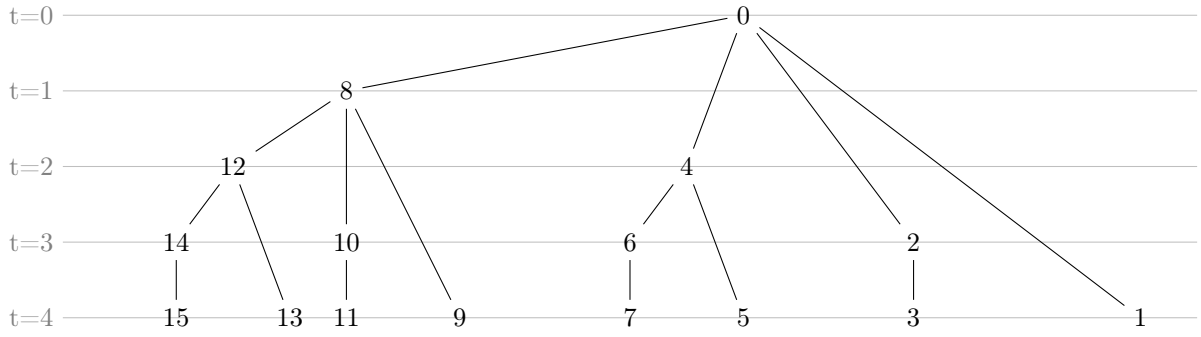
Le terme de latence est logarithmique en p, ce qui est un gros progrès par rapport à la méthode précédente. Par contre, le terme de volume est *deux fois plus gros* qu'avant. Si la taille des messages est très grande, cette méthode risque de mettre deux fois plus longtemps que la précédente. En fait, on obtient exactement deux fois la borne inférieure.

Un des avantages de cette méthode c'est qu'elle est très facile à programmer même si le nombre de processus n'est pas de la forme 2^k-1 : il suffit d'ignorer tous les envois à destinations de processus inexistants, qui sont tous au dernier niveau de l'arbre, sur la droite.

Méthode de l'arbre binomial On peut améliorer un peu la méthode précédente, en remarquant que le processus racine, après avoir glorieusement envoyé deux messages, se tourne les pouces le reste du temps.

On peut donc lui confier davantage de travail. On suppose qu'il y a 2^h noeuds numérotés de 0 à 2^h-1. Au départ la racine doit disperser un tableau qu'elle possède sur les processus de l'intervalle $[0;2^h[$. La procédure est la suivante :

- Pour disperser un tableau sur l'intervalle $[a;b[$, on repère le processus du milieu $c = (a+b)/2$.
- On envoie un message contenant les portions de l'intervalle $[c;b]$ à c.
- On s'occupe ensuite de disperser l'intervalle $[a;c[$, tandis que c s'occupe de disperser l'intervalle $[c;b[$.



Cet arbre, appelé « arbre binomial » possède 2^h noeuds et il est de hauteur h (il réapparaît dans une structure de donnée amusante, les « tas binomiaux »).

Tous les noeuds qui sont à distance d de la racine reçoivent leurs portions en même temps. Pour voir que c'est vrai, le plus simple est d'étudier la structure des arbres binomiaux. Ceci nous dépasse un peu, mais on va admettre qu'un arbre binomial de hauteur h, qu'on notera B_h est formé d'une racine, sur laquelle sont branchés h arbres binomiaux de hauteurs $0, 1, 2, \ldots, h-1$. Un arbre binomial de hauteur zéro est formé d'un seul noeud. Dans le dessin ci-dessus, « 0 » est la racine d'un B_4, tandis que « 8 » est celle d'un B_3 ; « 12 » et « 4 » sont celles d'un B_2, etc.

Ceci étant dit, on voit que les noeuds qui sont les racines de B_i reçoivent tous leurs messages en même temps (ceci pourrait se justifier plus formellement par récurrence : faites-le !). Un B_i comporte 2^i noeuds, donc les messages reçus par les racines de B_i contiennent 2^i portions de taille p/n. Prenons le B_i le plus à gauche, et remontons le chemin qui le relie à la racine : il reçoit son message *après* que le B_{i+1} qui est son père ait lui-même reçu le message du B_{i+2} qui est son propre père, etc.

Par conséquent, les B_i reçoivent leur message après que $h-i$ messages aient été envoyés, et ces messages sont de tailles respectives $2^i, 2^{i+1}, \ldots, 2^{h-1}$. Envoyer ces messages prend un temps :

$$T = \alpha(h-i) + \beta\frac{n}{p}\sum_{k=i}^{h-1} 2^i$$

On en déduit le temps d'exécution de la méthode pour faire un SCATTER complet. Il suffit pour cela de regarder quand est-ce que les racines des B_0, qui n'ont donc pas d'enfants, ont reçu leur message. D'après ce qu'on a dit précédemment, c'est au bout d'un temps :

$$\begin{aligned} T &= \alpha h + \sum_{k=0}^{h-1} 2^i \\ &= \alpha h + \left(2^h - 1\right)\frac{n}{p}\beta \\ &= \alpha h + (p-1)\frac{n}{p}\beta. \end{aligned}$$

Et ceci est en fait *exactement* la borne inférieure qu'on avait trouvée ci-dessus.

3.3.3 Renversement des communications

De la même manière que *scatter* et *gather* peuvent être réalisés par les mêmes algorithmes « mis à l'envers », c'est également le cas de *broadcast* et *reduce*. En effet, dans le premier, des tableaux de taille n partent de la racine et sont envoyés à tous les processus, tandis que dans le deuxième des tableaux de taille n partent de tous les processus pour être concentrés à la racine. Un algorithme de *broadcast* est donc facile à transformer en algorithme de *reduce* et vice-versa.

Il en va de même pour *all-gather* et *reduce-scatter*. Dans le premier, chaque processus récupère une tranche de taille n/p depuis chacun des autres processus. Dans le deuxième, on récupère la *somme* des tranches de taille n/p de chaque autre processus. Dans les deux cas semblent donc voisins et pourraient être implémentés de manière comparable.

3.3.4 Algorithmes pour `MPI_Allgather`

On va voir deux méthodes différentes pour réaliser l'opération « All-Gather », et on va tâcher de comparer, en théorie, leur performance.

Algorithme de l'anneau L'idée est relativement simple. S'il y a p processus dans le communicateur, l'algorithme s'exécute en p phases. Dans la première phase, le processus i envoie « ses » données au processus $i+1$ (tout le monde fait ça en même temps). Du coup, le processus i reçoit les données du processus $i-1$. Bien sûr, tout ceci se fait avec les rangs *modulo* p.

Dans les phases suivantes, le processus i transmet au processus $i+1$ les données qu'il a reçues dans la phase précédente. De la sorte, au bout de $p-1$ phases, tout le monde a reçu les données de tout le monde.

Notons n la quantité totale de données qui doit être rassemblée sur chacun des processus. La « contribution » de chaque processus est donc de taille n/p. Dans chaque étape, chaque processus envoie et reçoit une quantité de donnée n/p. Si on néglige la contention du réseau, tous ces échanges peuvent se faire simultanément.

On s'attend donc à ce que cela prenne un temps :

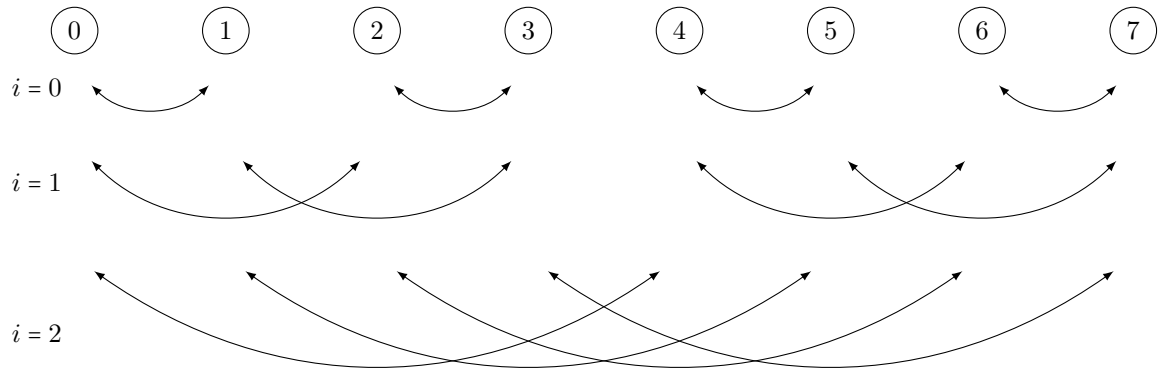
$$T_{ring} = (p-1)\alpha + (p-1)\frac{n}{p}\beta$$

Dans cette expression, le terme de bande passante (celui avec β) ne peut pas être amélioré : chaque processus doit recevoir n/p données depuis $p-1$ autres processus. Par contre, le terme de latence (celui avec α) peut être amélioré.

Algorithme du doublage récursif On peut en effet mettre sur pied un algorithme très classique avec $\mathcal{O}(\log p)$ phases au lieu de $\mathcal{O}(p)$ phases. On va supposer que le nombre de processus est une puissance de deux, donc que $p = 2^k$. Quand ce n'est pas le cas, la technique fonctionne aussi mais les détails sont sordides.

1: $r \leftarrow$ `MPI_Comm_Rank()`
2: **for** $0 \leq i < k$ **do**
3: $t \leftarrow r$ `XOR` 2^i
4: Envoyer au processus de rang t toutes les contributions connues
5: Ajouter ce qu'on reçoit de t à la liste des contributions connues
6: **end for**

Voici une illustration de l'algorithme avec $p = 8$:



Chaque processus envoie et reçoit $k = \log_2 p$ messages, dont la taille double à chaque phase. Chaque processus envoie et reçoit la même quantité de données que dans le cas précédent. Le temps que tout ceci prend est donc :

$$T_{2rec} = \alpha \log_2 p + (p-1)\frac{n}{p}\beta$$

Comparaison En théorie, il semble donc que la méthode du doublage récursif soit incontestablement meilleure (puisque le terme de latence est toujours plus faible). La réalité est malheureusement un peu plus compliquée. Quand les volumes de données sont *petits*, et que la latence domine les temps de communication, l'avantage de la méthode de doublage récursif devrait être plus prononcé — et des études empiriques confirment ce fait.

Par contre, quand les volumes de données sont importants, la latence n'a plus beaucoup d'importance car c'est alors la bande passante qui est le facteur déterminant. Les deux algorithmes ont pourtant la même bande passante.... Mais voilà : dans l'algorithme en anneau, on ne communique qu'avec ses *voisins*, tandis que dans l'algorithme de doublage récursif on communique avec des noeuds lointains. Selon la configuration du réseau (liens directs avec les voisins...) on peut obtenir moins de congestion, et une meilleure bande passante avec l'algorithme en anneau. Sur certaines machines, l'algorithme en anneau va deux fois plus vite dès que les messages font quelques Mo.

3.3.5 Algorithmes pour `MPI_Bcast`

Le *broadcast* semble être l'opération la plus simple, et pourtant... Le même raisonnement que pour Scatter permet de penser qu'un temps $\alpha \log_2 p + n\beta$ est nécessaire quoi qu'il arrive.

« Flat-tree » La technique la plus simple pour faire le *broadcast* consiste à envoyer les données aux $p-1$ autres processus successivement. Ceci prend un temps :

$$T_{flat} = (p-1)(\alpha + n\beta)$$

« Binomial-Tree » La technique en arbre est simple (surtout si $p = 2^k$). Elle s'accomplit en $\log_2 p$ phases. Chaque processus reçoit un message, et en expédie deux, de sorte à tracer un arbre binomial. Lors de la première étape, `root` contacte `root` $+ p/2$. Les deux processus se considèrent alors comme les deux racines de deux nouveaux arbres, et contactent chacun le processus qui est « au milieu » de leurs sous-arbres respectifs, et ainsi de suite.

Il y a $\log_2 p$ étapes, et donc le tout prend un temps :

$$\lceil \log_2 p \rceil (\alpha + n\beta)$$

On a donc un algorithme dont le terme de latence est optimal, mais le terme de bande passante est $\mathcal{O}(\log p)$ fois trop grand ! Cet algorithme est donc adapté aux *petits* messages.

Algorithme de Van de Geijn L'idée consiste à accomplir le *broadcast* en effectuant d'abord un *scatter* suivi d'un *all-gather*. En effectuant le *scatter* avec l'algorithme optimal (en arbre binomial), on a un temps :

$$T_{scatter} = \alpha \cdot \log_2 p + (p-1)\frac{n}{p}\beta$$

Comme on s'intéresse en priorité au cas des grands messages, on peut supposer qu'on utilise l'algorithme en anneau pour le *all-gather*. Cela donne donc un temps total pour le *broadcast* de :

$$T = \alpha \cdot (p - 1 + \log_2 p) + 2(p-1)\frac{n}{p}\beta$$

Le terme de bande passante est meilleur que dans l'algorithme de l'arbre binomial... Et il n'est plus que 2 fois plus grand que la borne inférieure.

3.4 Produit Matrice-Vecteur (dense)

On s'intéresse à un problème qui consiste à calculer le produit d'une matrice dense de taille $n \times n$ par un vecteur de taille n. Plus précisément, on veut calculer $y = M \times x$.

Il est bien connu que l'algorithme séquentiel effectue ce calcul en environ n^2 opérations.

On va supposer qu'avant le démarrage des algorithmes qu'on va considérer, la matrice et le vecteur ont été préalablement répartis sur les p processus dont on dispose. Pour le vecteur, cela signifie que le processus de rang i possède la i-ème « tranche » de taille n/p, c'est-à-dire qu'il possède les coefficients de x dans l'intervalle :

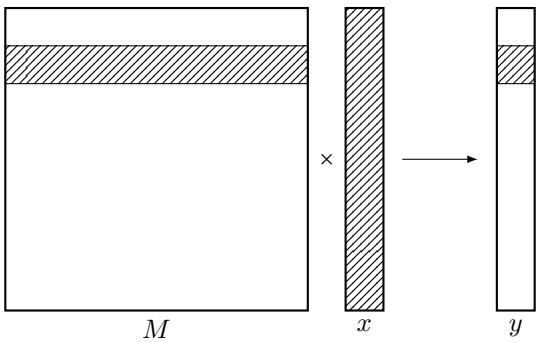
$$\left[i \cdot \frac{n}{p}; (i+1) \cdot \frac{n}{p} \right[.$$

On veut qu'à la fin du calcul, y soit réparti de la même manière. On notera donc x_i et y_i les « portions » qui appartiennent au i-ème processus.

La répartition de la matrice, elle, constitue tout l'enjeu de la discussion. On va envisager trois stratégies.

3.4.1 Stratégie n°1 : P_i possède la i-ème tranche des lignes de M

Si le processus de rang i (qu'on va noter P_i) possède la i-ème tranche des lignes de M, alors il peut calculer la i-ème tranche de y, à condition de connaître l'ensemble de x. Il n'y aura donc rien à faire pour s'assurer que la distribution de y sera correcte. Par contre, il faut « faire venir » l'ensemble de x sur chaque processeur. Il s'agit d'un « All-Gather ». Cela permet de faire en sorte que tout le monde récupère toutes les « tranches » de x, et possède donc l'intégralité du vecteur.



On peut utiliser au maximum n processeurs, car chaque processeur doit posséder au moins une ligne de la matrice. En terme de calculs, chaque processus doit multiplier une matrice rectangulaire de taille $n/p \times n$ par un vecteur de taille n. Ceci nécessite n^2/p opérations.

Pour ce qui est des communications, c'est dans le « all-gather » que tout se passe, et on a déjà vu la complexité de cette opération. Le temps total d'exécution de cette stratégie est donc :

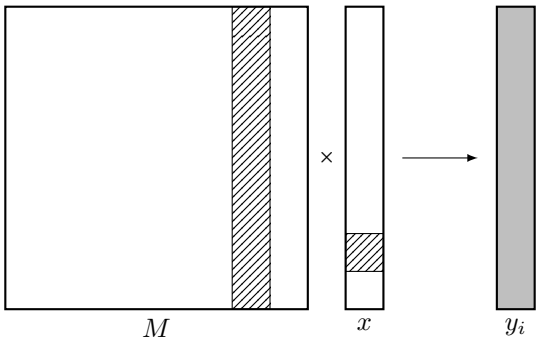
$$T = \underbrace{\frac{n^2}{p}}_{\text{calcul}} + \underbrace{\alpha \log_2 p + (p-1)\frac{n}{p}\beta}_{\text{communication}}$$

On note qu'avec n processeurs, on trouve $T = \mathcal{O}(n)$. Le temps de calcul domine asymptotiquement les communications lorsque n augmente, donc *a minima* le *weak-scaling* devrait être assez bon.

3.4.2 Stratégie n°2 : P_i possède la i-ème tranche des colonnes de M

Essayons de prendre le problème autrement, et d'éviter cette coûteuse opération collective au début.

Si le processus de rang i (qu'on va noter P_i) possède la i-ème tranche des *colonnes* de M, alors il n'a besoin que de la i-ème tranche de x, qu'il possède déjà par hypothèse sur la pré-distribution des données. « Jusque-là, tout va bien ».



Le problème, c'est que le vecteur y est la *somme* des produits $y_i = M_i \times x_i$:

$$y = \sum_{i=0}^{p-1} M_i \times x_i.$$

Il y a donc malgré tout bien une opération de re-distribution des données, mais en plus une opération de réduction ! Il faut calculer la somme des y_i que possèdent chaque processus, puis répartir le tout par « tranches » entre tout le monde. C'est précisément de ce que fait « Reduce-Scatter ».

Comme la complexité de « Reduce-Scatter » est sensiblement la même que celle de « all-gather », et que la quantité de calcul est la même que précédemment, on trouve une complexité comparable à celle de la première stratégie. On n'a donc rien gagné à ce stade.

3.4.3 Stratégie n°3 : chaque processus possède un bloc carré de M

Supposons que le nombre de processus est un carré, et donc que $\sqrt{p}$ est un entier. On découpe la matrice M en p blocs carrés de taille $n/\sqrt{p}$ (distribution 2D de données 2D).

Chacun des p blocs est identifié par deux coordonnées qui donnent sa position au sein de M. Chacun des p processus est identifié soit par son *rang* (compris entre 0 et $p-1$), soit par ses *coordonnées cartésiennes* : le processus (i,j) est celui qui possède $M_{i,j}$.

$M_{0,0}$	$M_{0,1}$	$\cdots$	
$M_{1,0}$	$M_{1,1}$	$\cdots$	
$\vdots$	$\vdots$	$\ddots$	

M

0	$\sqrt{p}$	$2\sqrt{p}$	
1	$\sqrt{p}+1$	$\cdots$	
$\vdots$	$\vdots$	$\ddots$	
$\sqrt{p}-1$			$p-1$

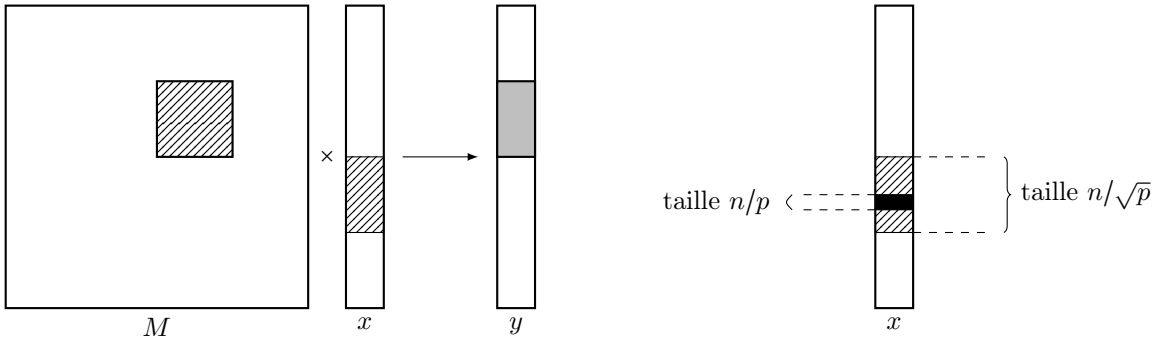
M

La conversion entre ces deux identifiants est facile :

$$r \longrightarrow \left(r \bmod \sqrt{p}, \left\lfloor \frac{r}{\sqrt{p}} \right\rfloor \right)$$
$$(i,j) \longrightarrow i + j \cdot \sqrt{p}$$

Le vecteur x est réparti sur tous les processus comme précédemment : le processus de rang r possède la r-ème tranche de x. Cela signifie que les processus de la première colonne de M possèdent collectivement les $\sqrt{p}$ première tranches de x.

Examinons maintenant le calcul que le processus (i,j) doit accomplir. Il possède $M_{i,j}$ ainsi qu'une portion de taille n/p de x (en noir sur le dessin de droite ci-dessous). Mais le problème, c'est que pour calculer un produit avec $M_{i,j}$, il lui faudrait une tranche de x plus grosse, de taille $n/\sqrt{p}$ (hachurée sur le dessin) :



Imaginons donc que x et y sont découpés à la fois en p « petites tranches » de taille n/p, qu'on note toujours x_i et y_i, mais aussi en $\sqrt{p}$ « grosses tranches » de taille $n/\sqrt{p}$, qu'on notera $\widetilde{X}_i$ et $\widetilde{Y}_i$.

Chaque « grosse tranche « , de taille $n/\sqrt{p}$ est constituée de $\sqrt{p}$ « petites tranches » de taille n/p. La première grosse tranche $\widetilde{X}_0$ est composée de $x_0, x_1, \ldots, x_{\sqrt{p}-1}$. De manière analogue, $\widetilde{X}_i$ est composé des petites tranches d'indices $[i\cdot\sqrt{p}; (i+1)\cdot\sqrt{p}[$. Vu la numérotation des processus, ces petites tranches se trouvent réparties sur les processus de la i-ème colonne.

On voit donc que le processus (i,j) a besoin de $\widetilde{X}_i$ et produit une contribution à $\widetilde{Y}_i$. On a en effet :

$$\widetilde{Y}_i = \sum_{j=0}^{\sqrt{p}-1} M_{i,j} \times \widetilde{X}_i, \qquad (0 \le i < \sqrt{p})$$

On peut donc utiliser la procédure suivante :

1. Pour tout $0 \le j < \sqrt{p}$, les processus de la colonne j collaborent pour que chacun obtienne $\widetilde{X}_i$.
2. Pour tout (i,j), chaque processus calcule sa contribution $M_{i,j} \times \widetilde{X}_i$
3. Pour tout $0 \le i < \sqrt{p}$, les processus de la ligne i collaborent pour calculer la somme de leurs contributions, obtenir ainsi $\widetilde{Y}_i$, et disperser le résultat entre eux.

Pour la première étape, où les processus de chaque colonne doivent mutualiser leurs petites tranches pour former la grosse tranche dont ils ont tous besoin, on fait un « all-gather ». Pour la troisième étape, où les processus de chaque ligne doivent sommer leur contribution, puis se la dispatcher, on fait un « reduce-scatter ». Ces deux opérations collectives concernent des tableaux de taille $n/\sqrt{p}$, et $\sqrt{p}$ processus y participent.

Le petit détail, c'est qu'au début les $\widetilde{X}_i$ sont répartis sur les colonnes tandis qu'à la fin les $\widetilde{Y}_i$ sont répartis sur les lignes. Ceci peut se régler avec une dernière étape :

4. Pour tout(i,j), les processus (i,j) et (j,i) effectuent une communication point-à-point pour échanger leur petite tranche de y.

La quantité de calculs est la même que précédemment : il s'agit de calculer le produit d'une matrice carrée de taille $n/\sqrt{p}$ et d'un vecteur de la bonne taille. Ceci nécessite n^2/p opérations.

Les deux opérations collectives coûtent chacune la même chose, à savoir environ $\alpha \log \sqrt{p} + (\sqrt{p}-1)\frac{n}{p}\beta$. Enfin, la dernière phase de « transposition », nécessite que chaque processus envoie et reçoive un message de taille n/p, donc ceci ajoute un temps $\alpha + n/p \cdot \beta$. Au final, on trouve donc :

$$T = \underbrace{\frac{n^2}{p}}_{\text{calcul}} + \underbrace{(1+\log_2 p)\,\alpha + (2\sqrt{p}-1)\,\frac{n}{p}\beta}_{\text{communication}}$$

Et en simplifiant un tout petit peu on obtient :

$$T \le \frac{n^2}{p} + \alpha \log_2 p + 2\frac{n}{\sqrt{p}}\beta + \mathcal{O}(1)$$

L'intérêt principal de cette méthode est que la quantité de communications est plus faible que dans les deux premières stratégies (elle *décroît* avec p).

Il est de plus possible possible d'utiliser plus de n processeurs. La limite naturelle est n^2 : on ne peut pas donner moins qu'un seul coefficient de la matrice à un processeur. Mais on peut répartir la matrice en n^2 si on a envie. Le petit détail, c'est qu'on ne peut pas répartir le vecteur x en plus que n^2. En pratique ce n'est pas un très gros problème. On peut envisager que seuls certains processus possèdent des tranches de x, ou bien que plusieurs processus possèdent la même tranche. Ce n'est pas très gênant.

Si on dispose de $p = n^2$ processeurs, le temps de calcul par processus tombe à $\mathcal{O}(1)$, de même que le terme de volume des communications. Tout ce qui reste est le terme de latence, qui est de l'ordre de $\mathcal{O}(\log n)$. On se heurte alors à une borne inférieure, car chaque tranche de y dépend de toutes les tranches de x, or il faut $\log_2 n$ messages successifs pour que tout ce petit monde puisse contribuer.